Fachbereich 4: Informatik

Workflows und Logik

Untersuchung der Beschreibungsmöglichkeiten Petri-Netz basierter Workflow-Netze durch klassische und nicht-klassische Logiken

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von
Christian Diefenthal

Betreuer: Prof. Dr. Kurt Lautenbach, Institut für Softwaretechnik, Fachbereich 4

Erstgutachter: Prof. Dr. Kurt Lautenbach, Institut für Softwaretechnik, Fachbereich 4
Zweitgutachter: Dr. Stephan Philippi, Institut für Softwaretechnik, Fachbereich 4

Koblenz, im März 2009

Erklärung des Verfassers

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Ort, Datum	Unterschrift des Verfassers

Inhaltsverzeichnis

1 Einführung

In vielen Bereichen des heutigen Lebens gibt es immer kompliziertere Anforderungen, wie eine bestimmte Aufgaben zu erledigen sind oder wie bestimmte Situationen zu behandeln sind. Vor allem in großen Unternehmen und anderen großen Organisationen werden Mittel und Wege benötigt, solche Arbeitsabläufe und Geschäftsprozesse zu organisieren. Auch Softwareprogramme, die auch nur eine besondere Form von Arbeitsabläufen sind, werden immer komplexer. Mit Hilfe sogenannter Workflows können diese Abläufe beschrieben werden. Dazu stehen verschiedene Möglichkeiten wie zum Beispiel die in [Petr62] eingeführten Petri-Netze, die in [AaHo05] vorgestellten und auf Petri-Netzen basierende Sprache YAWL oder die auch in der UML verwendeten Kontrollflussgraphen zur Verfügung. Petri-Netze eignen sich darüber hinaus sehr gut, Workflows auf ihre Korrektheit hin zu analysieren.

In Abb. 1.1 ist ein aus [Aals98, S. 24] entnommenes Bild eines Workflows gegeben. Dabei sind Vorgänge wie *register* als Quadrate, Zustände bzw. Ergebnisse wie i als Kreise (im Folgenden Stellen genannt) dargestellt. Die Pfeile zeigen den Übergang von einem Zustand in einen anderen an. Solche Übergänge werden durch Ausführen einzelner Vorgänge ausgelöst. Wann ein Vorgang ausgelöst wird, ist durch die zusätzlich an den Quadraten stehenden Symbole festgelegt. Dabei ist der Pfeil ein benutzergesteuerter, die Uhr ein zeitgesteuerter und der Brief ein ereignisgesteuerter Auslöser. Steht an einem Vorgang kein Auslöser, wird der Vorgang automatisch ausgeführt. Dies alles natürlich nur, wenn die Vorbedingungen des Vorgangs erfüllt sind. Der Workflow startet mit einer sogenannten Marke auf der Stelle i und ist beendet, wenn eine Marke auf die Stelle o gelegt wurde.

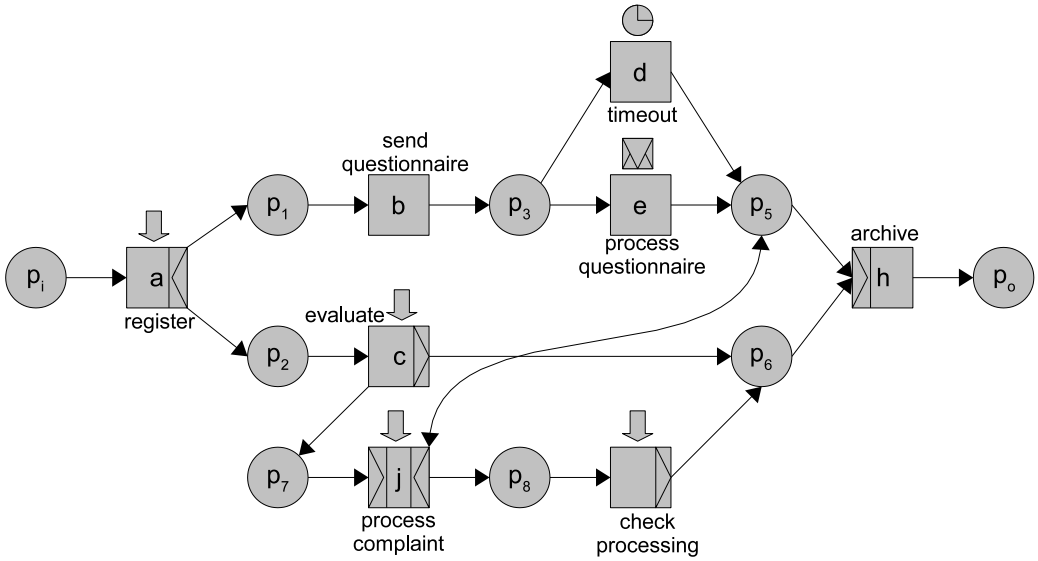


Abb. 1.1: Workflow eines Reklamationsvorgangs

Ziel dieser Arbeit ist es, als Petri-Netze dargestellte Workflows in die Aussagenlogik, Prädikatenlogik, Transaktionslogik und Aktionslogik zu transformieren, sodass es möglich ist, Workflows auch mit Hilfe der verschiedenen Inferenz-Techniken der einzelnen Logiken

analysieren zu können. Die ebenfalls durchgeführte umgekehrte Transformation soll es ermöglichen, Workflows als logische Ausdrücke zu erstellen und diese dann in Petri-Netze zu übertragen.

In Kapitel 2 werden zunächst alle für diese Arbeit wichtigen Begriffe der Petri-Netz-Theorie eingeführt, bevor in Kapitel 3 die in [RuHoAaMu] und [AaHoKiBa03] präsentierten Workflow-Pattern als Petri-Netze dargestellt werden. Im anschließenden Kapitel 4 werden die Transformationen zwischen Petri-Netzen und den einzelnen Logiken auch unter Zuhilfenahme der Workflow-Pattern beschrieben. Das abschließende Kapitel 5 fasst die Ergebnisse zusammen und bewertet diese. Zusätzlich wird ein Ausblick auf offene Fragen und Probleme bei den hier vorgenommenen Transformationen gegeben.

2 Grundlagen der Netztheorie

In diesem Kapitel werden zunächst einige Grundlagen der Netztheorie vorgestellt und anschließend eine kurze formale Einführung von Workflows gegeben. Die folgenden Definitionen für p/t-Netze und pr/t-Netze als eine ihrer Erweiterungen orientieren sich hierbei an [Laut02-1] und [Laut06].

2.1 p/t-Netze

Definition 1 Ein *Stellen/Transitions-Netz (p/t-Netz)* ist ein Quadrupel $\mathcal{N} = (P, T, F, W)$ mit

- $P = \{p_1, p_2, \ldots, p_m\}$, einer endlichen Menge von Stellen,
- $T = \{t_1, t_2, \ldots, t_n\}$, einer endlichen Menge von Transitionen,
- $P \cap T = \emptyset$,
- $F \subset (P \times T) \cup (T \times P)$, einer Menge gerichteter Kanten und
- $W : F \to \mathbb{N}\backslash\{0\}$, einer Abbildung, die jeder Kante ein Gewicht zuordnet.

[Laut02-1]

In der graphischen Darstellung werden Stellen durch Kreise und Transitionen durch Rechtecke dargestellt. Eine Kante $(x_1, x_2) \in F$ wird durch einen Pfeil von x_1 nach x_2 dargestellt. Die Kantengewichte werden durch an der zugehörigen Kante stehende Zahlen repräsentiert.
Im Fall $W : F \to 1$ schreibt man auch $\mathcal{N} = (P, T, F)$.

Beispiel 1 Für das folgende Beispiel-Netz $\mathcal{N} = (P, T, F, W)$ in Abb. 2.1

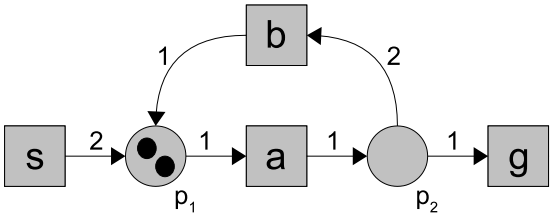


Abb. 2.1: Ein p/t-Netz-Beispiel

ist

- $P = \{p_1, p_2\}$,
- $T = \{s, a, b, g\}$,

- $F = \{(s, p_1), (p_1, a), (a, p_2), (p_2, g), (p_2, b), (b, p_1)\}$ und
- $W(f) = \begin{cases} 2 & \text{wenn } f \in \{(s, p_1), (p_2, b)\} \\ 1 & \text{sonst.} \end{cases}$

□

Definition 2 Sei $\mathcal{N} = (P, T, F, W)$ ein p/t-Netz. Der *Vorbereich* (*Nachbereich*) eines Knotens $x \in P \cup T$ ist definiert als

$$^\bullet x = \{y \in P \cup T | (y, x) \in F\}$$

$$(x^\bullet = \{y \in P \cup T | (x, y) \in F\}).$$

[Laut02-1]

So ist im Netz in Abb. 2.1 z. B. $^\bullet p_1 = \{s, b\}$ und $p_1^\bullet = \{a\}$

Definition 3 Ein p/t-Netz $\mathcal{N} = (P, T, F)$ heißt *stark zusammenhängend* gdw. der gerichtete Graph $(P \cup T, F)$ stark zusammenhängend ist. d. h. für jedes Paar (x, y) mit $x, y \in P \cup T$ gibt es einen Pfad von x nach y.

□

Definition 4 Ein p/t-Netz $\mathcal{N} = (P, T, F)$ heißt *zykelfrei*, wenn es im gerichteten Graph $(P \cup T, F)$ für kein Paar (x, y) mit $x, y \in P \cup T$ einen Pfad von x nach y gibt. Ansonsten heißt $\mathcal{N}$ *zyklisch*.

□

Definition 5 Sei $\mathcal{N} = (P, T, F, W)$ ein p/t-Netz.

- Eine *Markierung* von $\mathcal{N}$ ist eine Abbildung $M : P \to \mathbb{N}$. $M(p)$ gibt die Anzahl der Marken auf p an.
- $p \in P$ heißt *markiert* unter M gdw. $M(p) \geq 1$. Ansonsten heißt p *unmarkiert*.
- Eine Transition $t \in T$ heißt *aktiviert* oder *feuerbar* unter M, in Zeichen $M[t\rangle$, gdw.

$$\forall p \in {}^\bullet t : M(p) \geq W(p, t).$$

[Laut02-1]

Marken werden in der graphischen Darstellung durch Punkte dargestellt. So ist im Netz die Markierung $M_0(p_1) = 2$ und $M_0(p_2) = 0$. Die Stelle p_1 ist also markiert, die Stelle p_2 ist unmarkiert. Feuerbare Transitionen unter M_0 sind a und s.[1] Alle anderen Transitionen sind unter dieser Markierung nicht feuerbar.

[1] Da s keine Eingangskante besitzt, existieren keine Feuerbedingungen in Form von Stellen im Vorbereich von s und s ist immer aktiviert.

