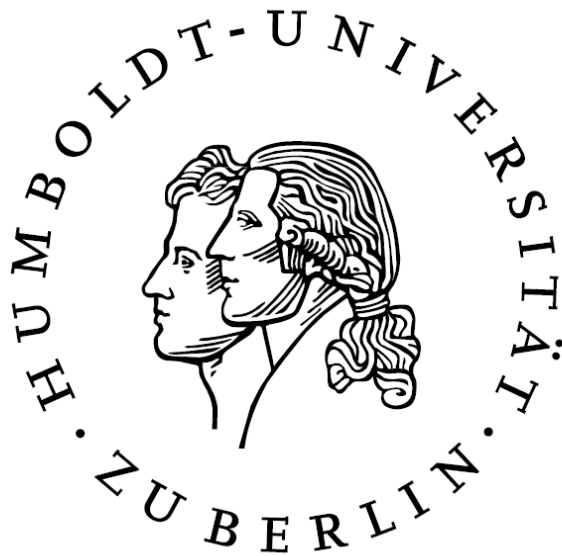


Studienarbeit

Implementierung zweier Algorithmen zur Abstraktion von Petri-Netzen

Daniel Janusz
06.04.2008



Humboldt-Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II

Institut für Informatik

Betreuer: Christian Stahl

Inhaltsverzeichnis

1 Einleitung	5
2 Grundlagen	7
2.1 Petrinetze.....	7
2.2 Erreichbarkeitsgraphen	8
2.3 IO-Netze	9
2.4 IO-Graphen.....	10
2.5 Beschränktheit und Deadlocks.....	11
2.6 Inäquivalente Knoten	12
2.7 Parallele, IO-äquivalente und schaltabhängige Pfade.....	13
2.8 IO-Pfad, Trace und IO-Trace	14
2.9 IOT-State-Äquivalenz.....	15
2.10 IOT-Failure-Äquivalenz.....	15
3 Algorithmen	17
Parallele Komposition	17
Regel 1	18
Regel 2	19
Regel 3	19
Regel 4	20
Regel 5	21
Regel 6	21
Regel 7	22
Regel 8	23
Regel 9	23
Regel 10-5 (modifizierte Regel 5).....	24
Regel 10-6 (modifizierte Regel 6).....	24
Regel 10-7 (modifizierte Regel 7).....	25
4 Werkzeug	27
5 Fallstudie	29
6 Fazit	33
7 Anhang	35
8 Literatur	37

1 Einleitung

Da heutige *Systeme* sehr groß und komplex sind, gibt es computergestützte Methoden zur Überprüfung von Systemeigenschaften. Ein System besteht aus interagierenden Gegenständen. Das Verhalten eines Systems, der *Systemzustand*, kann sich ändern. Zum Beispiel ist eine Ampel ein System mit dem sich zeitlich ändernden Systemzustand: Rotes Licht an und Rotes Licht aus. Der Systemzustand kann mit einem *Modell* des Systems untersucht werden. Unter einem Modell verstehen wir ein vereinfachtes, abstraktes Abbild des Systems, das die interessierenden Eigenschaften so genau wie nötig und möglich wiedergibt. *Petrinetze* sind Modelle zur Beschreibung von Systemen [Sta90]. Zum Beispiel kann das Verhalten einer Ampelschaltung mittels eines Petrinetzes beschrieben und untersucht werden. Ob ein System bestimmte Eigenschaften hat, kann anhand seines Systemmodells überprüft werden.

Eine formale *Verifikation* ist ein mathematischer Beweis, der ein Modell auf eine bestimmte Eigenschaft hin überprüft. Wird eine Verifikation mittels computergestützter Verfahren durchgeführt, sprechen wir von *computergestützter Verifikation*. Eine konkrete Art der computergestützten Verifikation ist das *Modelchecking*. Dabei werden einem Programm, dem Modelchecker, ein Modell und temporallogische Formeln übergeben. Die temporallogischen Formeln sind die *Systemspezifikation* und sie beschreiben formal die Systemeigenschaften, die überprüft werden sollen. Der Modelchecker analysiert nun das Modell und liefert als Ergebnis entweder eine Korrektheitsaussage, wenn das Modell die temporallogischen Formeln erfüllt, oder einen Systemzustand, der die Systemspezifikation verletzt. Mittels des Modells können wir alle erreichbaren Systemzustände berechnen. Haben wir den *Zustandsraum* eines Modells, brauchen wir nur noch überprüfen, ob die Systemspezifikation in jedem Zustand erfüllt ist. Bei der Berechnung des Zustandsraumes stoßen wir schnell an die Grenzen der Speicherkapazität heutiger Computer. Der Zustandsraum eines Systems wächst in Abhängigkeit von der Anzahl der Systemvariablen exponentiell. Dieses extreme Anwachsen wird als *Zustandsraumexplosion* bezeichnet. Die computergestützte Verifikation muss im Allgemeinen *Reduktionsmethoden* zur Abschwächung dieses Problems umfassen. Das Ziel ist, ein gegebenes Modell M in ein Modell M' mit einem kleineren Zustandsraum zu überführen, ohne die interessanten Eigenschaften des Ausgangsmodells zu verlieren.

In dieser Arbeit werden wir uns mit einer speziellen Verifikationsmethode beschäftigen, der *kompositionalen Verifikation*. Häufig besteht ein System aus vielen Komponenten, die jede für sich eine abgeschlossene Einheit bilden und über eine Schnittstelle miteinander verbunden sind.

Die Idee ist, erst den Zustandsraum jeder einzelnen Komponente zu berechnen und zu reduzieren und dann die kleineren Zustandsräume nacheinander zu dem Gesamtsystem zusammenzufügen. Dabei wird das komponierte System nach jedem Kompositionsschritt nochmals reduziert.

Der Ausgangspunkt unserer Betrachtungen ist das Papier *Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules* von Juan et al. [JTM98]. In diesem Papier wurde eine spezielle kompositionale Verifikation erarbeitet. Hierbei werden die Komponenten eines Systems mittels *Input-/Output-Netzen* modelliert. Diese sind Petrinetze, die um eine Schnittstelle erweitert wurden. Außerdem werden Methoden zur Reduktion der Zustandsraumexplosion samt Korrektheitsbeweisen präsentiert. Die Reduktionsmethoden erhalten die Erreichbarkeit von Systemzuständen, insbesondere solche, in denen das System verklemmt und somit zum Stillstand kommt. Weiterhin wird eine Fallstudie vorgestellt, in der, anhand von Input-/Output-Netzen für das Beispiel der speisenden Philosophen, die Leistungsfähigkeit der vorgeschlagenen Verifikation gezeigt wird. Leider ist das für die Fallstudie verwendete Werkzeug namens IOTA nicht verfügbar, so dass die gezeigten Resultate nicht nachvollziehbar sind.

Ziel dieser Arbeit ist es, das Werkzeug neu zu implementieren, die Ergebnisse des Papiers zu überprüfen und die Stärken bzw. Schwächen dieser Verifikationsmethode zu erarbeiten. Hierfür wurde der Modelchecker *ioKoCheck* implementiert, der die Reduktionstechniken aus dem Papier von Juan et al. implementiert.

Der weitere Verlauf der Arbeit gliedert sich wie folgt: In Kapitel 2 werden wir die theoretischen Grundlagen der kompositionalen Verifikation nach Juan et al. einführen. Anschließend werden in Kapitel 3 die einzelnen Reduktionsmethoden beschrieben. Kapitel 4 enthält Einzelheiten zum Werkzeug *ioKoCheck*. Danach wird im fünften Kapitel die Verifikationsmethode anhand zweier Fallstudien validiert. Wir vergleichen die Ergebnisse für das Beispiel der speisenden Philosophen mit denen des Papiers und zeigen Ergebnisse eines zweiten Beispiels. In Kapitel 6 werden die erzielten Resultate diskutiert. Wir erläutern Stärken, Schwächen und den praktische Einsatz dieser Technik. Zum Abschluss ziehen wir ein Fazit zur Programmierung von *ioKoCheck*.

2 Grundlagen

In den folgenden Definitionen beschränken wir uns auf die Begriffe, die für das Verständnis von kompositionaler Verifikation nach Juan et al. nötig sind. Dabei verwenden wir das Symbol \mathbb{N} für die natürlichen Zahlen und das Symbol \mathbb{Z} für die Ganzen Zahlen.

2.1 Petrinetze

Ein *Petrinetz* ist eine formale Methode zur Beschreibung und zur Analyse von Systemen. Es gibt viele Varianten; eine für unsere Betrachtungen passende finden wir bei Starke [Sta90]. Ein Petrinetz ist ein Quintupel gerichteter Graph $N = (P, T, F, W, m_0)$ von:

- einer endlichen Menge P von *Plätzen*,
- einer endlichen Menge T von *Transitionen* mit $P \cap T = \{\}$,
- einer Menge von *Bögen* $F \subseteq (P \times T) \cup (T \times P)$,
- einer Abbildung $W: f \rightarrow \mathbb{N} \setminus \{0\}, f \in F$, den *Bogenvielfachheiten* und
- der *Anfangsmarkierung* $m_0: P \rightarrow \mathbb{N}$.

Der Vorteil der Verwendung von Petrinetzen bei der Modellierung von Systemen liegt in der leicht verständlichen graphischen Darstellung. Einen Platz fassen wir als eine Systembedingung auf, er wird durch einen Kreis repräsentiert. Ein Platz kann Marken in Form von schwarzen Punkten enthalten. Diese zeigen den aktuellen Zustand der Systembedingung an. Transitionen stehen für Aktionen des Systems, welche Systembedingungen verändern können. Transitionen werden graphisch durch Rechtecke dargestellt. Ein Bogen wird durch eine gerichtete Kante angezeigt, diese verbindet immer genau einen Platz und eine Transition. Zum einen werden durch Bögen alle Aktionen zugeordnet, die von einem Systemzustand aus durchführbar sind; zum anderen zeigen sie alle Systembedingungen an, die sich durch eine Aktion verändern.