Definition 6 Die Menge der von einer Markierung M_0 aus erreichbaren Markierungen, in Zeichen $[M_0\rangle$, ist induktiv definiert als

$$M_0 \in [M_0\rangle,$$
$$M \in [M_0\rangle \wedge M[t\rangle M' \Rightarrow M' \in [M_0\rangle.$$

[Laut02-1]

Definition 7 Wenn $M[t\rangle$, kann die Transition t feuern. So entsteht eine neue Markierung M', in Zeichen $M[t\rangle M'$ mit

$$M' = \begin{cases} M(p) - W(p,t) & \text{wenn } p \in {}^\bullet t \backslash t^\bullet, \\ M(p) + W(t,p) & \text{wenn } p \in t^\bullet \backslash {}^\bullet t, \\ M(p) - W(p,t) + W(t,p) & \text{wenn } p \in {}^\bullet t \cap t^\bullet, \\ M(p) & \text{sonst.} \end{cases}$$

[Laut02-1]

Feuert im Netz aus Abb. 2.1 die Transition a, ergibt sich eine neue Markierung M':

$$M'(p_1) = M_0(p_1) - W(p_1, a) = 2 - 1 = 1$$

und

$$M'(p_2) = M_0(p_2) + W(a, p_2) = 0 + 1 = 1.$$

Diese neue Markierung ist auch in Abb. 2.2 zu sehen. Transition a ist weiß markiert, um zu zeigen, dass diese gerade gefeuert hat.

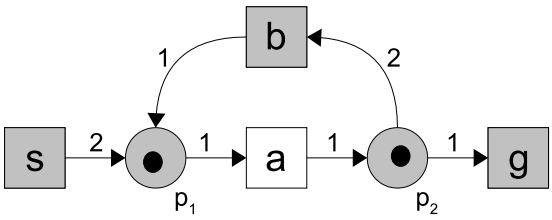


Abb. 2.2: Das p/t-Netz-Beispiel nach Feuern von a

2.2 pr/t-Netze

Vor der Definition von pr/t-Netzen müssen noch Multimengen und n-Tupel definiert werden. Diese beiden Begriffe werden bei pr/t-Netzen benötigt, um Markierungen und Kantenlabels zu beschreiben.

Definition 8 Sei A eine Menge. Eine Abbildung $m : A \to \mathbb{Z}$ ist eine *Multimenge* über A. $MS(A)$ bezeichnet die Menge der Multimengen über A.
Die Abbildung $m_+ : A \to \mathbb{N}$ ist eine nicht-negative Multimenge über A. $MS_+(A)$ bezeichnet die Menge der nicht-negativen Multimengen über A.

[Laut02-2]

Definition 9 Sei U eine endliche Menge von Konstanten und X eine endliche Menge von Variablen. Für $n \in \mathbb{N}$ ist

- $Y_0 = U_0 := <>$ das *0-Tupel* (Das 0-Tupel $<>$ ist die Marke von low-level-Netzen (wie z. B. p/t-Netze)),
- $Y_n := \{< e_1, \ldots, e_n > | e_i \in U \cup X, 1 \leq i \leq n\}$ die *Menge der n-Tupel* und
- $U_n := \{< e_1, \ldots, e_n > | e_i \in U, 1 \leq i \leq n\}$ die *Menge der konstanten n-Tupel.*

[Laut02-2]

Definition 10 Die *Arität* oder *Länge eines Tupels* ist gegeben durch

$$ar(t) = n, \text{für } t \in U_n \cup Y_n.$$

[Laut02-2]

Mit Hilfe dieser Begriffe können nun pr/t-Netze und weitere, zu pr/t-Netzen gehörige Begriffe definiert werden.

Definition 11 Sei Y eine endliche Menge von n-Tupeln.
Ein Netz $\mathcal{N} = (P, T, F, L)$ ist ein *Prädikat/Transitions-Netz (pr/t-Netz)*, gdw.

- (P, T, F) ein p/t-Netz ist,
- $L : F \to \{G | G : MS_+ \to MS_+\}$, wobei $MS_+ = \bigcup_{n \geq 0} Y_n$ und
- $\forall p \in P, \forall l_1, l_2 \in \bigcup_{(x=p) \vee (y=p)} L(x, y) : ar(l_1) = ar(l_2)$.

L ist hierbei die Menge der Kantenlabels.

[Laut02-2]

Beispiel 2 In Abb. 2.3 ist ein Beispiel eines pr/t-Netzes $\mathcal{N}$, dessen Stellen, Transitionen und Kanten gleich dem Netz in Abb. 2.1 sind.

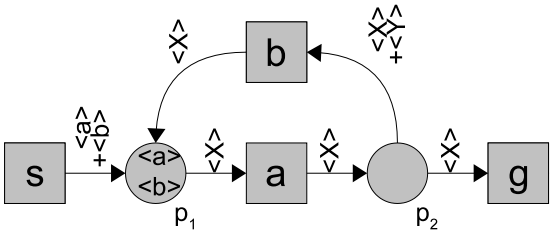


Abb. 2.3: Ein pr/t-Netz-Beispiel

Mit $\mathcal{N} = (P, T, F, L)$ ist also

- $P = \{p_1, p_2\}$,
- $T = \{s, a, b, g\}$,

- $F = \{(s, p_1), (p_1, a), (a, p_2), (p_2, g), (p_2, b), (b, p_1)\}$ und
- $L(f) = \begin{cases} \{< a >, < b >\} & \text{wenn } f = (s, p_1) \\ \{< X >, < Y >\} & \text{wenn } f = (p_2, b) \\ \{< X >\} & \text{sonst.} \end{cases}$

□

Definition 12 Sei $\mathcal{N} = (P, T, F, L)$ ein pr/t-Netz.

- Eine *Markierung* M von N ist eine Abbildung $M : P \to \bigcup_{n \geq 0} MS_+(U_n)$. $M(p)$ gibt die Tupel auf p an.
- $p \in P$ heißt *markiert* unter M gdw. $M(p) \neq \emptyset$. Ansonsten heißt p unmarkiert.

[Laut02-2]

Im pr/t-Netz in Abb. 2.3 ist also p_1 markiert mit $M(p_1) = \{< a >, < b >\}$, p_2 hingegen ist unmarkiert, d. h. $M(p_2) = \emptyset$.

Definition 13 Sei X eine endliche Menge von Variablen. In einem pr/t-Netz $\mathcal{N} = (P, T, F, L)$ ist die *Arität einer Stelle* $p \in P$ definiert als die Länge der Kantenlabels „um“ p:

$$ar(p) := ar(q) \text{ für } q \in \bigcup_{(x=p) \vee (y=p)} L(x, y).$$

Die *Arität einer Transition* $t \in T$ ist definiert als die Anzahl der freien Variablen „um“ t:

$$ar(t) := |\{x \in X \wedge \exists lb \in \bigcup_{(y=t) \vee (z=t)} L(y, z) : x \in lb\}|.$$

[Laut02-2]

Für das Netz in Abb. 2.3 heißt dies für die Arität der Stellen, dass $ar(p_1) = ar(p_2) = 1$. Da an den Kanten um Transition s keine freien Variablen vorhanden sind, ist $ar(s) = 0$. An den Kanten von a und g ist jeweils eine Variable (X) zu finden, sodass $ar(a) = ar(g) = 1$. An der Eingangskante von b befinden sich sogar zwei Variablen (X und Y). Also ist $ar(b) = 2$.

Definition 14 Ist $\mathcal{N} = (P, T, F, L)$ ein pr/t-Netz, so ist die *Domain* $k \in P \cup T$ definiert als eine Menge der Konstanten $ar(k)$-Tupel. Für $k \in P$ ($k \in T$) ist $C(k)$ die Menge der *Stellenfarben* (*Transitionsfarben*).

[Laut02-2]

Auf das Netz in Abb. 2.3 übertragen bedeutet dies

- $C(s) = \{< a >, < b >\}$,
- $C(a) = \{< a >, < b >\}$,
- $C(b) = \{< a, b >, < b, a >\}$,

- $C(g) = \{< a >, < b >\}$,
- $C(p_1) = \{< a >, < b >\}$ und
- $C(p_2) = \{< a >, < b >\}$.

Definition 15 Sei $\mathcal{N} = (P, T, F, L)$ ein pr/t-Netz, $t \in T$ eine Transition und M eine Markierung von $\mathcal{N}$. Weiterhin sei $\alpha \in C(t)$ eine Färbung von t und $L(p,t)(\alpha)$ sei eine Multimenge, die man von $L(p,t)$ durch Substitution der Variablen durch die korrespondierenden Elemente von α erhält. t ist *aktiviert* oder *feuerbar* für α unter M, gdw.

$$\forall p \in {}^\bullet t : M(p) \geq L(p,t)(\alpha).$$

[Laut02-2]

Auf das Beispielnetz übertragen, heißt dies, dass a aktiviert ist für $\alpha \in \{< a >, < b >\}$, da ${}^\bullet a = \{p_1\}$ und für $\alpha =< a >$ ($\alpha =< b >$) gilt, dass

$$M(p_1) = \{< a >, < b >\} \geq \{< a >\} = L(p_1, a)(a)$$

$$(M(p_1) = \{< a >, < b >\} \geq \{< b >\} = L(p_1, a)(b)).$$

Definition 16 Wenn eine Transition t für α unter M aktiviert ist, kann t feuern. Durch Feuern von t entsteht eine neue Markierung M' mit

$$M' = \begin{cases} M(p) - L(p,t)(\alpha) & \text{wenn } p \in {}^\bullet t \backslash t^\bullet, \\ M(p) + L(t,p)(\alpha) & \text{wenn } p \in t^\bullet \backslash {}^\bullet t, \\ M(p) - L(p,t)(\alpha) + L(t,p)(\alpha) & \text{wenn } p \in {}^\bullet t \cap t^\bullet \text{ und} \\ M(p) & \text{sonst.} \end{cases}$$

[Laut02-2]

2.3 Hierarchische Netze

In der graphischen Darstellung werden p/t-Netze (und somit auch pr/t-Netze) schnell sehr groß und unübersichtlich. Dies kann vermieden werden, wenn hierarchische p/t-Netze verwendet werden. So erhalten die Netze eine Struktur, die einem prozeduralen Programm ähnelt. Um Teile des Netze zu realisieren, werden diese in Subnetzen dargestellt, analog zu Prozeduren. Jedes Subnetz besitzt eine oder mehrere *Eingabestellen*, die die Eingabeschnittstelle realisieren, und eine oder mehrere *Ausgabestellen*, die die Ausgabeschnittstelle realisieren. An diese Eingabestellen werden dann die Parameter in Form von Marken (bzw. Tupeln bei pr/t-Netzen) übergeben, mit denen das Subnetz vom umgebenden Netz aufgerufen wird. Nachdem das Subnetz fertig gerechnet hat, liegen auf den Ausgabestellen, wiederum in Form von Marken bzw. Tupeln, die Ergebnisse der Berechnung. Diese werden so an das aufrufende bzw. umgebende Netz zurückgegeben. Jedes Subnetz erhält auch noch einen eindeutigen Namen, anhand dessen es später von anderen Subnetzen unterschieden werden kann. Es folgt ein Beispiel eines Netzes mit einem Subnetz.