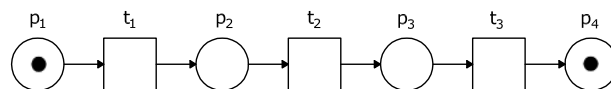


Abbildung 1: Petrinetz mit 4 Plätzen und 3 Transitionen

Abbildung 1 zeigt ein Petrinetz. In den Plätzen p_1 und p_4 ist je eine Marke zu sehen.

Sei $N = (P, T, F, W, m_0)$ ein Petrinetz. Jede Funktion $m: P \rightarrow \mathbb{N}$ wird als *Markierung* von N bezeichnet.

Demnach beschreibt eine Petrinetzmarkierung einen konkreten Zustand aller Systembedingungen und eine Anfangsmarkierung m_0 beschreibt den Anfangszustand eines Systems. Das Netz in der Abbildung 1 hat die Markierung $m = \{(p_1, 1), (p_2, 0), (p_3, 0), (p_4, 1)\}$.

Sei m eine Markierung im Petrinetz $N = (P, T, F, W, m_0)$. Eine Funktion $m|_{TP}: TP \rightarrow \mathbb{N}$ heißt *Teilmarkierung* von N , wenn gilt:

- $TP \subseteq P$ und
- $\forall p \in TP: m|_{TP}(p) = m(p)$.

$m|_{TP}$ repräsentiert einen Teil der Markierung m , daher wird $m|_{TP}$ Teilmarkierung in m genannt. $m|_{Teil} = \{(p_1, 1), (p_2, 0)\}$ mit $Teil = \{p_1, p_2\}$ ist eine Teilmarkierung des Beispielnetzes in Abbildung 1.

Das Ausführen einer Aktion im System wird als *Schalten* einer Transition modelliert. Für jede Aktion gibt es bestimmte Systembedingungen unter denen sie durchführbar ist. Damit eine Transition schalten kann, muss sie *aktiviert* sein. Eine Transition $t \in T$ ist bei der Markierung m aktiviert, wenn:

- $\forall p \in P: f[p, t] \in F \rightarrow m(p) \geq W(f)$.

Durch das Schalten einer bei m aktivierten Transition $t \in T$ wird die Markierung m in die Markierung m' überführt. Den Übergang von m zu m' notieren wir wie folgt: $m \xrightarrow{t} m'$. Dabei werden Marken von Plätzen entfernt, was als konsumieren bezeichnet wird, und es werden Marken zu Plätzen hinzugefügt, also produziert. Für alle Plätze $p \in P$ gilt:

- $$m'(p) := \begin{cases} m(p) - W(f), & \text{falls } f[p, t] \in F \\ m(p) + W(f), & \text{falls } f[t, p] \in F \\ m(p) - W(f) + W(a), & \text{falls } f[p, t] \in F \text{ und } a[t, p] \in F \\ m(p), & \text{sonst} \end{cases}$$

Es schaltet immer nur eine Transition. Zum Beispiel ist in Abbildung 1 die Transition t_1 aktiviert, während t_2 nicht aktiviert ist.

2.2 Erreichbarkeitsgraphen

Von der Anfangsmarkierung beginnend können wir durch das nacheinander Schalten jeweils aktivierter Transitionen $m_0 \xrightarrow{t_0} m_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} m_n \xrightarrow{t_n} m'$ neue Markierungen eines Petrinetzes erreichen. Jede dieser Markierung wird als *erreichbare Markierung* bezeichnet.

Zu jedem Petrinetz $N = (P, T, F, W, m_0)$ kann ein *Erreichbarkeitsgraph* $EG(N) = (V, E)$ erzeugt werden, mit:

- einer Menge V von *Knoten* und
- einer Menge $E \subseteq V \times V$ von *gerichteten Kanten*.

Die in N erreichbaren Markierungen sind die Knoten V von $EG(N)$. Eine Kante $e \in E$ beschreibt das Schalten einer Transition $m \xrightarrow{t} m'$ und wird mit t , dem Namen der Transition, beschriftet. Der Knoten, der die Anfangsmarkierung von N repräsentiert, wird Startknoten von $EG(N)$ genannt. Die Menge $\sigma = \{v_1, e_1, \dots, v_{n-1}, e_{n-1}, v_n\}$ heißt *Pfad* vom Knoten v_1 zu v_n in $EG(N) = (V, E)$ wenn:

- $v_1, \dots, v_n \in V$,
- $e_1, \dots, e_n \in E$ und
- e_x ist Kante von v_x zu v_{x+1} mit $1 \leq x < n$.

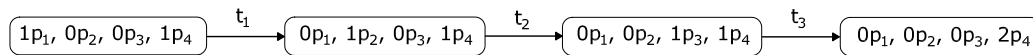


Abbildung 2: Erreichbarkeitsgraph des Petrinetzes in Abbildung 1

In der Abbildung 2 sehen ist der zur Abbildung 1 gehörige Erreichbarkeitsgraph zu sehen. Dieser enthält unter anderem den Pfad

$$\sigma = \{ \{(p_1, 0), (p_2, 1), (p_3, 0), (p_4, 1)\}, t_2, \{(p_1, 0), (p_2, 0), (p_3, 1), (p_4, 1)\} \}.$$

2.3 IO-Netze

Mit Petrinetzen haben wir bis jetzt eine allgemein bekannte Modellierungstechnik kennengelernt. Für die kompositionale Verifikation nach Juan et al. sind diese aber noch nicht praktikabel. Wir benötigen eine Technik, die es uns erlaubt, modulare Systeme abzubilden. Ein dafür verfeinertes Netzmodell sind die *Input/Output-Netze*, zukünftig *IO-Netze* genannt. Ein IO-Netz kombiniert ein Petrinetz mit einer Schnittstelle und wird als gerichteter Graph $N_{io} = (P, T, F, W, m_0, P_{io}, F_{io}, W_{io})$ definiert, mit:

- $N = (P, T, F, W, m_0)$ ist ein Petrinetz,
- einer endlichen Menge P_{io} von *Schnittstellenplätzen* mit $P_{io} \cap P = \{\}$,
- einer Menge von *Schnittstellenbögen* $F_{io} \subseteq (P_{io} \times T) \cup (T \times P_{io})$, und
- einer Abbildung $W_{io}: f \rightarrow \mathbb{N} \setminus \{0\}, f \in F_{io}$, den *Bogenvielfachheiten der Schnittstellenbögen*.

Die Schnittstellenkomponenten werden mit *IO-Plätzen* und *IO-Bögen* abgekürzt. Bei den Plätzen des Petrinetzes P sprechen wir von *internen Plätzen*. Ebenso bezeichnen wir die Bögen von P als *interne Bögen*. Durch IO-Plätze können IO-Netze miteinander verbunden und zu einem

Gesamtnetz komponiert werden. Viel interessanter ist aber, dass durch Partitionieren der Plätze eines Petrinetzes $N = (P, T, F, W, m_0)$ in zwei disjunkte Mengen von internen Plätzen P_{intern} und IO-Plätzen P_{io} mit $P = P_{\text{io}} \cup P_{\text{intern}}$ und $P_{\text{io}} \cap P_{\text{intern}} = \{\}$, N in das IO-Netz $N_{\text{io}} = (P_{\text{intern}}, T, F_{\text{intern}}, W_{\text{intern}}, m_0, P_{\text{io}}, F_{\text{io}}, W_{\text{io}})$ überführt werden kann. Dabei teilt sich die Menge der Schnittstellenbögen F von N durch die Festlegung der IO-Plätze automatisch in IO-Bögen und interne Bögen auf. Durch die IO-Bögen und internen Bögen kann letztlich auch die Funktion für die Bogenvielfachheiten W in W_{intern} und W_{io} gegliedert werden.

Für die IO-Plätze, die zwei IO-Netze miteinander verbinden, gilt, dass sie in beiden IO-Netzen enthalten sind. Aber in der Schnittstelle eines IO-Netzes befinden sich keine Informationen, ob das IO-Netz mit anderen IO-Netzen verbunden ist, deshalb wird bei der Analyse und Reduktion eines IO-Netzes nur die Anfangsmarkierung der internen Plätze berücksichtigt. Bei der Komposition zweier IO-Netze werden ihre gemeinsamen IO-Plätze in interne Plätze gewandelt. Erst jetzt wird die Anfangsmarkierung dieser Plätze berücksichtigt.

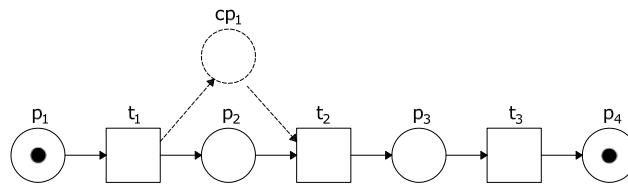


Abbildung 3: IO-Netz mit 4 internen Plätzen, 1 IO-Platz und 3 Transitionen

In Abbildung 3 ist das Petrinetz aus Abbildung 1 zu erkennen, das um den IO-Platz cp_1 und zwei IO-Bögen erweitert wurde und damit zu einem IO-Netz wird.

2.4 IO-Graphen

Nun können wir das eigentliche Analysemodell einführen: die *IO-Graphen*. Zu jedem IO-Netz kann ein IO-Graphen erzeugt werden. Ähnlich wie beim IO-Netz so besteht auch der IO-Graph aus zwei Teilen, dem Erreichbarkeitsraum/Erreichbarkeitsgraph und einer Schnittstelle.

Formal ist der zum IO-Netz N_{io} gehörende IO-Graph ein 5-Tupel

$IOG(N_{\text{io}}) = (V, E, IEL, OEL, BFnonstable)$ mit:

- $EG(N) = (V, E)$ dem Erreichbarkeitsgraph, des in N_{io} enthaltenen Petrinetzes N ,
- einer endlichen Menge $IEL: E \rightarrow 2^{(P_{\text{io}} \rightarrow \mathbb{Z}^-)}$, den *Input-Edge-Labels*,
- einer endlichen Menge $OEL: E \rightarrow 2^{(P_{\text{io}} \rightarrow \mathbb{Z}^+)}$, den *Output-Edge-Labels* und
- der Wahrheitsfunktion $BFnonstable: V \rightarrow \{ON, OFF\}$.