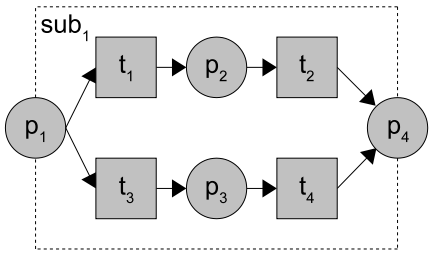


Abb. 2.4: Subnetz-Beispiel

Beispiel 3 Das p/t-Netz in Abb. 2.4 stellt ein Beispiel eines Subnetzes dar. Der Name des Netzes sub_1 steht in der oberen linken Ecke des Netzes. Die Ein- und die Ausgabeschnittstelle bestehen jeweils aus einer einziger Stelle p_1, respektive p_4. Zur besseren graphischen Abgrenzung ist das Netz von einer gestrichelten Linie umgeben, die nur von den Ein- und Ausgabestellen unterbrochen wird.

□

In dem aufrufenden Netz wird das Subnetz nur durch die Ein- und Ausgabestellen, die über eine neue Transition verbunden sind, und durch den Namen dargestellt. Das Feuern dieser Transition wird durch das Feuern der Transitionen des Subnetzes realisiert, d. h. wenn das Subnetz fertig gerechnet hat und ein Ergebnis auf den Ausgabestellen liegt, hat es von außen den Anschein, als hätte die neue Transition gefeuert. Auch in dieser Darstellung wird das Subnetz durch eine gestrichelte Linie von der Umgebung abgegrenzt. Als Beispiel hierfür wird das Subnetz aus Beispiel 3 in ein umgebendes Netz integriert.

Beispiel 4 Die Umgebung des p/t-Netzes in Abb. 2.5 besteht hier nur aus zwei Transitionen t_{ein} und t_{aus}. Transition t_{ein} aktiviert das Subnetz, Transition t_{aus} ist aktiviert, sobald das Subnetz eine Ausgabe erzeugt und diese auf der Ausgabestelle p_4 abgelegt hat.

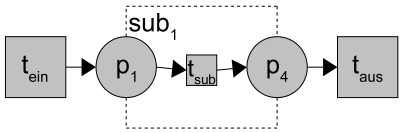


Abb. 2.5: Netz mit einem Subnetz

Abschließend sei noch gezeigt, dass das Netz auch ohne Subnetze realisiert werden kann. Die zum Netz in Abb. 2.5 äquivalente Darstellung ohne Subnetze ist in Abb. 2.6 gegeben.

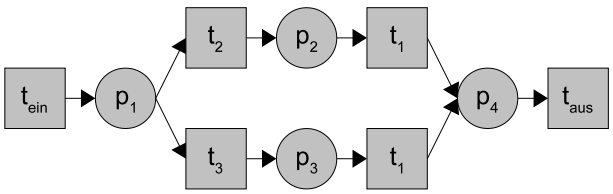


Abb. 2.6: Äquivalente Darstellung ohne Subnetz

□

2.4 Workflow-Netze

Um die bisher definierten p/t-Netzen und pr/t-Netzen in Workflow-Netze umzuwandeln werden zwei besondere Stellen benötigt. Davon wird die erste durch eine neu hinzugefügte Starttransition markiert. Eine zweite hinzugefügte Transition ist dann aktiviert, wenn die zweite der beiden Stellen ausreichend markiert ist. Durch Hinzufügen einer zusätzlichen Stelle zwischen diesen beiden Transitionen muss das Netz stark zusammenhängend werden, um ein Workflow-Netz zu sein. Abschließend wird noch der im weiteren Verlauf dieser Arbeit verwendete Begriff „vernünftig“ definiert. Die Definitionen im Abschnitt „p/t-Netze als Workflow-Netze“ sind an [Aals98] angelehnt; allerdings um eine Starttransition s und eine Zieltransition g erweitert, um die in [Laut01] beschriebene Reproduzierbarkeit der leeren Markierung für die spätere Inferenz zu ermöglichen.

2.4.1 p/t-Netze als Workflow-Netze

Definition 17 Sei $\mathcal{N} = (P, T, F)$ ein p/t-Netz. $\mathcal{N}$ ist ein Workflow-Netz gdw.

- $\mathcal{N}$ zwei spezielle Transitionen $s, g \in T$ besitzt mit ${}^\bullet s = \emptyset$, $s^\bullet = p_i$, ${}^\bullet g = p_o$ und $g^\bullet = \emptyset$,
- $\mathcal{N}$ zwei spezielle Stellen $p_i, p_o \in P$ besitzt mit ${}^\bullet p_i = s$ und $p_o^\bullet = g$ und
- das Netz $\mathcal{N}^* = (P^*, T^*, F^*)$ mit
  - $P^* = P \cup p^*$ wobei $p^* \notin P$,
  - $T^* = T$ und
  - $F^* = F \cup \{(g, p^*)\} \cup \{(p^*, s)\}$

  stark zusammenhängend ist.[2]

□

Bevor der Begriff der Vernunft eingeführt werden kann, sollen noch zwei spezielle Markierungen von Workflow-Netzen eingeführt werden. Die erste ist die Eingabemarkierung zu Beginn des Prozesses, die auf allen Stellen bis auf p_i unmarkiert ist. Auf p_i selbst hingegen liegt genau eine Marke. Die zweite ist die Ausgabemarkierung, die überall unmarkiert ist bis auf p_o, auf der genau wie auf p_i bei der Eingabemarkierung genau eine Marke liegt. Formal ausgedrückt heißt dies:

Definition 18 Sei $\mathcal{N} = (P, T, F)$ ein Workflow-Netz.

- Eine Markierung M_i von $\mathcal{N}$ heißt *Eingabemarkierung* (engl. „input marking“) von $\mathcal{N}$, wenn
  - $M_i(p_i) = 1$,
  - $M_i(p_j) = 0 \ \forall p_j \in P/p_i$.
- Eine Markierung M_o von $\mathcal{N}$ heißt *Ausgabemarkierung* (engl. „output marking“) von $\mathcal{N}$, wenn
  - $M_o(p_o) = 1$,

[2] D.h. eine Stelle wird hinzugefügt, die g mit s verbindet.

– $M_o(p_j) = 0 \; \forall p_j \in P/p_o$.

□

Mit Hilfe dieser beiden Begriffe lässt sich nun definieren, wann ein Workflow-Netz vernünftig ist.

Definition 19 Sei $\mathcal{N} = (P, T, F)$ ein Workflow-Netz. M_i sei Eingabemarkierung und M_o sei Ausgabemarkierung von $\mathcal{N}$. $\mathcal{N}$ heißt *vernünftig* (engl. „sound“) gdw.

- $\forall M \in [M_i\rangle : \; M_o \in [M\rangle$,
- $\forall M \in [M_i\rangle : \; M(p_o) \geq 1 \Rightarrow M = M_o$ und
- $\forall t \in T, \exists M \in [M_i\rangle : \; M[t\rangle$.

□

Ein Workflow-Netz ist also vernünftig, wenn es zu Beginn bis auf p_i leer ist und dort nur eine Marke liegt, wenn eine Marke auf p_o liegt, der Rest des Netzes leer ist und die Abarbeitung somit abgeschlossen ist und wenn für jede Transition eine Markierung erreicht werden kann, in der diese aktiviert ist.

2.4.2 pr/t-Netze als Workflow-Netze

Äquivalent zu den Definitionen im vorigen Abschnitt werden die zuvor eingeführten Begriffe hier auf pr/t-Netze übertragen.

Definition 20 Sei $\mathcal{N} = (P, T, F, L)$ ein pr/t-Netz. $\mathcal{N}$ ist ein Workflow-Netz gdw.

- $\mathcal{N}$ zwei spezielle Transitionen $s, g \in T$ besitzt mit ${}^\bullet s = \emptyset$, $s^\bullet = p_i$, ${}^\bullet g = p_o$ und $g^\bullet = \emptyset$,
- $\mathcal{N}$ zwei spezielle Stellen $p_i, p_o \in P$ besitzt mit ${}^\bullet p_i = s$ und $p_o^\bullet = g$ und
- das Netz $\mathcal{N}^* = (P^*, T^*, F^*, L^*)$ mit
  - $P^* = P \cup p^*$ wobei $p^* \notin P$,
  - $T^* = T$,
  - $F^* = F \cup \{(g, p^*)\} \cup \{(p^*, s)\}$ und
  - $L^* : F^* \rightarrow \{G | G : MS_+ \rightarrow MS_+\}$, wobei $MS_+ = \bigcup_{n \geq 0} Y_n$ und der Wert von $L^*(x, y)$ dabei bestimmt ist durch

$$L^*(x,y) = \begin{cases} L(x,y) & \text{wenn } (x,y) \in F, \\ <> & \text{wenn } (x,y) = (g,p^*) \text{ und} \\ <> & \text{wenn } (x,y) = (p^*,s) \end{cases}$$

stark zusammenhängend ist.

□

Wie bei p/t-Netzen wird auch hier eine Stelle hinzugefügt, die g mit s verbindet. L^* nimmt für alle Werte, die auch schon durch L abgebildet werden, den gleichen Wert wie L an. An den beiden neuen Kanten wird jeweils auf das leere Tupel <> abgebildet.

Die Begriffe Eingabemarkierung und Ausgabemarkierung werden wie oben definiert, jedoch mit dem Unterschied, dass man nicht genau festlegen kann, welche Tupel auf p_i bzw. p_o liegen müssen. Also wird auf den Begriff der Markiertheit von Stellen in pr/t-Netzen zurückgegriffen.

Definition 21 Sei $\mathcal{N} = (P, T, F, L)$ ein Workflow-Netz.

- Eine Markierung M_i von $\mathcal{N}$ heißt *Eingabemarkierung* (engl. „input marking“) von $\mathcal{N}$, wenn
  - $M_i(p_i)$ markiert ist und
  - $M_i(p_j)$ unmarkiert ist $\forall p_j \in P/p_i$.
- Eine Markierung M_o von $\mathcal{N}$ heißt *Ausgabemarkierung* (engl. „output marking“) von $\mathcal{N}$, wenn
  - $M_o(p_o)$ ist markiert und
  - $M_o(p_j)$ ist unmarkiert $\forall p_j \in P/p_o$.

□

Jetzt kann auch für Workflow-Netze, die auf pr/t-Netzen basieren, der Begriff „vernünftig“ definiert werden.

Definition 22 Sei $\mathcal{N} = (P, T, F, L)$ ein Workflow-Netz. M_i sei Eingabemarkierung und M_o sei Ausgabemarkierung von $\mathcal{N}$. $\mathcal{N}$ heißt *vernünftig* (engl. „sound“) gdw.

- $\forall M \in [M_i\rangle : M_o \in [M\rangle$,
- $\forall M \in [M_i\rangle : M(p_o)$ *ist markiert* $\Rightarrow M = M_o$ und
- $\forall t \in T, \exists M \in [M_i\rangle : M[t\rangle$.