Die *Input/Output-Edge-Labels*, kurz *IO-Edge-Labels*, zeigen Interaktionen mit der Schnittstelle an. Ein Input-Edge-Label steht für das Konsumieren und ein Output-Edge-Label für das Produzieren von Marken eines IO-Platzes durch eine geschaltete Transition. Edge-Labels sind Kantenmarkierungen. Sie werden in geschweiften Klammern an die zugehörige Kante geschrieben. Eine Kante ohne IO-Edge-Label heißt interne Kante. Für das Input-Edge-Label einer Kante e schreiben wir $e.IEL$ und für das Output-Edge-Label von e schreiben wir $e.OEL$. Außerdem gibt es die Wahrheitsfunktion $BF_{nstable}$. Sie zeigt an, ob ein Knoten *IOTstable* ist. Eine Markierung m ist *IOTstable*, wenn der zu m gehörende Knoten im Erreichbarkeitsgraph keine ausgehende Kante ohne Input-Edge-Label hat. Wenn wir von einem Pfad im IO-Graph sprechen, ist damit ein Pfad in $EG(N) = (V, E)$ gemeint.

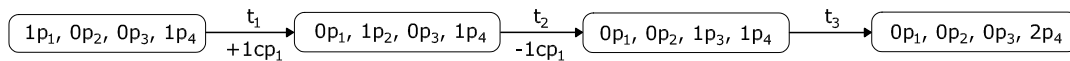


Abbildung 4: IO-Graph des IO-Netzes in Abbildung 3

Unser Beispiel in Abbildung 4 zeigt den IO-Graph des IO-Netzes aus Abbildung 3. Der einzige Unterschied zu dem Erreichbarkeitsgraph aus Abbildung 2 liegt in den beiden IO-Edge-Labels $+1cp_1$ und $-1cp_1$. In den folgenden Beispielen werden wir eine vereinfachte Markierungsschreibweise benutzen, wobei wir alle mit Null belegten Plätze weglassen. Da die Markierungen dann wie Teilmarkierungen aussehen, werden wir Teilmarkierungen ab jetzt immer explizit benennen. Zusätzlich werden wir auch die Kantenbeschriftung mit den Transitionsnamen weglassen. Die Vereinfachung von Abbildung 4 ist in Abbildung 5 zu sehen.

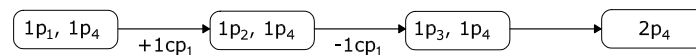


Abbildung 5: Vereinfachter IO-Graph des IO-Netzes in Abbildung 3

2.5 Beschränktheit und Deadlocks

Ein IO-Netz N_{io} heißt genau dann *beschränkt*, wenn:

- der zu N_{io} gehörende IO-Graph $IOG(N_{io})$ einen endlichen Zustandsraum hat.

Das bedeutet: die Menge erreichbarer Markierungen des IO-Netzes ist endlich. Ist sie es nicht, handelt es sich um ein unbeschränktes Netz. Der IO-Graph in Abbildung 5 hat vier Zustände. Damit ist der Zustandsraum des IO-Graphen endlich und das IO-Netz in Abbildung 3 ist beschränkt.

Eine Markierung m wird genau dann *Deadlock* des IO-Graph IOG genannt, wenn:

- m einen erreichbaren Knoten v von IOG repräsentiert und
- v keine ausgehenden Kanten hat.

Die Markierung $m = \{(p_4, 2)\}$ ist ein *Deadlock* des in Abbildung 5 gezeigten IO-Graphen. Bei einem *Deadlock* ist das System verklemmt und kann seinen Zustand nicht mehr verändern. Ein *Deadlock* ist immer *IOTstable*.

2.6 Inäquivalente Knoten

Zwei Knoten v_1 und v_2 eines IO-Graph IOG sind genau dann *inäquivalent* zueinander, wenn:

- v_1 mindestens eine eingehende Kante hat,
- für jede eingehende Kante e_1 von v_1 gilt:
 - entweder
 - e_1 hat keine IO-Edge-Labels $e_1.IEL = \{\}$, und e_1 hat v_2 als Ausgangsknoten,
 - oder
 - e_1 hat IO-Edge-Labels $e_1.IEL \neq \{\}$, und es gibt eine eingehende Kante e_2 von v_2 , die identische IO-Edge-Labels $e_1.IEL = e_2.IEL$ und den selben Ausgangsknoten wie e_1 hat,
- und vice versa für jede eingehende Kante von v_2 .

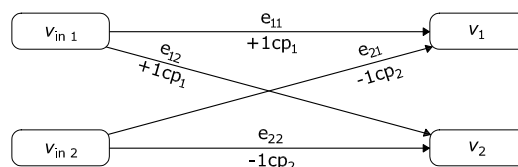


Abbildung 6: Inäquivalente Knoten in einem IO-Graph

Abbildung 6 zeigt einen IO-Graph mit den Knoten v_1 und v_2 . v_1 hat die eingehenden Kanten e_{11} und e_{21} . v_2 hat die eingehenden Kanten e_{12} und e_{22} . Da für die Kanten gilt: $e_{11}.IEL = e_{12}.IEL$ und $e_{21}.IEL = e_{22}.IEL$, sind die Knoten v_1 und v_2 zueinander inäquivalent.

2.7 Parallele, IO-äquivalente und schaltabhängige Pfade

Zwei Pfade eines IO-Graphen heißen *parallel*, wenn für beide Pfade gilt:

- sie haben identische Startknoten,
- sie haben identische Endknoten und
- sie haben keine gemeinsamen Kanten.

Die Pfade $\sigma_1 = \{v_1, e_1, v_2, e_2, v_3, e_3, v_4\}$ und $\sigma_2 = \{v_1, e_4, v_4\}$ sind parallele Pfade des IO-Graphen in Abbildung 7.

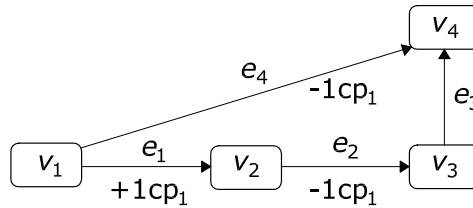


Abbildung 7: Parallele Pfade

Seien L_1 und L_2 IO-Edge-Labels für IO-Plätze P_{L_1} und P_{L_2} . Die *Summe* L der IO-Edge-Labels L_1 und L_2 : $L = L_1 + L_2$ ist ein IO-Edge-Label für die IO-Plätze $P_L = P_{L_1} \cup P_{L_2}$, sodass:

- $\forall p \in P_L: L(p) = L_1(p) + L_2(p)$, mit $L_1(p) = 0$, wenn $p \notin P_{L_1}$, und $L_2(p) = 0$, wenn $p \notin P_{L_2}$.

IO-Edge-Labels eines Pfades können zu *IO-Path-Labels* zusammengefasst werden. Das *Input-Path-Label* eines Pfades σ ist die Summe der Input-Edge-Labels aller Kanten von σ und das *Output-Path-Label* eines Pfades σ ist die Summe der Output-Edge-Labels aller Kanten von σ . In Abbildung 7 ist das IO-Path-Label des Pfades $\sigma = \{v_1, e_1, v_2, e_2, v_3, e_3, v_4\}$: $\sigma.IPL + \sigma.OPL = \{+1cp_1\} + \{-1cp_1\} = \{+1cp_1 -1cp_1\}$. Dabei steht $\sigma.IPL$ für das Input-Path-Label von σ und $\sigma.OPL$ für das Output-Path-Label von σ .

Zwei Pfade σ_1 und σ_2 eines IO-Graphen heißen *IO-äquivalent*, wenn:

- $\sigma_1.IPL + \sigma_1.OPL = \sigma_2.IPL + \sigma_2.OPL$.

In dem IO-Graph in Abbildung 7 ist der Pfad $\sigma_1 = \{v_1, e_4, v_4\}$ IO-äquivalent zum Pfad $\sigma_2 = \{v_2, e_2, v_3, e_3, v_4\}$.

Ein Pfad $\sigma_n = \{v_1, e_1, \dots, e_{n-1}, v_n\}$ eines IO-Graphen heißt *schaltabhängig* vom Pfad $\sigma_s = \{v_1, e_s, v_s\}$ mit der Kante e_s , wenn:

- $(-e_s.IEL) \geq (-\sigma.IPL)$.

Der Pfad $\sigma_1 = \{v_1, e_1, v_2, e_2, v_3, e_3, v_4\}$ in Abbildung 7 ist schaltabhängig vom Pfad $\sigma_2 = \{v_1, e_4, v_4\}$.

2.8 IO-Pfad, Trace und IO-Trace

Mit Hilfe von *IO-Pfaden* im IO-Graphen lassen sich konkrete Systemverhalten beschreiben. Ein IO-Pfad besteht aus einer Reihe von IO-Edge-Labels und einem Endzustand. Die IO-Kantenmarkierungen einer Komponente zeigen das für andere Komponenten sichtbare Verhalten an. Demnach zeigen IO-Pfade das Außenverhalten bei der Überführung vom Anfangszustand der Komponente zum Endzustand des Pfades an. Auch eine leere Reihe von IO-Edge-Labels ist zulässig. Damit gibt es zu jeder Markierung m eines IO-Graphen mindestens einen IO-Pfad, der von der Anfangsmarkierung m_0 zu m führt. Unser Beispielgraph in Abbildung 4 hat folgende IO-Pfade:

- $io\sigma_1 = null(1p_1, 1p_4)$,
- $io\sigma_2 = \{+1cp_1\}(1p_2, 1p_4)$,
- $io\sigma_3 = \{+1cp_1\} \{-1cp_1\}(1p_3, 1p_4)$ und
- $io\sigma_4 = \{+1cp_1\} \{-1cp_1\} \{\}(2p_4)$.

Mit den Traces lernen wir eine erste Möglichkeit kennen, einen IO-Graph zu reduzieren. Ein Trace ist ein reduzierter IO-Pfad, bei dem sämtliche leeren IO-Edge-Labels $\{\}$ weggelassen werden. Dadurch wird internes Systemverhalten versteckt. Im Beispielgraph ist $t\sigma = \{+1cp_1\} \{-1cp_1\}(2p_4)$ der Trace, der aus dem IO-Pfad $io\sigma_4 = \{+1cp_1\} \{-1cp_1\} \{\}(2p_4)$ erzeugt werden kann.