□

Der Begriff „Workflow-Netz“ unterscheidet also nicht zwischen p/t-Netzen und pr/t-Netzen. Wenn im weiteren Verlauf dieser Arbeit von Workflow-Netzen gesprochen wird, sollte aus dem Zusammenhang deutlich sein, ob es sich um ein auf p/t-Netzen oder pr/t-Netzen basierendes Workflow-Netz handelt. Ansonsten wird an betroffenen Stellen ausdrücklich gesagt, um welchen der beiden Typen es sich bei dem Workflow-Netz handelt.

3 Workflows

Nachdem im vorigen Kapitel einige Grundlagen der Netztheorie vorgestellt wurden, sollen in diesem Kapitel die in der Einleitung angesprochenen Workflow-Pattern genauer erläutert werden. Diese Pattern wurden von Wil v. d. Aalst et al. nach Analyse vieler Workflow-Prozess-Definitionen als immer wiederkehrend identifiziert und in [RuHoAaMu] und [AaHoKiBa03] aufgelistet.

Da Workflow-Prozess-Definitionen nicht formal definiert sind, ergeben sich hieraus Probleme bei der automatischen Analyse. Petri-Netze hingegen basieren auf mathematischen Grundlagen und sind formal wohldefiniert. Auch gibt es vielfältige Analysemöglichkeiten für Petri-Netze.

Zunächst wird kurz auf die grafischen Besonderheiten der Workflow-Prozess-Definitionen in [RuHoAaMu] eingegangen. Anschließend werden die einzelnen Pattern kurz beschrieben und als Petri-Netze grafisch dargestellt. Dort, wo es nötig ist (z. B. wenn einzelne Instanzen eines Prozesses voneinander unterscheidbar sein müssen), wird dies durch pr/t-Netze getan. Ansonsten werden p/t-Netze verwendet, um die Netzdarstellungen möglichst einfach zu halten.Durch Hinzufügen von leeren Tupeln an den Kanten können diese jederzeit einfach zu pr/t-Netze erweitert werden.

Nach Vorstellung der Pattern wird am Ende des Kapitels gezeigt, wie die in der Einleitung vorgestellte Workflow-Prozess-Definition aus Abb.1.1 als Petri-Netz dargestellt werden kann.

3.1 Grafische Darstellung

[Aals98] Die grafische Darstellung des in der Einleitung gegebenen Workflows erinnert an die im vorigen Kapitel gegebenen Grafiken von Petri-Netzen. Dies ist nicht überraschend, da Workflow-Prozess-Definitionen aus Petri-Netzen hervorgegangen sind. Es gibt hier wie bei Petri-Netzen zwei verschiedene Arten von Knoten, nämlich Aktionen (hier durch Quadrate - analog zu Transitionen dargestellt), Bedingungen (abgebildet durch Kreise - analog zu Stellen) und gerichtete Kanten (Pfeile), die diese Knoten miteinander verbinden. Zusätzlich existieren aber noch einige wenige andere Zeichen. Zuerst sollen die neuen Aktionssymbole, die noch einmal einzeln in Abb. 3.1 bis 3.3 aufgeführt sind, erläutert werden.

Das Ausführen der dem UND-Split zugehörigen Aktion in Abb. 3.2 macht alle Bedingungen wahr, die an einer der ausgehenden Kanten hängen. D. h., dass auf alle diese Bedingungen eine Marke gelegt wird.

Gegenstück zum UND-Split ist der in Abb. 3.2 dargestellte UND-Join, der nur ausgeführt werden kann, wenn alle Bedingungen, die sich an einer Eingangskante des UND-Join befinden, wahr sind. Während diese beiden Aktionen auch genauso durch eine „normale“ Transition in Petri-Netzen dargestellt werden können, ist dies mit den beiden folgenden Aktionen nicht möglich.

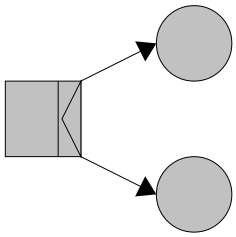


Abb. 3.1: UND-Split

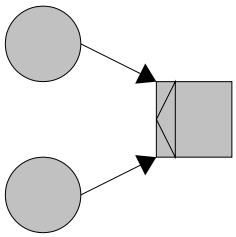


Abb. 3.2: UND-Join

Das Ausführen der Aktion eines XODER-Splits macht genau eine der nachfolgenden Bedingungen wahr. Hierbei wird zwischen dem impliziten und expliziten ODER-Split unterschieden. Bei dem expliziten XODER-Split in Abb. 3.3 ist ausdrücklich an den Kanten angegeben, welche der nachfolgenden Bedingungen wahr wird.

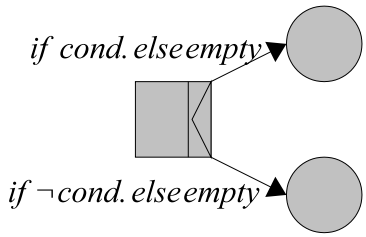


Abb. 3.3: Expliziter ODER-Split

Beim impliziten XODER-Split in Abb. 3.4 ist dies nicht der Fall. Wenn aber eine der beiden Aktionen ausgeführt wird, ist die Stelle nicht mehr markiert, und die andere Aktion kann nicht ausgeführt werden.

Als zugehöriger Join zu beiden XODER-Splits fungiert hier der implizite XODER-Join[1], der ebenfalls wie der implizite XODER-Split in Abb. 3.4 ohne besondere Zeichen dargestellt werden kann.

Bevor nun beispielhaft gezeigt wird, wie ein Workflow in ein Petri-Netz überführt werden kann, werden im nächsten Abschnitt einzelne Workflow-Pattern als Petri-Netz aufgeführt.

[1]Es gibt auch noch einen expliziten XODER-Join. Aber da es beim XODER-Join keinen Unterschied zwischen explizit und implizit gibt und der implizite XODER-Join auch mit Petri-Netzen ohne weiteres modellierbar ist, wird der explizite XODER-Join hier und auch im Folgenden nicht verwendet.

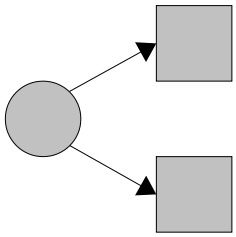


Abb. 3.4: Impliziter ODER-Split

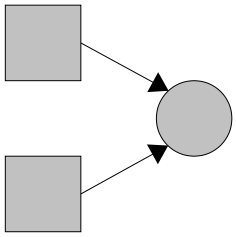


Abb. 3.5: Expliziter ODER-Join

3.2 Workflow-Pattern

Zu jedem Pattern wird eine kurze Erläuterung und anschließend die Darstellung des Pattern als Petri-Netz gegeben. Die Pattern sind hierzu aus [RuHoAaMu, S. 8ff.] entnommen. Von dort sind auch die geklammerten englischen Bezeichnungen genommen. Zudem ist dort auch eine exakte Beschreibung der einzelnen Pattern zu finden. Um sich auf die Übersetzung in Petri-Netze konzentrieren zu können, wurde an dieser Stelle auf eine genauere Erläuterung der Pattern verzichtet. Auf eventuelle Besonderheiten und Probleme bei der Übertragung der Pattern in Petri-Netze wird jedoch bei den einzelnen Pattern eingegangen. In Konkordanz zur Definition von Workflow-Netzen in Abschnitt 2.4 und zu [Laut01] und [Laut02-1] wurden in den Netzen jeweils eine Starttransition s und eine Zieltransition g angefügt. Zudem wurden die Pattern teilweise um einige weitere Knoten und Kanten erweitert, um die Netze vernünftig zu halten. Diese Knoten und Kanten sind im Rest dieses Abschnitt weiß markiert.

Sequenz

Die Aktivität b in Abb. 3.6 wird aktiviert, nachdem die vorhergehende Aktivität a vollständig abgearbeitet wurde.

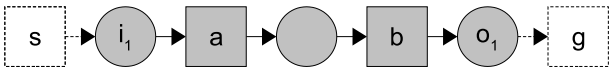


Abb. 3.6: Sequenz

UND-Split (Parallel-Split)

Nachdem Aktivität a in Abb. 3.7 beendet wurde, wird aus dem laufenden Prozess heraus ein neuer Prozess gestartet, sodass die folgenden Aktivitäten b und c parallel ausgeführt werden können.

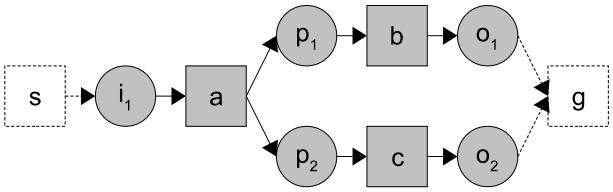


Abb. 3.7: UND-Split

UND-Join (Synchronization)

In Abb. 3.8 wird Aktivität c erst ausgeführt, nachdem alle vorhergehenden Aktivitäten a und b beendet wurden. Die beiden Prozesse a und b werden hierbei zu einem Prozess synchronisiert.

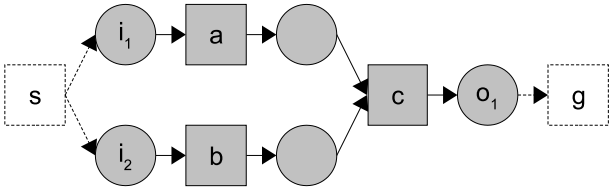


Abb. 3.8: UND-Join

XODER-Split (Exclusive Choice)

Nach Ausführen von Aktivität a wird eine der beiden nachfolgenden Aktivitäten b oder c ausgeführt, die andere wird hingegen nicht ausgeführt. Der in Abb. 3.9 abgebildete Split ist implizit, d. h. es ist im Netz spezifiziert, unter welcher Bedingung b ausgeführt wird und wann c.

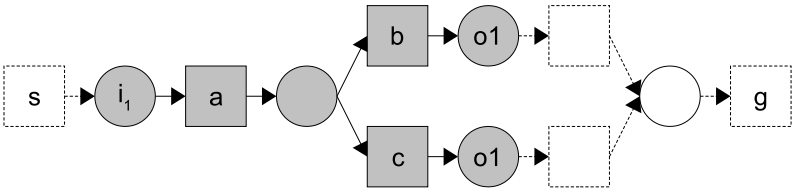


Abb. 3.9: XODER-Split

XODER-Join (Simple Merge)

Nachdem eine der beiden Aktivitäten a oder b ausgeführt wurde, steht fest, dass die andere nicht ausgeführt wird. Es wird direkt mit der folgenden Aktivität c fortgefahren. Dies ist in Abb. 3.10 dargestellt.

ODER-Split (Multi-Choice)

Im Gegensatz zum vorhergehenden XOR-Split können in Abb. 3.11 auch die Aktivitäten b und c ausgeführt werden. Da es so möglich ist, dass nicht nur eine, sondern auch zwei

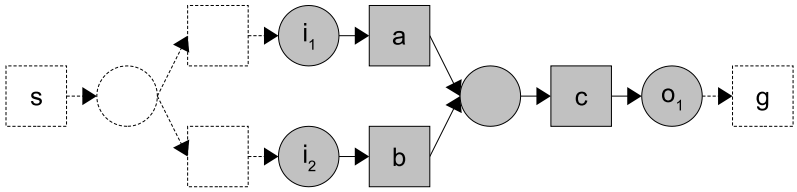


Abb. 3.10: XODER-Join

Marken auf den Ausgangsstellen o_1 und o_2 liegen, ist es notwendig, diesen Sachverhalt an einer gesonderten Stelle zu speichern, um mit einem später folgenden ODER-Join (s. u.) wieder so synchronisieren zu können, dass daraus ein vernünftiges Workflow-Netz entstehen kann. Dafür werden drei zusätzliche Stellen o'_1, o'_2 und o'_3 benötigt, von denen jeweils eine markiert ist. Nämlich o'_1, wenn nur b ausgeführt wird, o'_2 wenn b und c ausgeführt werden und o'_3 wenn nur c ausgeführt wird. Diese Stellen werden später bei dem zugehörigen ODER-Join als Eingangsstellen verwendet.

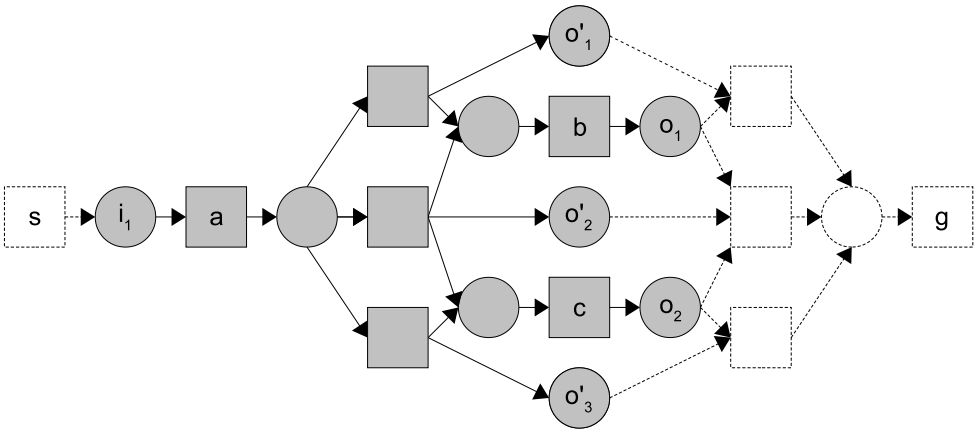


Abb. 3.11: ODER-Split

ODER-Join (Synchronising Merge)

Nachdem Aktivität a oder b oder auch beide ausgeführt wurden, wird Aktivität c ausgeführt. Die im vorigen Abschnitt beschriebenen Stellen o'_1, o'_2 und o'_3 werden in Abb. 3.12 respektive wie oben beschrieben als Eingangsstellen i'_1, i'_2 und i'_3 verwendet.

Multipler-Join (Multi-Merge)

Wenn eine der Aktivitäten a oder b in Abb. 3.13 (oder beide) ausgeführt wurde(n), wird anschließend Aktivität c ausgeführt. (Diese kann auch zweimal ausgeführt werden.)

Diskriminator

Der Diskriminator folgt auf einen UND-Split. Hierbei werden zwei Untertypen unterschieden, nämlich der in Abb. 3.14 abgebildete „1 aus m“- und der in Abb. 3.15 stehende „n aus m“-Diskriminator. In den hier abgebildeten Netzen ist m = 2.

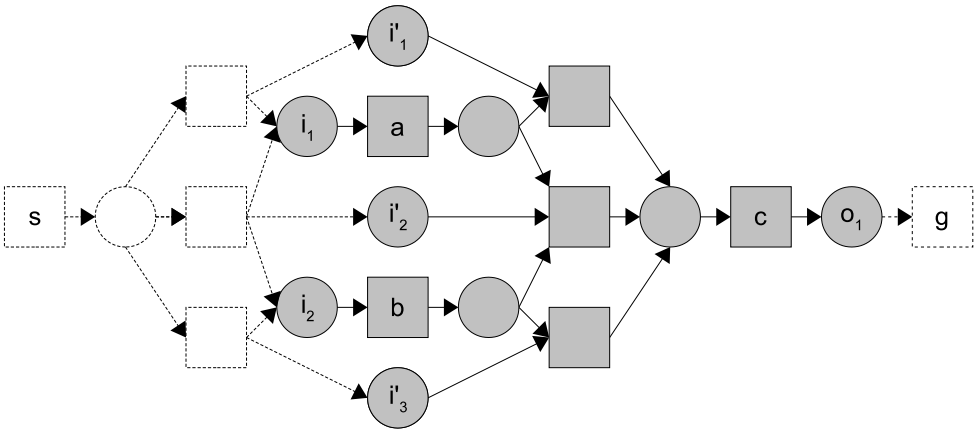


Abb. 3.12: ODER-Join

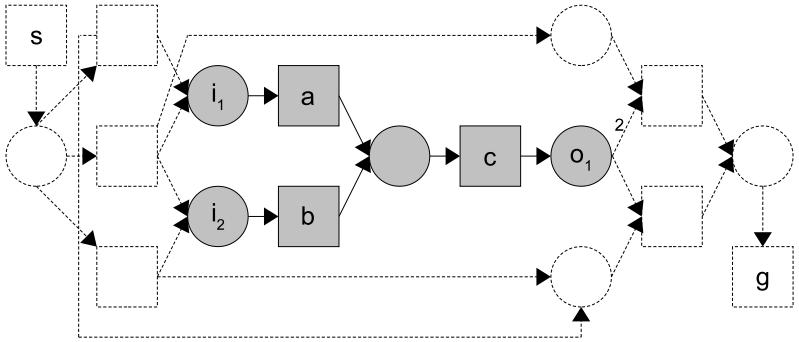


Abb. 3.13: Multipler Join

„1 aus m“-Diskriminator

Nachdem die Aktivitäten a und b ausgeführt wurden, wird eines der beiden Ergebnisse[2] ausgewählt und von der Aktivität c als Eingangsmarke verwendet. Nach Ausführen von c ist die Transition t_1 aktiviert. Diese feuert nun so lange, bis keine Marken mehr auf p_1 vorhanden sind. Anschließend wird das Ergebnis von c mit Hilfe der Transition t_2 an die Ausgangsstelle weitergeleitet und die Marke von p_3 wird entfernt.

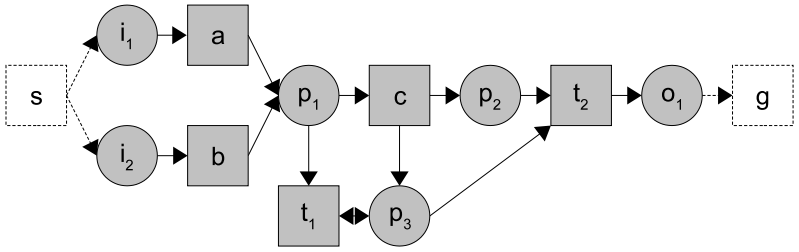


Abb. 3.14: Diskriminator 1 aus m

„n aus m“-Diskriminator

Der „n aus m“-Diskriminator funktioniert genau wie der „1 aus m“-Diskriminator mit dem Unterschied, dass die Aktivität c die Ergebnisse von n vorhergehenden Aktivitäten als

[2] De Ergebnisse können auch beliebige Tupel sein (s. o.).

Eingangsmarken verwendet.

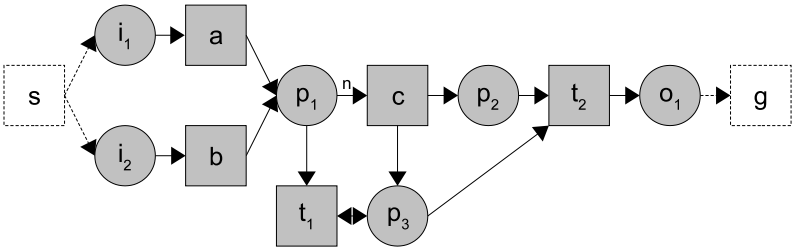


Abb. 3.15: Diskriminator n aus m

Arbiträre Zykel

Arbiträre Zykel wie in Abb. 3.16 sind Zykel, die mehr als einen Eingangs- oder Ausgangspunkt haben. Zum Beispiel besitzt der Zykel, der über b und c läuft, zwei Eingangspunkte, nämlich p_1 und p_2. Der über c und d laufende Zykel hingegen besitzt zwei Ausgangspunkte, nämlich t_1 und t_2.

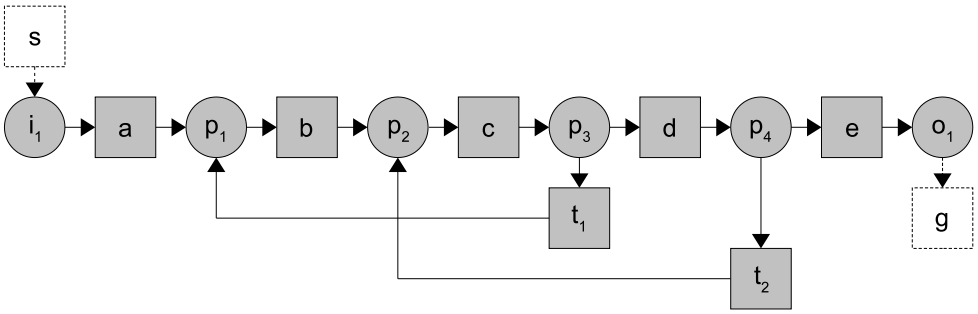


Abb. 3.16: Arbiträre Zykel

Multiple Instanzen ohne Synchronisierung

Nach Ausführen von a werden in Abb. 3.17 n Instanzen des von a produzierten Ergebnisses erzeugt. So kann die auf a folgende Aktivität b n-mal ausgeführt werden.

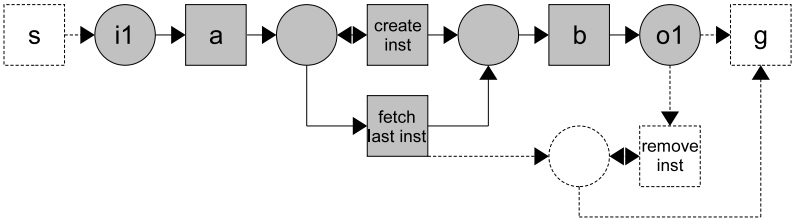


Abb. 3.17: Multiple Instanzen ohne Synchronisierung

Multiple Instanzen mit Entwurfszeitwissen

Wie im vorigen Abschnitt werden auch in Abb. 3.18 n Instanzen des Ergebnisses der Aktivität a erzeugt. Nach der n-fachen Ausführung von b werden diese Ergebnisse wieder synchronisiert, bevor die abschließende Aktivität c ausgeführt wird.