Auch durch *IO-Traces* wird wieder Systemverhalten versteckt. Ein IO-Trace entsteht aus einem Trace, indem wir jeweils benachbarte IO-Edge-Labels nacheinander addieren, wobei das Input-Edge-Label des zweiten IO-Edge-Labels leer sein muss. Ein Input-Edge-Label zeigt an, dass zum Schalten einer Transition Marken von einem IO-Platz benötigt werden; da das zweite IO-Edge-Label kein Input-Edge-Label hat, kann diese Systemaktion ohne Rücksicht auf die Gegebenheiten der verbundenen IO-Netze durchgeführt werden, wenn sie intern aktiviert ist. Die Information der Output-Edge-Labels wird durch IO-Traces gebündelt und, statt in mehreren, mit einer einzigen Systemaktion an die verbundenen IO-Netze weitergegeben. Zum Beispiel wird der Trace $t\sigma = \{+1cp_1\} \{-1cp_1\}(2p_4)$ zum IO-Trace $iot\sigma = \{+1cp_1-1cp_1\}(2p_4)$.

2.9 IOT-State-Äquivalenz

Sei IOG ein IO-Graph. Wir schreiben: $v_0 \xrightarrow{iot} v_n$ wenn:

- es gibt einen Pfad $\sigma = \{v_0, e_0, \dots, e_{n-1}, v_n\}$ von v_0 zu v_n in IOG ,
- v_0 ist Startknoten von IOG und,
- iot ist der IO-Trace von σ .

(iot, v) heißt *IO-Trace-State* oder kurz *IOT-State* von IOG , wenn:

- $v_0 \xrightarrow{iot} v$.

$IOTstate(IOG)$ ist die Menge aller IOT-States eines IO-Graphen IOG . Zwei IO-Graphen IOG_1 und IOG_2 werden genau dann *IOT-State-äquivalent* genannt, wenn:

- die Mengen der auf IOG_1 und IOG_2 erzeugbarer IOT-States gleich sind:
 $IOTstate(IOG_1) = IOTstate(IOG_2)$.

Zwei IOT-State-äquivalente IO-Graphen können gegeneinander ausgetauscht werden, ohne erreichbare Markierungen zu verlieren.

2.10 IOT-Failure-Äquivalenz

Das Tripel $(iot, s, s.IEL)$ heißt *IO-Trace-Failure* oder kurz *IOT-Failure* des IO-Graphen IOG , wenn:

- (iot, s) ein IOT-State ist und
- s *IOTstable* ist.

$IOTfailure(IOG)$ ist die Menge aller IOT-Failures des IO-Graphen IOG . Zwei IO-Graphen IOG_1 und IOG_2 werden genau dann *IOT-Failure-äquivalent* genannt, wenn:

- die Mengen der auf IOG_1 und IOG_2 erzeugbarer IOT-Failures gleich sind:
 $IOTfailure(IOG_1) = IOTfailure(IOG_2)$.

Zwei IOT-Failure-äquivalente IO-Graphen können gegeneinander ausgetauscht werden, ohne Deadlocks zu verlieren.

3 Algorithmen

Die folgenden Algorithmen werden in einer Pseudocodenotation präsentiert. Die Grundlage für die Algorithmen bilden die Methoden in dem Papier von Juan et al. [JTM98, Kap. 8. S.945 ff. + Anhang A. S.961 ff.]. Als erstes stellen wir nun die Funktion Parallele Komposition zur Komposition zweier IO-Graphen vor.

Seien NA und NB zwei endliche IO-Netze, GA der IO-Graph zu NA , GB der IO-Graph zu NB und $M_0(P_{CMP})$ eine Markierung von gemeinsamen IO-Plätzen P_{CMP} . Achtung: P_{CMP} kann auch leer sein. Dann ist $ION = NA \cup NB \cup M_0(P_{CMP}) = (P, T, F, W, m_0, P_{io}, F_{io}, W_{io})$ ein komponiertes IO-Netz mit den in NA und NB enthaltenen IO-Plätzen P_{CMP} . Der komponierte Graph $IOG(ION) = GA \parallel GB \parallel M_0(P_{CMP}) = (V, E, IEL, OEL, BFnonstable)$ zu ION wird wie folgt erzeugt:

Parallele Komposition

begin

- (1) $m_0(ION) = m_0(NA) \cup m_0(NB) \cup m_0(P_{CMP})$, die neue Anfangsmarkierung $m_0(ION)$ ergibt sich aus den Anfangsmarkierungen von NA und NB vereinigt mit der nun festgelegten Anfangsmarkierung der gemeinsamen IO-Plätze,
- (2) setze Knotenliste $VList = \{ m_0(ION) \}$ und **do**
 - entferne einen Knoten v mit der Markierung m_v vom Anfang der Knotenliste $VList$,
 - für jede Kante e der IO-Graphen GA und GB mit der zum Startknoten von e gehörenden Markierung $m_{e-start}$ und der zum Zielknoten von e gehörenden Markierung m_{e-ziel} :
 - wenn e bei m_v aktiviert ist (d.h. die Markierung $m_{e-start}$ ist eine Teilmarkierung in m_v und die Anzahl der Marken auf den gemeinsamen IO-Plätzen P_{CMP} ist größer gleich der Markenanzahl, die durch die Input-Edge-Label $e.IEL$ von ihnen gefordert wird)
 - dann entsteht die neue Markierung m_{neu} durch:
 - setze $m_{neu} = m_v$,
 - ersetze die in m_{neu} enthaltenen Teilmarkierung $m_{e-start}$ durch m_{e-ziel} und
 - ändere die durch die gemeinsamen IO-Plätzen P_{CMP} erzeugte Teilmarkierung m_{CMP} entsprechend der Input-/Output-Edge-Labels, die P_{CMP} betreffen.
 - wenn es bereits einen Knoten v_{alt} mit der Markierung m_{alt} in der Knotenmenge V von IOG gibt, sodass:
 - $m_{alt} = m_{neu}$,
 - dann:
 - füge die neue Kante $e_{neu} = (v, v_{alt})$ zur Kantenmenge E von IOG hinzu,
 - sonst:
 - erzeuge einen neuen Knoten k ,
 - weise k die Markierung m_{neu} zu,
 - füge k in $VList$ ein,
 - füge k in die Knotenmenge V von IOG ein und
 - füge die neue Kante $e_{neu} = (v, k)$ in die Kantenmenge E von IOG ein.
 - setze für e_{neu} die IO-Edge-Labels wie folgt:
 - $e_{neu}.IEL = e.IEL$,
 - $e_{neu}.OEL = e.OEL$ und
 - entferne alle IO-Edge-Labels von e_{neu} , die mit den gemeinsamen IO-Plätzen P_{CMP} interagieren.

until $VList = \emptyset$

- (3) für alle Knoten $v \in IOG$ mit der Markierung m_v :
- wenn es einen Knoten s im IO-Graph GA oder GB gibt, mit:
 - $BFnonstable(s) = ON$ und
 - s enthält eine Markierung, die Teilmarkierung in m_v ist,
 - dann:
 - $BFnonstable(v) = ON$,
 - sonst:
 - $BFnonstable(v) = OFF$.

end

Nun können IO-Graphen nacheinander zu einem Gesamtsystem komponiert werden. Dabei wächst dieses sehr schnell an und es kommt zur Zustandsraumexplosion. Daher kommen wir nun zu den Reduktionsmethoden, die den komponierten IO-Graph verkleinern, um der Zustandsraumexplosion entgegenzuwirken. Es handelt sich um 10 Methoden, im Folgenden als Regeln bezeichnet. Den Regeln wird jeweils ein IO-Graph übergeben, um ihn in einen IOT-State-äquivalenten oder in einen IOT-Failure-äquivalenten IO-Graphen umzuwandeln. Im neuen IO-Graph werden die Deadlocks oder alle erreichbaren Markierungen erhalten.

Die erste Regel überprüft einen IO-Graph auf parallele Kanten mit identischen IO-Edge-Labels. Eine der Kanten kann entfernt werden, wobei weder Deadlocks noch sonstige Markierungen verloren gehen.

Regel 1

begin

- (1) wenn für zwei Kanten e_1 und e_2 eines IO-Graph IOG gilt:
- e_1 und e_2 haben den gleichen Ausgangsknoten,
 - e_1 und e_2 haben den gleichen Zielknoten und
 - e_1 und e_2 haben die gleichen IO-Edge-Label,
- dann:
- entferne die Kante e_1 aus IOG .

end

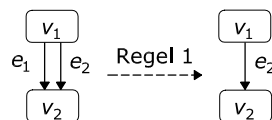


Abbildung 8: Regel 1 – Entfernen paralleler Kanten

Regel 2 durchsucht einen IO-Graph nach internen Schleifen. Eine interne Schleife ist ein Pfad $\sigma = \{v_1, e_1, \dots, v_{n-1}, e_{n-1}, v_n\}$, der nur aus internen Kanten besteht und bei dem der Startknoten des Pfades gleich dem Endknoten des Pfades ist ($v_1 = v_n$). Die Knoten einer Schleife enthalten keine Deadlocks, daher können sie zu einem Knoten verschmolzen werden. Es entsteht ein IOT-Failure-äquivalenten IO-Graph. Damit der IO-Graph auch IOT-State-äquivalent bleibt, müssen

die Knoten der Schleife zu einem Makroknoten verschmolzen werden. Ein Makroknoten ist ein Knoten, der mehrere Markierungen enthält.