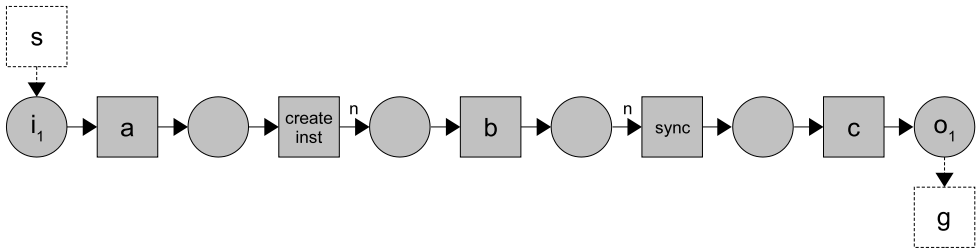


Abb. 3.18: Multiple Instanzen mit Wissen zur Entwurfszeit

Multiple Instanzen mit Laufzeitwissen

Wenn erst zur Laufzeit bekannt ist, wieviele Instanzen eines Prozesses erzeugt werden sollen, muss wie in Abb. 3.19 gezeigt die Erzeugung der Instanzen variabel gehalten werden. Statt einer simultanen Erzeugung wird dann eine sequentielle verwendet. Die spezielle Eingabestelle i'_1 enthält hierbei zu Beginn der Generierung die Anzahl der zu erstellenden Instanzen von b.

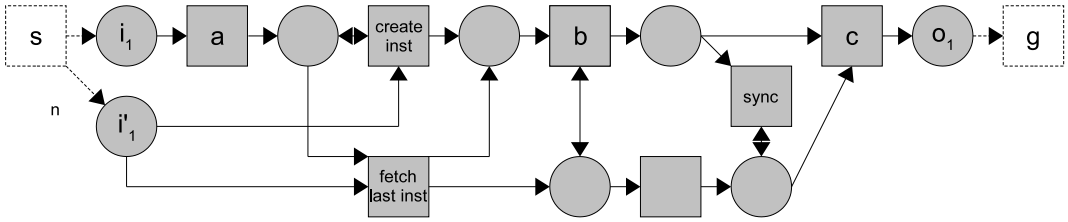


Abb. 3.19: Multiple Instanzen mit Wissen zur Laufzeit

Verzögerter Split (Deferred Choice)

Beim verzögerten Split in Abb. 3.20 wird die Wahl, welche der beiden Aktivitäten b und c ausgeführt wird, möglichst lange hinausgezögert. Spätestens bei Ausführen einer der beiden Aktivitäten steht dies aber fest. So ist der verzögerte Split gleich dem impliziten XODER-Split, bei dem auch die geXODERten Aktionen selbst entscheiden, welche Aktion ausgeführt wird.

Verschachtelt paralleles Routing (Interleaved Parallel Routing)

In Abb. 3.21 können die Aktivitäten a_1 bis a_n aufgrund einer zu teilenden Ressource nur verschachtet parallel ausgeführt werden, d. h. sie können nicht gleichzeitig ausgeführt werden. Die Reihenfolge der Ausführung ist aber nicht vorgegeben. Bei Initialisierung werden Aktivierungsmarken auf die zu a_1 bis a_n gehörigen Stellen p_1 bis p_n gelegt. Auch

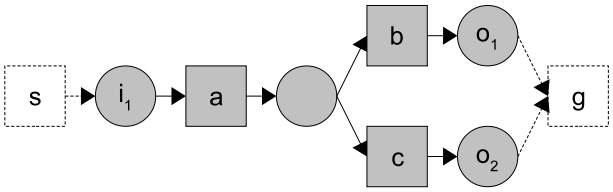


Abb. 3.20: Verzögerter Split

die zu teilende Ressource wird auf der Stelle p_0 zur Verfügung gestellt. Nach Ausführung aller Aktivitäten a_1 bis a_n kann dann mit der Aktivität b fortgefahren werden.

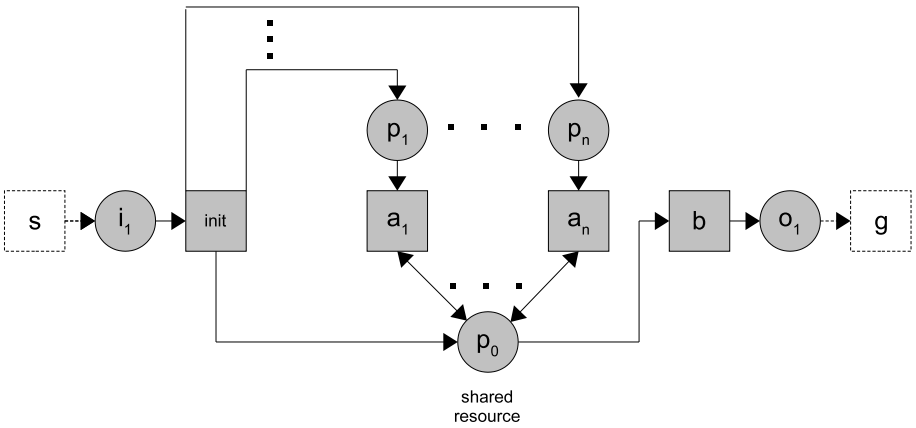


Abb. 3.21: Verschachtelt paralleles Routing

Meilenstein

Wenn es wie im Netz in Abb. 3.22 mehrere parallel ablaufende Prozesse gibt, können manche Aktivitäten (hier c) erst ausgeführt werden, wenn eine Aktivität eines parallel laufenden Prozesses (hier a) ausgeführt wurde.

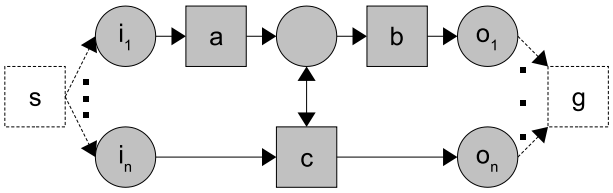


Abb. 3.22: Meilenstein

Aktivität abbrechen

Manchmal ist es nötig, eine bereits laufende Aktivität b abzubrechen und evtl. bereits produzierte Ergebnisse zu entfernen. Dazu wird in Abb. 3.23 von Aktivität a nicht nur eine Ausgabe produziert, die als Eingabe für Aktivität b dient, sondern es wird gleichzeitig eine Marke auf die Stelle p_4 gelegt, die ein Abbrechen von b ermöglicht. b kann nun sowohl

vor Beginn der Ausführung von b abgebrochen werden, als auch nach Beginn aber vor Ende von b.

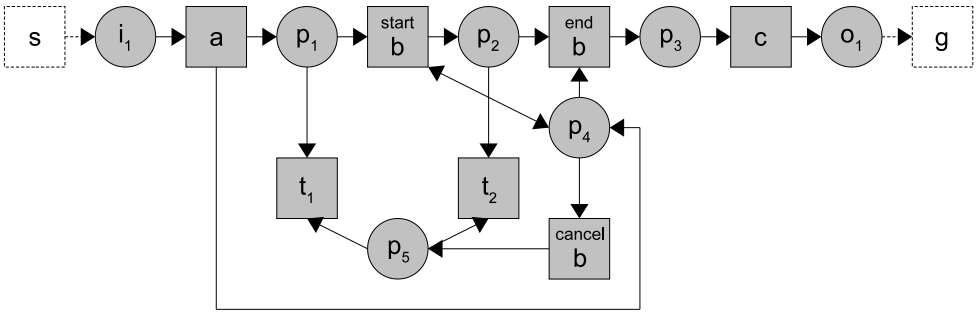


Abb. 3.23: Aktivität abbrechen

Instanz abbrechen (Cancel Case)

Soll nicht eine bestimmte Aktivität abgebrochen werden, sondern die gesamte Instanz eines Prozesses (d. h. es sollen alle zu einem Prozess gehörigen Aktivitäten eingestellt werden), wird das in Abb. 3.24 gezeigte Pattern verwendet. Die Instanzen müssen von anderen Instanzen des Prozesses anhand einer ID unterscheidbar sein. Dafür wird zunächst eine Kopie der ID in p_1' gespeichert und der Prozess aktiviert. Soll die Instanz nun abgebrochen werden, so wird die Aktivität „start cancel“ durchgeführt. Danach werden alle zur Instanz gehörigen Marken durch die Transitionen *cancel* p_i aus dem Prozess entfernt. Ist der Abbruch vollständig ausgeführt, feuert die Transition *end cancel* und das Ergebnis wird dann an die Ausgangsstelle o_1 weitergereicht.

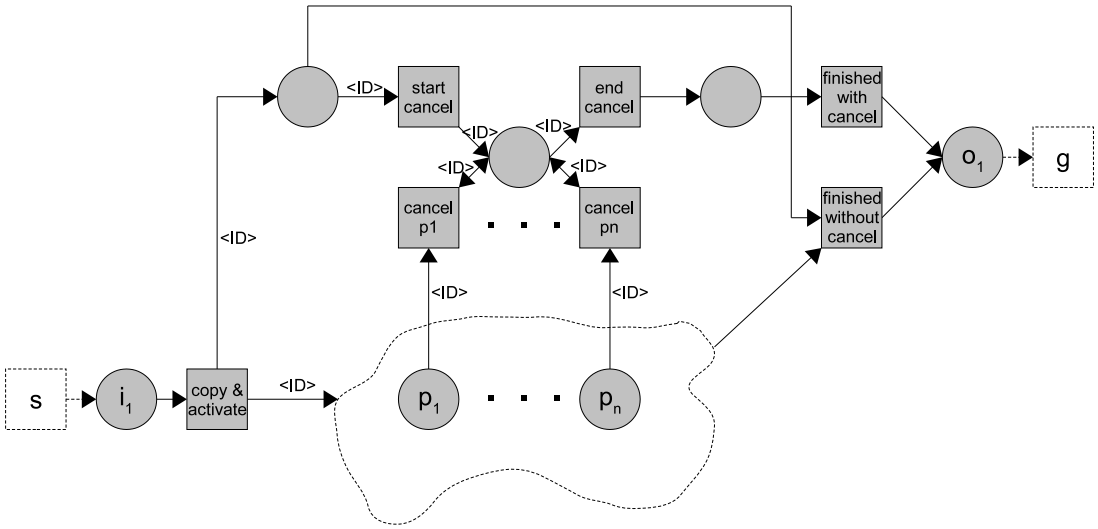


Abb. 3.24: Instanz abbrechen

3.3 Beispiel

Mit dem in den beiden vorigen Abschnitten gegebenen Wissen ist es nun möglich, den Workflow aus Abb. 1.1 als Petri-Netz darzustellen. Das Ergebnis wird in Abb. 3.25 gezeigt.

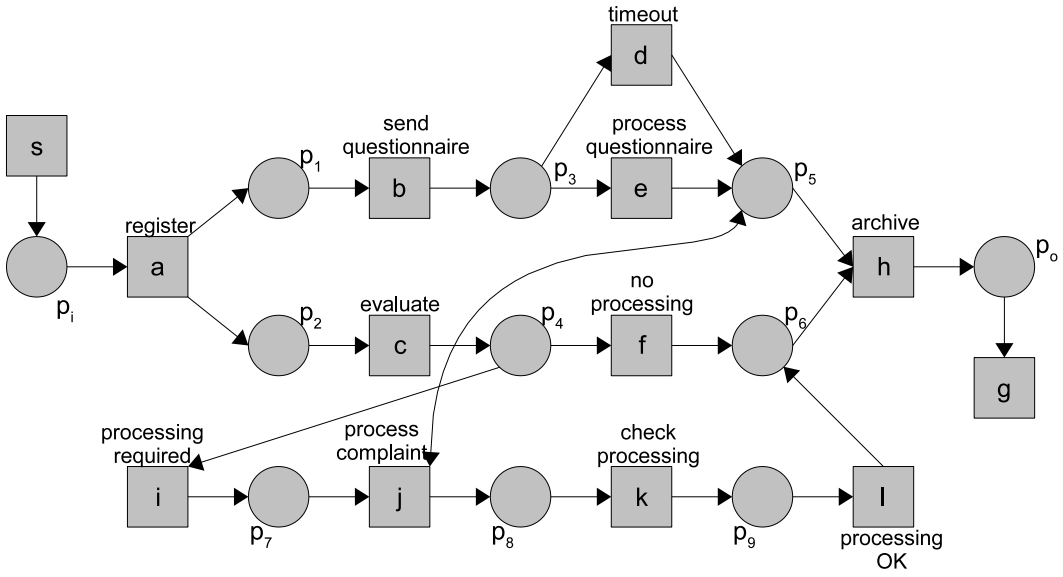


Abb. 3.25: Reklamtionsprozess als p/t-Netz

4 Transformationen

Bisher wurden die für diese Arbeit nötigen Begriffe der Netztheorie vorgestellt und der Begriff Workflow-Netz formal definiert. In Kapitel 3 wurden dann die von W. v. d. Aalst in [RuHoAaMu] angeführten Workflow-Pattern in die zuvor definierte Form eines Workflow-Netzes transformiert. In diesem Kapitel sollen nun für verschiedene Arten der Logik Methoden vorgestellt werden, mit denen die logische Darstellung eines Sachverhalts in Petri-Netze übertragen werden kann. Auch der umgekehrte Weg, also zu einem Petri-Netz eine äquivalente logische Formel zu konstruieren, wird beschrieben.

Der Weg führt dabei von der Aussagenlogik in Abschnitt 4.1 über die Prädikatenlogik in Abschnitt 4.2 und die Transaktionslogik in Abschnitt 4.3 bis zur Aktionslogik in Abschnitt 4.4. Dabei müssen teilweise bei den zu übertragenden Formeln bzw. Petri-Netzen Einschränkungen getroffen werden, um diese übertragen zu können. Diese Einschränkungen werden dann jeweils ausdrücklich erwähnt.

Dazu werden jeweils im ersten Unterabschnitt Definitionen aufgelistet, die die Syntax und Semantik der einzelnen Logik-Arten festlegen und zentrale Begriffe definieren. Anschließend wird im zweiten Unterabschnitt gezeigt, wie Formeln der verschiedenen Logiken als Petri-Netz dargestellt (implementiert) werden können, bevor schließlich im letzten Unterabschnitt der umgekehrte Weg der Implementierung von Petri-Netzen als Formel beschritten wird. Dazu werden auch zahlreiche Beispiele aufgeführt, die helfen sollen, die verschiedenen Methoden besser verstehen zu können.

Das gesamte Kapitel wird dann in Abschnitt 4.5 durch zwei größere Beispiele abgeschlossen.

4.1 Aussagenlogik

Die Aussagenlogik bietet sich als einfachste und bekannteste der hier behandelten Logiken als Einstieg an. Jede Aussage der Aussagenlogik ist entweder wahr oder falsch. Dabei kann eine Aussage aus anderen Aussagen zusammengesetzt sein, sodass komplexere Aussagen formuliert werden können. Die notwendigen Begriffe stehen in Abschnitt 4.1.1 und sind aus [Laut02-1] entnommen. In [Laut02-1] ist auch bereits ein Weg beschrieben, wie eine bestimmte Klasse aussagenlogischer Formeln als p/t-Netze dargestellt werden kann. Dies wird hier der Vollständigkeit halber in Abschnitt 4.1.2 zitiert. Im anschließenden Abschnitt 4.1.3 wird dann der umgekehrte Weg beschrieben, also wie p/t-Netze als aussagenlogische Formel implementiert werden können.

4.1.1 Definitionen und wichtige Begriffe der Aussagenlogik

Zunächst muss festgelegt werden, aus welchen Zeichen aussagenlogische Formeln bestehen dürfen.

Definition 23 Das *Alphabet der Aussagenlogik* besteht aus

- einer zählbaren Menge von Atomen $\{x_1, x_2, x_3, \dots\}$,

- den Operatoren
  - $\wedge$ (logisches UND),
  - $\vee$ (logisches ODER) und
  - $\neg$ (logisches NICHT) und
- den Klammern '(' und ')'.

[Laut02-1]

Von diesem Alphabet ausgehend kann nun gesagt werden, wie diese Formeln aufgebaut sein müssen.

Definition 24 Sei $X = \{x_1, \ldots x_n\}$ eine Menge von Atomen. Die *Menge der aussagenlogischen Formeln* $\mathcal{F}_{AL}(X)$ über X ist induktiv definiert als

- $\forall x \in X : x \in \mathcal{F}_{AL}(X)$,
- $\alpha, \beta \in \mathcal{F}_{AL}(X) \Rightarrow (\alpha \wedge \beta), (\alpha \vee \beta) \in \mathcal{F}_{AL}(X)$ und
- $\alpha \in \mathcal{F}_{AL}(X) \Rightarrow \neg\alpha \in \mathcal{F}_{AL}$.

[Laut02-1]

Darüber hinaus gibt es noch zwei wichtige und häufig gebrauchte Abkürzungen: Die Implikation und die Äquivalenz.

Definition 25 Seien α und β aussagenlogische Formeln. Die *Implikation* $\alpha \Rightarrow \beta$ ist eine Abkürzung für $\neg\alpha \vee \beta$. Die *Äquivalenz* $\alpha \Leftrightarrow \beta$ zweier Formeln ist eine Abkürzung für $((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$.

[Laut02-1]

Um die Lesbarkeit von Formeln zu erhöhen, wird die gewöhnliche Operatorreihenfolge[1] verwendet und Klammern werden, wenn möglich, weggelassen.

Um den Wert einer Formel bestimmen zu können, wird noch die Menge aller Atome benötigt, die in einer Formel vorkommen.

Definition 26 Ist α eine Formel, so bezeichnet $\mathbb{A}(\alpha)$ die *Menge der Atome*, die in α enthalten sind.

[Laut02-1]

Außerdem wird eine Funktion benötigt, mit der man die Aussage treffen kann, ob ein Atom wahr oder falsch ist.

Definition 27 (**Interpretation**) Ist α eine aussagenlogische Formel, so ist die Abbildung $I : \mathbb{A}(\alpha) \rightarrow \{\textit{wahr, falsch}\}$ eine *Interpretation* von α.

[Laut02-1]

[1] D. h. '$\wedge$' hat eine höhere Priorität als '$\vee$'.

Mit Hilfe dieser Funktion können auch zusammengesetzte Formeln wie folgt evaluiert werden.

Definition 28 (**Semantik der Aussagenlogik**) Seien α und β aussagenlogische Formeln, und I eine Interpretation von α und von β. Der Wert I einer Formel wird bestimmt mit

- $I(\alpha \wedge \beta) = \begin{cases} wahr, & \text{wenn } (I(\alpha) = wahr) \wedge (I(\beta) = wahr) \text{und} \\ falsch, & \text{sonst,} \end{cases}$
- $I(\alpha \vee \beta) = \begin{cases} wahr, & \text{wenn } (I(\alpha) = wahr) \vee (I(\beta) = wahr) \text{und} \\ falsch, & \text{sonst,} \end{cases}$
- $I(\neg\alpha) = \begin{cases} wahr, & \text{wenn } I(\alpha) = falsch \text{und} \\ falsch, & \text{sonst.} \end{cases}$

□

Zuletzt bedarf es noch der Definition einer bestimmte Form von Formeln, da nur diese in p/t-Netz transformiert werden können.

Definition 29 Ein *Literal* ist ein Atom oder die Negation eines Atoms. Eine *Klausel* ist eine Disjunktion (ODER-Verknüpfung) von Literalen. Eine Formel F ist in *konjunktiver Normalform (KNF)*, wenn sie eine Konjunktion (UND-Verknüpfung) von Klauseln ist. Ist eine Formel α in KNF, so bezeichnet $\mathbb{C}(\alpha)$ die *Menge der Klauseln*, die in F enthalten sind.

[Laut02-1]

4.1.2 Transformation der Aussagenlogik in p/t-Netze

In diesem Abschnitt wird gezeigt, wie eine beliebige in KNF stehende aussagenlogische Formel α als p/t-Netz implementiert werden kann. Die Methode ist aus [Laut02-1] entnommen und wird hier nur aus Gründen der Vollständigkeit mit aufgelistet.

Zuerst wird die Struktur der Formel in eine Netzstruktur übertragen.

Definition 30 Sei α eine aussagenlogische KNF-Formel. Das p/t-Netz $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha)$ ist die *p/t-Netz-Implementierung*[2] von α gdw.

- $P_\alpha = \mathbb{A}(\alpha)$,
- $T_\alpha = \mathbb{C}(\alpha)$ und
- $\forall\, \tau = (\neg a_1 \vee \cdots \vee \neg a_m \vee b_1 \vee \cdots \vee b_n) \in \mathbb{C}(\alpha)$ wobei $\{a_1 \vee \cdots \vee a_m \vee b_1 \vee \cdots \vee b_n\} \in \mathbb{A}(\alpha)$, wird F_α bestimmt durch ${}^\bullet\tau = \{a_1, \ldots, a_m\}$ und $\tau^\bullet = \{b_1, \ldots b_n\}$.

[Laut02-1]

Außerdem muss noch die Bewertung einer Formel, d. h. ihre Interpretation in das implementierte p/t-Netz überführt werden.

[2] In [Laut02-1] wird dies auf S. 8 als *kanonische p/t-Netz-Repräsentation* bezeichnet.

Definition 31 Sei α eine aussagenlogische KNF-Formel, I eine Interpretation von α, $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha)$ die p/t-Netz-Implementierung von α und $p \in P$. Die *Markierung* M_I *von* $\mathcal{N}_\alpha$ bezogen auf die Interpretation I ist definiert als

$$M_I(p) := \begin{cases} 1 & \text{wenn } I(p) = \textit{wahr}\text{und} \\ 0 & \text{wenn } I(p) = \textit{falsch}. \end{cases}$$

[Laut02-1]

Mit diesen beiden Definitionen kann nun jede in KNF stehende aussagenlogische Formel als p/t-Netz implementiert werden. An einem Beispiel soll dies gezeigt werden.