Regel 2

begin

- (1) wenn n Knoten v_1, v_2, \dots, v_n über n Kanten e_1, e_2, \dots, e_n ($n > 0$) eine interne Schleife im IO-Graph IOG bilden, mit:
- loop $p = \{v_1 e_1 v_2 e_2 \dots v_n e_n v_1\}$ und
 - alle Kanten $e_i \in p$ ($1 \leq i \leq n$) sind interne Kanten ($e_i.IEL = \emptyset$ und $e_i.OEL = \emptyset$),
- dann:
- entferne die Kanten e_1, e_2, \dots, e_n aus IOG ,
 - verschmelze die Knoten v_1, v_2, \dots, v_n zu einem Makroknoten s durch:
 1. für jeden Knoten $v_i \in p$ ($1 \leq i \leq n$) mache:
 - jede eingehende Kante (v_x, v_i) wird zu (v_x, s) und
 - jede ausgehende Kante (v_i, v_y) wird zu (s, v_y) .
 2. wenn einer der Knoten v_1, v_2, \dots, v_n Startknoten von IOG ist, - dann:
 - wird s neuer Startknoten von IOG .
 3. die Markierungen der Knoten v_1, v_2, \dots, v_n werden in s eingefügt.
 - entferne die Knoten v_1, v_2, \dots, v_n aus IOG ,
 - setze $BFnonstable(s) = ON$ und
 - füge s in die Knotenmenge von IOG ein.

end

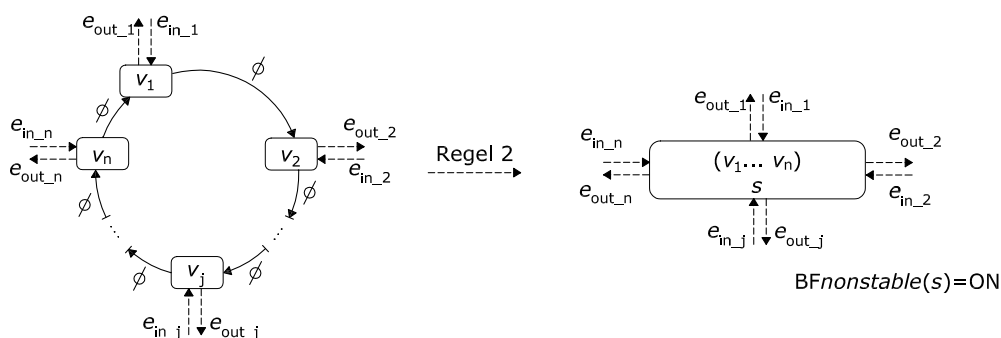


Abbildung 9: Regel 2 – Verschmelzen interner Schleifen

Die Regeln 3 und 4 verschmelzen inäquivalente Knoten eines IO-Graph zu Makroknoten. Regel 3 erzeugt aus dem Ausgangsgraphen einen IOT-State-äquivalenten IO-Graph und Regel 4 erzeugt einen IOT-Failure-äquivalenten IO-Graph.

Regel 3

begin

- (1) wenn für zwei Knoten v_1 und v_2 eines IO-Graph IOG gilt:
- v_1 und v_2 sind inäquivalent zueinander und
 - weder v_1 noch v_2 ist Startknoten von IOG ,
- dann:
- verschmelze die Knoten v_1 und v_2 zu einem Makroknoten s durch:
 1. für Knoten v_i ($1 \leq i \leq 2$) mache:
 - jede eingehende Kante (v_x, v_i) wird zu (v_x, s) und
 - jede ausgehende Kante (v_i, v_y) wird zu (s, v_y) .

2. die Markierungen der Knoten v_1 und v_2 werden in s eingefügt.

- entferne die Knoten v_1 und v_2 aus IOG ,
- setze $BFnonstable(s) = ON$ und
- füge s in die Knotenmenge von IOG ein.

end

Regel 4

begin

- (1) wenn für zwei Knoten v_1 und v_2 eines IO-Graph IOG gilt:
- v_1 und v_2 sind inäquivalent zueinander,
 - weder v_1 noch v_2 ist Startknoten von IOG und
 - entweder
 - v_1 und v_2 sind nicht $IOTstable$,
 - oder
 - $BFnonstable(v_1) = BFnonstable(v_2)$,
 - für jede ausgehende Kante e_1 von v_1 gibt es eine ausgehende Kante e_2 von v_2 mit:
 - e_1 und e_2 haben identische IO-Edge-Labels und
 - für jede ausgehende Kante e_2 von v_2 gibt es eine ausgehende Kante e_1 von v_1 mit:
 - e_1 und e_2 haben identische IO-Edge-Labels,

- dann:

- verschmelze die Knoten v_1 und v_2 zu einem Makroknoten s durch:
 1. für Knoten $v_i (1 \leq i \leq 2)$ mache:
 - jede eingehende Kante (v_x, v_i) wird zu (v_x, s) und
 - jede ausgehende Kante (v_i, v_y) wird zu (s, v_y) ,
 2. die Markierungen der Knoten v_1 und v_2 werden in s eingefügt,
- entferne die Knoten v_1 und v_2 aus IOG ,
- setze $BFnonstable(s) = ON$ und
- füge s in die Knotenmenge von IOG ein.

end

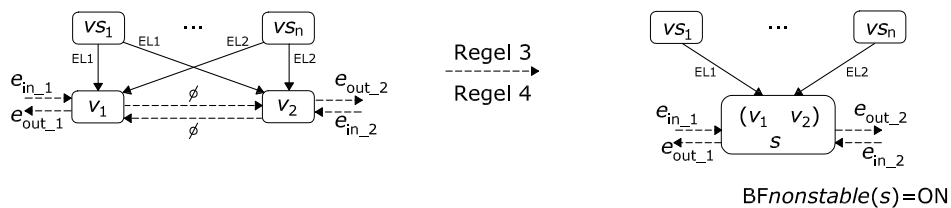


Abbildung 10: Regel 3 und Regel 4 – Verschmelzen inäquivalenter Knoten

Durch Anwenden von Regel 3 und Regel 4 entstehen parallele Kanten und Kanten von Knoten auf sich selbst, die *Selfloop-Kanten*. Diese werden mit Regel 1 und Regel 2 entfernt.

Mittels der Regel 5 wird ein Startknoten eines IO-Graph entfernt, der eine einzelne interne ausgehende Kante und keine eingehende Kante hat. Der entstehende IO-Graph ist IOT-Failure-äquivalent.

Regel 5

begin

- (1) wenn für den Startknoten v eines IO-Graph IOG gilt:
- v hat keine eingehenden Kanten,
 - v hat genau eine ausgehende Kante e ,
 - e ist interne Kante ($e.IEL = \emptyset$ und $e.OEL = \emptyset$) und
 - e ist keine Selfloop-Kante,
- dann:
- entferne den Knoten v aus IOG ,
 - entferne die Kante e aus IOG und
 - der Zielknoten von e wird neuer Startknoten von IOG .

end



Abbildung 11: Regel 5 – Entfernen redundanter Startknoten mit interner Kante

Regel 6 zeigt, unter welchen Bedingungen wir einen von zwei durch eine interne Kante verbundenen Knoten aus einem IO-Graph entfernen können, ohne dass ein Deadlock verloren geht. Es ist demnach eine IOT-Failure-äquivalente Umwandlung des IO-Graphen.

Regel 6

begin

- (1) wenn für zwei Knoten v_1 und v_2 eines IO-Graph IOG gilt:
- v_1 hat eine interne ausgehende Kante e ($e.IEL = \emptyset$ und $e.OEL = \emptyset$)
 - e ist eingehende Kante von v_2 und
 - für jede ausgehenden Kanten $e_{1-i} \neq e$ von v_1 gibt es eine ausgehende Kante e_{2-j} von v_2 mit:
- entweder 6.1
- v_2 ist nicht Startknoten von IOG ,
 - e ist die einzige eingehende Kante von v_2 und
 - $e_{1-i}.IEL = e_{2-j}.IEL$,
- oder 6.2
- $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$ und
 - e_{1-i} und e_{2-j} haben den gleichen Zielknoten,
- oder 6.3
- $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$ und
 - e_{1-i} und e_{2-j} sind Selfloop-Kanten,
- oder 6.4
- $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$,
 - v_2 ist Zielknoten von e_{1-i} und

- v_1 ist Zielknoten von e_{2-j} .
- dann:
 - jede eingehende Kante (v_x, v_1) von v_1 wird zu (v_x, v_2) ,
 - jede ausgehende Kante $(v_1, v_y) \neq e$ von v_1 wird zu (v_2, v_y) ,
 - wenn v_1 der Startknoten von IOG ist,
 - dann:
 - wird v_2 neuer Startknoten von IOG ,
 - entferne den Knoten v_1 aus IOG und
 - entferne die Kante e aus IOG .

end

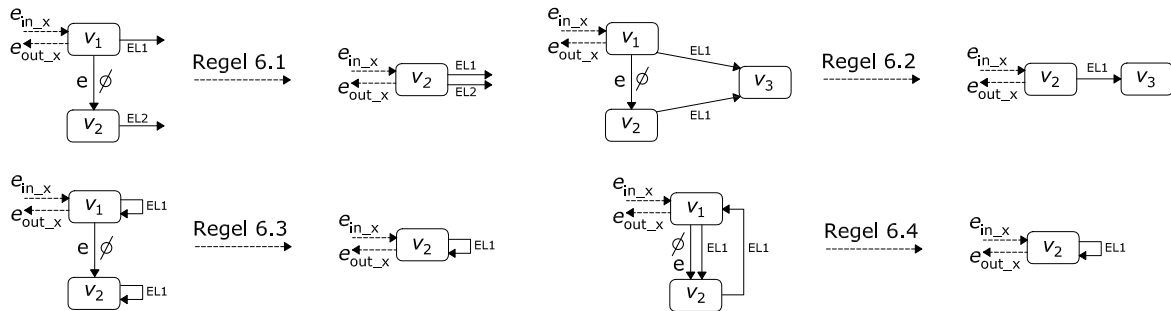


Abbildung 12: Regel 6 – Entfernen eines redundanten Knoten mit interner Kante

Auch durch Regel 6 entstehen parallele Kanten. Diese werden mit Regel 1 entfernt.

Durch die Regel 7 werden Pfade mit zwei Kanten überprüft und gegebenenfalls in einen Pfad mit einer Kante überführt. Dabei kann ein Knoten des IO-Graphen gelöscht werden. Dennoch entsteht ein IOT-Failure-äquivalenter IO-Graph.

Regel 7

begin

- (1) wenn für einen Knoten v eines IO-Graph IOG gilt:
 - v hat mindestens eine eingehende Kante,
 - v hat mindestens eine ausgehende Kante,
 - v hat keine Selfloop-Kanten und
 - für jede ausgehenden Kanten e_{out} von v gilt:
 - $e_{out} \cdot IEL = \emptyset$,
- dann:
 - für jede eingehenden Kanten $e_{in} = (v_{in}, v)$ von v :
 - für jede ausgehenden Kanten $e_{out} = (v, v_{out})$ von v :
 - erzeuge die Kante $e_{neu} = (v_{in}, v_{out})$,
 - setze $e_{neu} \cdot IEL = e_{in} \cdot IEL$,
 - setze $e_{neu} \cdot OEL = e_{in} \cdot OEL + e_{out} \cdot OEL$ und
 - füge die neue Kante e_{neu} in die Kantenmenge E von IOG ein,
 - wenn v Startknoten von IOG ist,
 - dann:
 - entferne alle eingehenden Kanten $e_{in} = (v_{in}, v)$ von v aus IOG .
 - sonst:
 - entferne den Knoten v aus IOG ,
 - entferne alle eingehenden Kanten $e_{in} = (v_{in}, v)$ von v aus IOG und
 - entferne alle ausgehenden Kanten $e_{out} = (v, v_{out})$ von v aus IOG .

end

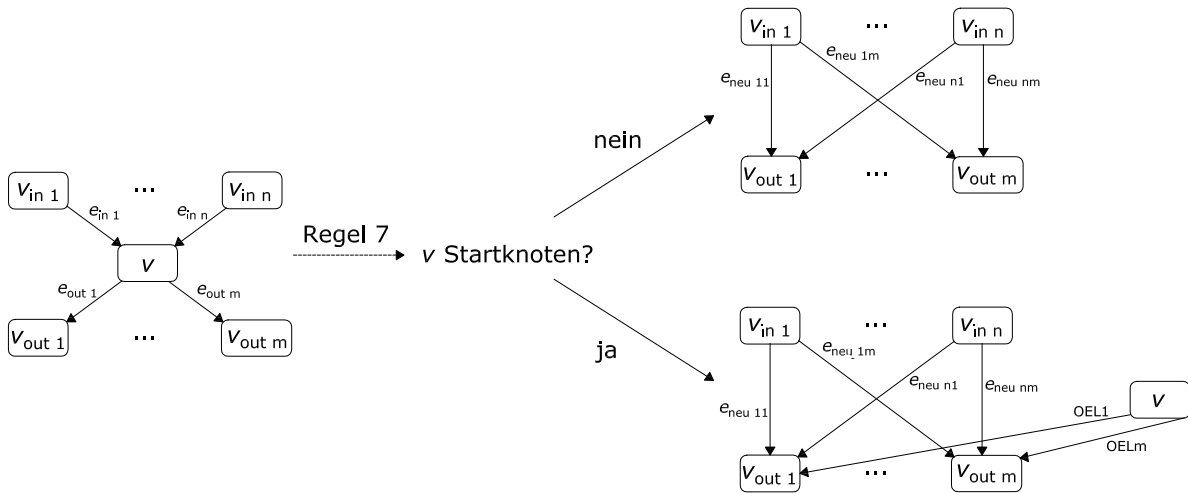


Abbildung 13: Regel 7 – Verschmelzen von Pfaden mit zwei Kanten zu Pfaden mit einer Kante

In Regel 8 gehen wir jetzt noch einen Schritt weiter als in Regel 1 und suchen zu einer Kante parallele Pfade. Bei erfolgreicher Suche kann die Kante entfernt werden. Da nur Kanten entfernt werden, gehen keine Systemzustände verloren.

Regel 8

begin

- (1) wenn für einen Pfad σ_{rm} mit einer einzigen Kante e eines IO-Graph IOG gilt:
 - es gibt einen Pfad σ im IO-Graph IOG mit:
 - σ ist parallel zu σ_{rm} ,
 - σ_{rm} und σ sind IO-äquivalent und
 - σ ist schaltabhängig von σ_{rm} ,
- dann:
 - entferne die Kante e aus IOG .

end



Abbildung 14: Regel 8 – Löschen einer redundanten Kante

Durch Regel 9 werden Selfloop-Kanten entfernt. Wieder bleiben alle Knoten erhalten und es gehen keine Systemzustände verloren.

Regel 9

begin

- (1) wenn für Kante e eines IO-Graph IOG gilt:
 - der Ausgangsknoten von e ist gleich dem Zielknoten von e ,
 - $(-e.IEL) = e.OEL$ und
 - es gibt eine weitere Kante $e_{out} \neq e$ im IO-Graph IOG mit:

- e und e_{out} haben den gleichen Ausgangsknoten und
 - $(-e_{out}.IEL) \leq (-e.IEL)$,
- dann:
- entferne die Kante e aus IOG .

end



Abbildung 15: Regel 9 – Löschen einer Selfloop-Kante

Alle bis hier gezeigten Kondensationsregeln erhalten entweder alle Markierungen eines Graphen oder nur die Deadlocks. Für die Menge der erreichbaren Markierungen bedeutet dies: Wir können zwar den IO-Graph reduzieren, müssen aber weiterhin sämtliche Markierungen des IO-Netzes speichern. Meist ist aber nur von Interesse, ob eine bestimmte Teilmarkierung und damit ein konkreter Systemzustand überhaupt erreicht werden kann. Daher schauen wir uns als nächstes eine Möglichkeit an, wie die Regeln 5, 6 und 7 modifiziert werden können, um den IO-Graph weiter zu reduzieren und dennoch den Erhalt bestimmter Teilmarkierungen zu garantieren.

Regel 10-5 (modifizierte Regel 5)

begin

- (1) wenn für den Startknoten v und der gesuchten Teilmarkierung m_{search} eines IO-Graph IOG gilt:
- v hat keine eingehenden Kanten,
 - v hat genau eine ausgehende Kante e ,
 - e ist interne Kante ($e.IEL = \emptyset$ und $e.OEL = \emptyset$),
 - e ist keine Selfloop-Kante,
 - und
 - entweder
 - m_{search} ist nicht Teilmarkierung einer in v enthaltenen Markierungen m ,
 oder
 - m_{search} ist Teilmarkierung einer im Zielknoten der Kante e enthaltenen Markierungen m ,
- dann:
- entferne den Knoten v aus IOG ,
 - entferne die Kante e aus IOG und
 - der Zielknoten von e wird neuer Startknoten von IOG .

end

Regel 10-6 (modifizierte Regel 6)

begin

- (1) wenn für zwei Knoten v_1, v_2 und der gesuchten Teilmarkierung m_{search} eines IO-Graph IOG gilt:
- v_1 hat eine interne ausgehende Kante e ($e.IEL = \emptyset$ und $e.OEL = \emptyset$)
 - e ist eingehende Kante von v_2 ,

- für jede ausgehenden Kanten $e_{1-i} \neq e$ von v_1 gibt es eine ausgehende Kante e_{2-j} von v_2 mit:
entweder 10-6.1
 - v_2 ist nicht Startknoten von IOG
 - e ist die einzige eingehende Kante von v_2 und
 - $e_{1-i}.IEL = e_{2-j}.IEL$,
 oder 10-6.2
 - $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$ und
 - e_{1-i} und e_{2-j} haben den gleichen Zielknoten,
 oder 10-6.3
 - $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$ und
 - e_{1-i} und e_{2-j} sind Selfloop-Kanten,
 oder 10-6.4
 - $e_{1-i}.IEL = e_{2-j}.IEL$,
 - $e_{1-i}.OEL = e_{2-j}.OEL$,
 - v_2 ist Zielknoten von e_{1-i} und
 - v_1 ist Zielknoten von e_{2-j} .
 - und
entweder
 - m_{search} ist nicht Teilmarkierung einer in v enthaltenen Markierungen m ,
 oder
 - m_{search} ist Teilmarkierung einer im Zielknoten der Kante e enthaltenen Markierungen m ,
- dann:
- jede eingehende Kante (v_x, v_1) von v_1 wird zu (v_x, v_2) ,
 - jede ausgehende Kante $(v_1, v_y) \neq e$ von v_1 wird zu (v_2, v_y) ,
 - wenn v_1 der Startknoten von IOG ist,
- dann:
 - wird v_2 neuer Startknoten von IOG ,
 - entferne den Knoten v_1 aus IOG und
 - entferne die Kante e aus IOG .

end

Regel 10-7 (modifizierte Regel 7)

begin

- (1) wenn für einen Knoten v und der gesuchten Teilmarkierung m_{search} eines IO-Graph IOG gilt:
- v hat mindestens eine eingehende Kante,
 - v hat mindestens eine ausgehende Kante,
 - v hat keine Selfloop-Kanten,
 - für jede ausgehenden Kanten e_{out} von v gilt:
 - $e_{\text{out}}.IEL = \phi$,
 - und
entweder
 - m_{search} ist nicht Teilmarkierung einer in v enthaltenen Markierungen m ,
 oder
 - es gibt eine ausgehenden Kanten e_{out} von v mit:
 1. die Output-Edge-Labels von e_{out} ($e_{\text{out}}.OEL$) produzieren keine Marken auf IO-Plätzen, die Teil der gesuchten Markierung m_{search} sind,

2. m_{search} ist Teilmarkierung einer im Zielknoten der Kante e_{out} enthaltenen Markierungen m ,

- dann:

- für jede eingehenden Kanten $e_{\text{in}} = (v_{\text{in}}, v)$ von v :
 - für jede ausgehenden Kanten $e_{\text{out}} = (v, v_{\text{out}})$ von v :
 - erzeuge die Kante $e_{\text{neu}} = (v_{\text{in}}, v_{\text{out}})$,
 - setze $e_{\text{neu}}.IEL = e_{\text{in}}.IEL$,
 - setze $e_{\text{neu}}.OEL = e_{\text{in}}.OEL + e_{\text{out}}.OEL$ und
 - füge die neue Kante e_{neu} in die Kantenmenge E von IOG ein,
- wenn v Startknoten von IOG ist,
 - dann:
 - entferne alle eingehenden Kanten $e_{\text{in}} = (v_{\text{in}}, v)$ von v aus IOG .
 - sonst:
 - entferne den Knoten v aus IOG ,
 - entferne alle eingehenden Kanten $e_{\text{in}} = (v_{\text{in}}, v)$ von v aus IOG und
 - entferne alle ausgehenden Kanten $e_{\text{out}} = (v, v_{\text{out}})$ von v aus IOG .

end

4 Werkzeug

Das Programm *ioKoCheck* ist ein Modelchecker, der IO-Netze auf konkrete erreichbare Zustände, insbesondere auf Deadlocks, überprüfen kann. Das Modelleingabeformat ist die IO-Netznotation *ioLoLA*. Diese ist eine Erweiterung des Eingabeformats des Modelcheckers LoLA [Sch00]. Das ioLoLA-Format erweitert das LoLA-Eingabeformat um Schnittstellenplätze zwischen Netzkomponenten. (Eine Grammatik für das Eingabeformat ioLoLA finden wir im Anhang A). ioKoCheck implementiert die im Kapitel 3 beschriebenen Methoden zur Graphenreduktion.

Das Werkzeug ist kommandozeilenbasiert. Dem Programmaufruf kann eine ioLoLA-Datei als Parameter übergeben werden. Nach dem Programmstart wird der Benutzer aufgefordert eine ioLoLA-Datei aus dem aktuellen Programmverzeichnis zu benennen, falls diese nicht schon durch den Programmaufruf übergeben wurde. Im Hauptmenü hat der Benutzer die Optionen *Modell auf Erreichbarkeit eines bestimmten Zustands prüfen*, *Modell auf Deadlocks prüfen*, *neues Netz laden* und *Programm beenden*. Will der Benutzer die Erreichbarkeit eines Systemzustandes prüfen, wird er aufgefordert, den zu überprüfenden Systemzustand in Form einer Petrinetz-Teilmarkierung anzugeben. Es müssen jeweils ein Platzname und anschließend die Markenanzahl eingegeben werden. Danach startet die Analyse und es werden Programminformationen zu den aktuell durchgeführten Methoden ausgegeben. Falls die gesuchte Markierung erreichbar ist, wird am Ende der Analyse ein konkret erreichbarer Modellzustand ausgegeben, welcher die gesuchte Teilmarkierung enthält. Das Prüfen auf Deadlocks startet sofort und gibt ebenfalls Statusinformationen aus. Als Ergebnis werden alle gefundenen Deadlocks ausgegeben.

Das Programm wurde in C++ [Sch06] programmiert und mit dem Compiler gcc 4.0.1 kompiliert. Außerdem wurde der Lexer Flex in der Version 2.5.33 und der Parser Bison in der Version 2.3 verwendet. Für die Datenstruktur wurde die Petrinetz API des GNU BPEL2oWFN Projektes [LGZ07] verwendet, die Klassen zur Speicherung von Petrinetzen bereitstellt.

Das Hauptproblem bei der Implementierung des Programms war die Optimierung der Datenstruktur für die Graphen und IO-Graphen. Da vorher wenig Erfahrungen im Bereich der Entwicklung von Algorithmen für Graphen vorhanden war, war der erste Datenstrukturentwurf daran ausgerichtet, keine Redundanzen zu haben und wenig Speicherplatz zu belegen. Es sollte eine definitionsnahe und dadurch leicht verständliche Graphenstruktur sein. Für das eingesetzte Testmodell, die speisenden Philosophen, erfüllte dieser Entwurf auch alle Anforderungen. Die Re-

duktionsregeln arbeiteten schnell und das Programm lieferte das richtige Ergebnis. Leider wurde außer Acht gelassen, dass das Philosophenmodell den Idealfall eines komponentenbasierten Systems darstellt, denn es besteht aus identischen und sehr kleinen Komponenten. Der erste Test mit dem im nächsten Kapitel vorgestellten Fluglinienmodell offenbarte ein Geschwindigkeitsproblem bei der parallelen Komposition und bei den Reduktionsregeln. Da wir den IO-Graphen jeder neu hinzukommenden Komponente erst aufbauen und mit dem bereits reduzierten IO-Graphen komponieren müssen, erhalten wir schon bei kleinen Komponenten sehr große IO-Graphen. Auf den Graphen des Fluglinienmodells benötigte die Suche in den Listen des ersten Datenstrukturentwurfs sehr viel Zeit. Deshalb werden die Markierungen und Kanten nun in Maps gespeichert. Als Schlüssel für die Markierungsmap wird eine für jede mögliche Markierung eindeutige Zeichenfolge verwendet. Diese Zeichenfolge ermöglicht einen schnellen Vergleich zweier Markierungen und damit auch eine schnellere parallele Komposition. Eine Geschwindigkeitsoptimierung für die Reduktionsregeln wird durch das Speichern aller eingehenden und ausgehenden Kanten einer Markierung in der Markierungsklasse erreicht. Das ist vor allem in der Regel 8 von Vorteil, da hier eine Tiefensuche durch den IO-Graph nötig ist. Durch diese Optimierungen läuft ioKoCheck erheblich schneller.

Dennoch muss auch dieses Programm mit beschränkten Speicherressourcen arbeiten. Die Größe der erzeugbaren IO-Graphen hängt direkt von der Größe des zur Verfügung stehenden Arbeitsspeichers ab. In der folgenden Fallstudie sehen wir die Effektivität der Reduktionsregeln zur Abschwächung des Problems der Zustandsraumexplosion und bekommen eine Vorstellung von der Größe der durch ioKoCheck erzeugbaren IO-Graphen.

5 Fallstudie

Um einen ersten Eindruck des Reduktionspotentials eines IO-Graphen zu bekommen, werden wir uns das Beispiel der *speisenden Philosophen* anschauen.

Die Ausgangssituation ist:

- eine beliebige Anzahl Philosophen sitzt an einem runden Tisch,
- zwischen ihnen liegt jeweils eine Gabel und
- vor ihnen steht ein Teller mit Nudeln.

Jeder Philosoph hat folgende Handlungsmöglichkeiten:

- hat er keine Gabel in der Hand, kann er mit der linken Hand die Gabel zu seiner linken Seite aufnehmen, falls sich dort eine befindet,
- hat er nur in der linken Hand eine Gabel, kann er mit der rechten Hand die Gabel zu seiner rechten Seite aufnehmen, falls sich dort eine befindet, oder er wartet bis sie auf den Platz zurückgelegt wurde und
- hat er in beiden Händen eine Gabel, kann er essen. Danach legt er die Gabel in der linken Hand auf seiner linken Seite und die Gabel in der rechten Hand auf seiner rechten Seite ab.

Die Philosophen sitzen am Tisch und diskutieren, ab und zu bekommt einer Hunger und führt die Handlungen zum Essen aus. Wollen nun alle Philosophen auf einmal essen, kann das System zum Stillstand kommen. Wenn alle eine Gabel in der linken Hand halten, befindet sich bei keinem eine Gabel auf der rechten Seite. Das System verklemmt und die Philosophen verhungern.

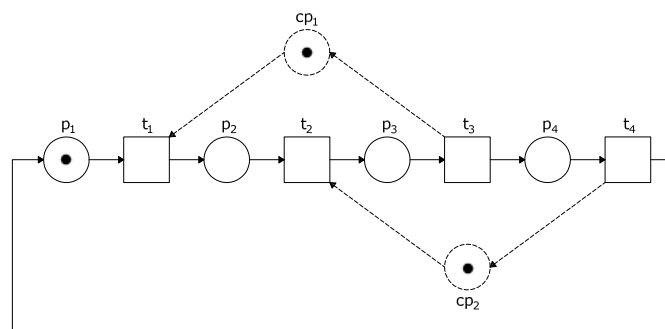


Abbildung 16: IO-Netz eines speisenden Philosophen

In Abbildung 16 sehen wir das IO-Netzmodell für einen einzelnen Philosophen. Das Modell für die speisenden Philosophen mit x Philosophen wird durch das Komponieren von x einzelnen Philosophennetzen generiert, wobei x eine natürliche Zahl größer 1 ist.

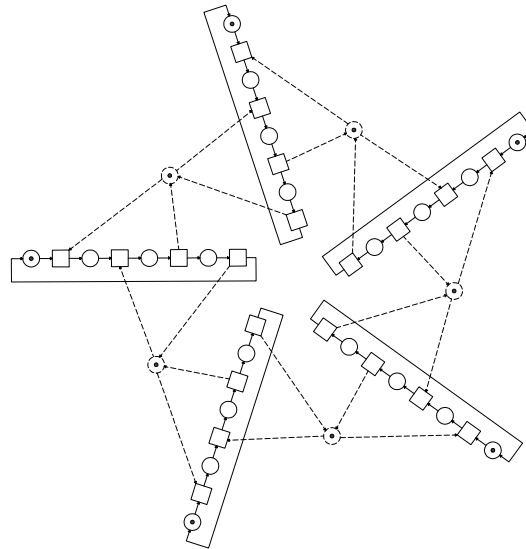


Abbildung 17: IO-Netz für fünf speisende Philosophen

Abbildung 17 zeigt das IO-Netzmodell für fünf Philosophen. Die Tabelle in Abbildung 18 zeigt die Größen der unreduzierten IO-Graphen des Philosophenmodells in Abhängigkeit von der Philosophenanzahl. Außerdem sehen wir in der Tabelle Werte der reduzierten IO-Graphen, die mittels unserer kompositionalen Verifikation deadlock-bewahrend reduziert wurden. Wir erkennen: Die Größen der finalen, reduzierten IO-Graphen scheint unabhängig von der Anzahl der Philosophen zu sein.

Anzahl Philosophen	2	3	4	5	8	10	50	100
Plätze	10	15	20	25	40	50	250	500
Transitionen	8	12	16	20	32	40	200	400
unreduzierter IO-Graph								
Zustände	8	26	80	242	6560	59048	-	-
Kanten	10	51	212	805	34984	393650	-	-
reduzierter IO-Graph								
Zustände	1	1	1	1	1	1	1	1
Kanten	0	0	0	0	0	0	0	0
maximaler Zwischengraph								
Zustände	4	4	4	4	4	4	4	4
Kanten	4	4	4	4	4	4	4	4

Abbildung 18: Tabelle mit IO-Graphengrößen für das Philosophenmodell

Kommen wir zu unserem zweiten Beispiel: das *Fluglinienmodell* [LKLR07]. Ein Reisender sucht einen Flug und beauftragt einen Agenten, bei allen zu ihm verbundenen Fluglinien nach der Verfügbarkeit des gesuchten Fluges anzufragen. Haben alle Fluglinien geantwortet, wird das Ergebnis dem Reisenden mitgeteilt. Das Modell besteht aus den Komponenten:

- ein Reisender,
- ein Agent und
- eine beliebige Anzahl Fluglinien.

Im Unterschied zu dem Philosophenmodell enthält das Fluglinienmodell neben einer beliebigen Anzahl identischer Komponenten, den Fluglinien, zusätzlich zwei Steuerungskomponenten, den Reisenden und den Agenten. Die Fluggesellschaften interagieren nur mit dem Agenten und nicht untereinander.

Anzahl Fluglinien	2	3	4	5	6	7	8
Plätze	52	69	86	103	120	137	154
Transitionen	39	53	67	81	95	109	123
unreduzierter IO-Graph							
Zustände	441	2992	23787	213778	-	-	-
Kanten	1154	10822	112906	1263610	-	-	-
reduzierter IO-Graph							
Zustände	3	3	3	3	3	3	-
Kanten	2	2	2	2	2	2	-
Maximaler Zwischengraph							
Zustände	19	38	98	332	1103	4780	-
Kanten	32	85	362	1967	7915	77841	-

Abbildung 19: Tabelle mit IO-Graphengrößen für das Fluglinienmodell

Die ioLoLA-Dateien für das Fluglinienmodell wurden mit GNU BPEL2oWFN [LGZ07] erstellt.

Die Tabelle in Abbildung 19 zeigt die Größen der unreduzierten IO-Graphen und die Größen der bzgl. der Bewahrung aller Deadlocks reduzierten IO-Graphen, soweit diese mit ioKoCheck berechenbar sind. Auch bei diesem Modell scheinen die Größen der finalen, reduzierten IO-Graphen unabhängig von der Anzahl der Komponenten zu sein. Dennoch zeigt die Tabelle, dass der reduzierte IO-Graph schon bei acht Fluglinien nicht mehr berechenbar ist. Bei genauerer Betrachtung der Komponenten zeigt sich, dass der Agent mit der Anzahl der Fluggesellschaften größer wird. Pro Gesellschaft kommt ein zusätzlicher Verwaltungsaufwand im Agenten hinzu. Dieser wird zwar bei der Komposition reduziert, muss aber erstmal im IO-Graphen des Agenten berücksichtigt werden. Bei unserer kompositionalen Verifikation ist demnach nicht nur der finale Erreichbarkeitsgraph von Interesse, sondern auch die Größe der IO-Graphen der einzelnen Komponenten. Diese werden vor der Reduktion erstellt, sodass keine Abschwächung der Zustandsraumexplosion stattfindet. In den Tabellen sehen wir deshalb auch die Werte der größten Zwischenergebnis-Graphen für beide Beispiele. Diese Zahlen zeigen für das Fluglinienmodell ein ganz anderes Ergebnis und erklären, warum wir das Modell nur für sieben Fluggesellschaften

betrachten können. Und obwohl eine Graphengröße von 4780 Zuständen und 77841 Kanten für den maximalen Zwischen-IO-Graphen bei sieben Fluglinien nicht sehr groß erscheint, genügen die zwei Gigabyte Arbeitsspeicher des Testsystems nicht für die Berechnung der Zwischen-IO-Graphen bei acht Fluglinien.

Wenn wir die reduzierten Graphengrößen des Modells der speisenden Philosophen mit den Resultaten aus dem Papier von Juan et al. [JTM98, Fig. 29. S. 956] vergleichen, stellen wir fest: Mit 2 Zuständen und 3 Kanten weisen die finalen reduzierten IO-Graphen andere Werte als die in unserer Tabelle auf. Die unterschiedlichen Ergebnisse kommen durch die unterschiedlichen Eingabenetze zustande. In dem Papier wird ein geöffnetes Philosophennetz benutzt, d.h. die Philosophen sitzen nicht an einem runden Tisch, sondern an einer langen Tafel und mit einer zusätzlichen Gabel. In der Abbildung 20 sehen wir das offene Philosophennetz für fünf Philosophen. Benutzen wir ein solches IO-Netz als Eingabe für ioKoCheck, erhalten wir die Ergebnisse aus dem Papier.

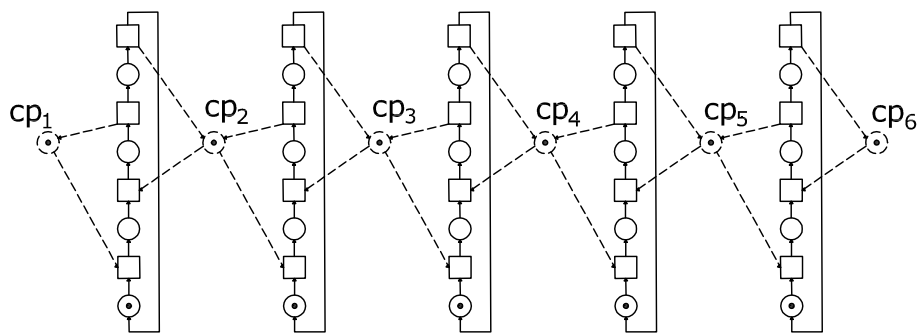


Abbildung 20: IO-Netz eines speisenden Philosophen

6 Fazit

Ziel dieser Arbeit war es, das von Juan et al. beschriebene Werkzeug IOTA neu zu implementieren, die Ergebnisse des Papiers zu überprüfen und die Stärken bzw. Schwächen dieser Verifikationsmethode zu erarbeiten. Mit dem Modelchecker ioKoCheck steht jetzt ein Werkzeug zur Verfügung, das mit den Reduktionsmethoden aus dem Papier von Juan et al. arbeitet. Mit Hilfe dieses Werkzeuges konnten wir die Ergebnisse des Papiers für das Beispiel der speisenden Philosophen überprüfen und bestätigen. Damit wurden die ersten beiden Ziele erreicht. Mit Hilfe des Fluglinienmodells haben wir die Grenzen der kompositionalen Verifikation nach Juan et al. gesehen. Nun wollen wir zusammenfassend die Stärken und Schwächen der Verifikationsmethode aufzeigen.

Die Ergebnisse der Fallstudie zeigen, dass es Modelle gibt, die durch unsere kompositionale Verifikation besser analysierbar sind, als dies mit anderen Verifikationstechniken möglich ist. Vor allem solche Modelle, die aus vielen kleinen und möglichst identischen Komponenten bestehen, wie das Philosophenmodell, bieten großes Reduktionspotential. Dennoch ergeben sich aus den Ergebnissen auch ein paar entscheidende Nachteile. Das Einsatzfeld unserer kompositionalen Verifikation beschränkt sich auf Modelle, bei denen eine Komponentenstruktur bekannt ist, da wir ein IO-Netz als Eingabe benötigen. Die Hauptschwachstelle des kompositionalen Verifikationsansatzes von Juan et al., die Notwendigkeit einer Komponentenstruktur aus kleinen Komponenten, bietet zugleich großes Potential zur Verbesserung. In dem gezeigten Fluglinienmodell ist scheinbar eine weitere, feinere Komponentenstruktur in der Agentenkomponente vorhanden, da die finalen, reduzierten Systeme scheinbar eine konstante Größe haben. Hätten wir die Möglichkeit diese automatisch zu erkennen, würden wir entschieden weiter kommen. Daher sollte das Ziel einer Weiterentwicklung sein, eine automatische Erkennung von Komponenten in die Verifikationstechnik zu integrieren.

Abschließend ziehen wir ein Fazit zur Implementierung des Modelcheckers ioKoCheck. Die Umsetzung der leicht zu verstehenden Algorithmen im Papier erforderte Kenntnisse über die Programmierung von Graphenstrukturen und Graphenalgorithmien. Viele dieser Kenntnisse wurden erst im Laufe der Implementierung gewonnen. Vor allem die mehrmalige Änderung der Datenstruktur für die Erreichbarkeitsgraphen verursachte einen erheblichen Aufwand, da zum Teil ganze Regeln neu programmiert werden mussten. In seiner nun vorliegenden Version

arbeitet das Programm fehlerfrei und berechnet ein Ergebnis, wenn die Computerressourcen ausreichen.


```
transitionlist ::=      /* empty */ |
                       transitionlist transition
transition ::=         TRANSITION NAME CONSUME arclist_opt ;
                       PRODUCE arclist_opt ;
arclist_opt ::=        /* empty */ |
                       arclist
arclist ::=            NAME : NUMBER |
                       arclist , NAME : NUMBER
}
```

8 Literatur

- [CGP99] Clarke, Edmund M.; Grumberg, Orna; Peled, Doron A.: *Model Checking*. MIT Press, 1999. - ISBN 0-262-03270-8
- [JTM98] Juan, Eric Y.T.; Tsai, Jeffrey J.P.; Murata, Tadao: *Compositional Verification of Concurrent Systems using Petri-Net-Based Condensation Rules*. University of Illinois at Chicago. ACM Transactions on Programming Languages and Systems, Vol. 20, No. 5, September 1998.
- [LGZ07] Lohmann, Niels; Gierds, Christian; Znamirovski, Martin: *Translating BPEL Process to Open Workflow Nets*, GNU BPEL2oWFN Version 2.0.3, 29 June 2007; <http://www.gnu.org/software/bpel2owfn/>
- [LKLR07] Lohmann, Niels; Kopp, Oliver; Leymann, Frank; Reisig, Wolfgang: Analyzing BPEL4Chor: Verification and Participant Synthesis. In: Dumas, Marlon (Hrsg.); Heckel, Reiko (Hrsg.): *Web Services and Formal Methods, Proceedings of 4th International Workshop (WS-FM 2007)*, Springer-Verlag, 2008, (LNCS 4937), S. 46-60
- [Sch00] Schmidt, Karsten: LoLA: A Low Level Analyser. In: Nielsen, Mogens (Hrsg.); Simpson, Dan (Hrsg.): *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, Springer-Verlag, Juni 2000 (LNCS 1825), S. 465–474
- [Sch06] Schneeweiß, Ralf: *Moderne C++ Programmierung*. Springer-Verlag, 2006. - ISBN 3-540-22281-2
- [Sta90] Starke, Peter H.: *Analyse von Petri-Netz-Modellen*. B. G. Teubner Stuttgart, 1990. - ISBN 3-519-02244-3