Beispiel 5 (Vgl. S. 9 bei [Laut01]) Gegeben sei die folgende Formel

$$\alpha = \underbrace{(p \vee q)}_{(1)} \wedge \underbrace{(\neg p \vee q)}_{(2)} \wedge \underbrace{(\neg q \vee r)}_{(3)} \wedge \underbrace{(\neg q \vee \neg r \vee s)}_{(4)} \wedge \underbrace{(\neg q \vee \neg s)}_{(5)}.$$

Damit ergibt sich für die p/t-Netz-Implementierung $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha)$ von α:
$S_\alpha = \mathbb{A}(\alpha) = \{p, q, r, s\}$
$T_\alpha = \mathbb{C}(\alpha) = \{1, 2, 3, 4, 5\}$
$F_\alpha = \{(1,p), (1,q), (p,2), (2,q), (q,3), (3,r), (q,4), (r,4), (4,s), (q,5), (s,5)\}$
Ist I eine Interpretation von α mit $I(p) = falsch$, $I(q) = wahr$, $I(r) = wahr$ und $I(s) = falsch$, ergibt sich für die Markierung M_I unter der Interpretation I für $\mathcal{N}_\alpha$, dass $M_I^t = (0, 1, 1, 0)$. Das so bestimmte Netz ist in Abb. 4.1 dargestellt.

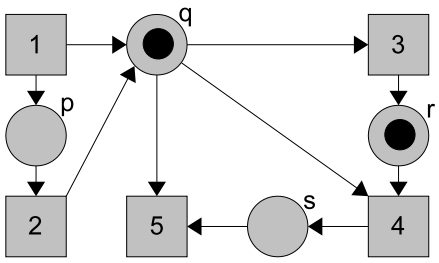


Abb. 4.1: Netz zu Beipiel 5

□

4.1.3 Transformation von p/t-Netze in die Aussagenlogik

Darauf aufbauend kann nun die aktionslogische Implementierung eines p/t-Netzes beschrieben werden. Sei dazu $\mathcal{N} = (P, T, F)$ ein p/t-Netz. Die Implementierung des Netzes $\mathcal{N}$ als aussagenlogische Formel $\alpha_\mathcal{N}$ geschieht durch Umkehrung der im vorigen Abschnitt in Def. 30 verwendeten Zuweisungen. Also ist die Menge der Atome bestimmt durch die Menge der Stellen mit

$$\mathbb{A}(\alpha_\mathcal{N}) = S.$$

Ebenso wird für jede Transition des Netzes eine Klausel konstruiert, sodass die Menge der Klauseln

$$\mathbb{C}(\alpha_\mathcal{N}) = T$$

ist. Die einzelnen Klauseln sind dabei wie folgt aufgebaut:

Sei $\tau_i \in \mathbb{C}(\alpha_\mathcal{N})$ die zur Transition $t_i \in T$ gehören Klausel und sei ${}^\bullet t_i = \{p_{i_1}, \ldots, p_{i_m}\}$ der Vorbereich von t_i und $t_i^\bullet = \{p_{i_{m+1}}, \ldots, p_{i_n}\}$ der Nachbereich von t_i. Die Klausel τ_i ist dann $\tau_i = \neg p_{i_1} \vee \cdots \vee \neg p_{i_m} \vee p_{i_{m+1}} \vee \cdots \vee, p_{i_n}$.

Die folgende Definition fasst das Ganze noch einmal formal zusammen.

Definition 32 Sei $\mathcal{N} = (P, T, F)$ ein p/t-Netz. Eine aussagenlogische Formel $\alpha_\mathcal{N}$ ist die *aussagenlogische Implementierung* von $\mathcal{N}$ gdw.

- $\mathbb{A}(\alpha_\mathcal{N}) = P$,
- $\mathbb{C}(\alpha_\mathcal{N}) = T$ und
- $\forall t_i \in T : \tau_i = (\neg p_{i_1} \vee \cdots \vee \neg p_{i_m} \vee p_{i_{m+1}} \vee \cdots \vee, p_{i_n}) \in \mathbb{C}(\alpha_\mathcal{N})$ wobei $\{p_{i_1}, \ldots, p_{i_m}\} = {}^\bullet t_i$ und $\{p_{i_{m+1}}, \ldots, p_{i_n}\} = t_i^\bullet$.

□

Umgekehrt zur Bestimmung der Markierung M aus der Interpretation I wie im vorigen Abschnitt muss nun die Interpretation I aus der Markierung abgeleitet werden. Da eine Stelle $p_i \in P$ des p/t-Netzes $\mathcal{N}$ i. A. mehr als eine Marke enthalten kann, das entsprechende Atom a_i der Formel aber nur einfach *wahr* sein kann, wird hier nur unterschieden, ob eine Stelle markiert ist oder nicht. Ist sie markiert, so ist das zugehörige Atom *wahr*, ansonsten *falsch*. Formal ausgedrückt ergibt sich so die folgende Definition.

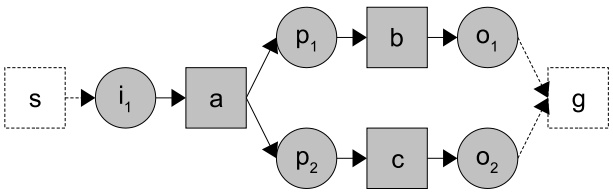
Definition 33 Sei $\mathcal{N} = (P, T, F)$ ein p/t-Netz, M eine Markierung von $\mathcal{N}$ und $\alpha_\mathcal{N}$ sei die aussagenlogische Implementierung von $\mathcal{N}$. Für $p \in P = \mathbb{A}(\alpha_N)$. ist die Interpretation I_M bezogen auf die Markierung M ist definiert als

$$I_M(p) := \begin{cases} \textit{wahr}, & \text{wenn } M(p) \text{ markiert ist,} \\ \textit{falsch}, & \text{wenn } M(p) \text{ unmarkiert ist.} \end{cases}$$

□

Auch hier soll das Ganze noch einmal an einem Beispiel gezeigt werden. Das umzuwandelnde p/t-Netz ist hierfür in Abb. 3.7 auf S. 19 abgebildete Workflow-Pattern für den UND-Split. Es ist hier aus Gründen der Übersichtlichkeit noch einmal mit abgebildet.

Beispiel 6 Zu dem folgenden Netz soll eine äquivalente aussagenlogische Formel in KNF konstruiert werden.



□

Das p/t-Netz ist also $\mathcal{N} = (P, T, F)$ mit

- $P = \{i_1, p_1, p_2, o_1, o_2\}$,

- $T = \{s, a, b, c, g\}$ und
- $F = \{(s, i_1)(i_1, a), (a, p_1), (a, p_2), (p_1, b), (p_2, c), (b, o_1), (c, o_2), (o_1, g), (o_2, g)\}$.

Die Markierung M von $\mathcal{N}$ ist $M(p) = 0 \forall p \in P$ Die zu implementierende aussagenlogische Formel sei $\alpha_{\mathcal{N}}$. Zunächst wird die Menge der Atome und Klauseln von α bestimmt.

- $\mathbb{A}(\alpha_{\mathcal{N}}) = P = \{i_1, p_1, p_2, o_1, o_2\}$
- $\mathbb{C}(\alpha_{\mathcal{N}}) = T = \{s, a, b, c, g\}$

Die einzelnen Klauseln s, a, b, c und g sind dann

- $s = (i_1)$,
- $a = (\neg i_1 \vee p_1 \vee p_2)$,
- $b = (\neg p_1 \vee o_1)$,
- $c = (\neg p_2 \vee o_2)$ und
- $g = (\neg o_1 \vee \neg o_2)$.

Daraus folgt für $\alpha_{\mathcal{N}}$:

$$\alpha_{\mathcal{N}} = (\underbrace{(i_1)}_{s}) \wedge \underbrace{(\neg i_1 \vee p_1 \vee p_2)}_{a} \wedge \underbrace{(\neg p_1 \vee o_1)}_{b} \wedge \underbrace{(\neg p_2 \vee o_2)}_{c}) \wedge \underbrace{(\neg o_1 \vee \neg o_2)}_{g})$$

Da das gesamte Netz leer ist, werden in der auf die Markierung M bezogenen Interpretation I_M alle Atome auf $falsch$ gesetzt:

$$I_M(p) = falsch \quad \forall p \in \mathbb{A}(\alpha_{\mathcal{N}})$$

4.2 Prädikatenlogik

Nachdem im vorigen Abschnitt die Implementierung eines p/t-Netzes zu einer gegebenen aussagenlogischen Formel und umgekehrt beschrieben wurde, erfolgt dies nun mit prädikatenlogischen Formeln und pr/t-Netzen. Die Prädikatenlogik ist eine Erweiterung der Aussagenlogik, bei der die Atome Eigenschaften von Objekten oder Beziehungen zwischen Objekten beschreiben. Das Prädikat $istVaterVon(Hans, Katrin)$ kann zum Beispiel ausdrücken, dass Hans der Vater von Katrin ist. Zunächst werden wieder die wichtigsten Begriffe eingeführt.

4.2.1 Definitionen und wichtige Begriffe der Prädikatenlogik

Definition 34 Das *Alphabet der Prädikatenlogik* besteht aus

- einer endlichen Menge von Funktionssymbolen $F = \{f_i^k | i, k \in \mathbb{N}\}$,
- einer endlichen Menge von Variablen $X = \{x_1, x_2, \dots\}$,
- einer Menge von Prädikatsymbolen $P = \{p_i^k | i, k \in \mathbb{N}\}$,
- den Konnektoren
  - $\wedge$ (logisches UND),
  - $\vee$ (logisches ODER) und
  - $\neg$ (logisches NICHT),
- den Quantoren
  - $\forall$ (Allquantor) und
  - $\exists$ (Existenzquantor) und
- den Hilfssymbolen '(', ')', und ','.

□

Mit diesem Alphabet kann zunächst die Menge der Terme $\mathcal{T}$ und anschließend die Menge der (wohlgeformten) Formeln $\mathcal{F}$ definiert werden.

Definition 35 Sei F eine Menge von Funktionssymbolen, und sei X eine Menge von Variablen. Die *Menge der Terme* $\mathcal{T}$ ist induktiv definiert als

- $\forall x \in X : x \in \mathcal{T}$,
- $(t_1, \dots, t_k \in \mathcal{T}) \wedge (f_i^k \in F) \Rightarrow f_i^k(t_1, \dots, t_k) \in \mathcal{T}$.

□

Ein besonderer Typ von Funktionen sind so genannte Konstanten, die keine Argumente besitzen und somit immer den gleichen Wert annehmen.

Definition 36 Terme des Typs $f_i^0()$ heißen *Konstanten* und werde ohne Klammern notiert als f_i^0 oder f_i.

□

Definition 37 Sei $P = \{p_i^k | i, k \in \mathbb{N}\}$ eine zählbare Menge von Prädikatsymbolen, $\mathcal{T}$ eine Menge von Termen und X eine Menge von Variablen. Die *Menge der (wohlgeformten) Formeln* $\mathcal{F}$ ist wie folgt induktiv definiert:

- $t_1, \ldots, t_k \in \mathcal{T}, p_i^k \in P \Rightarrow p_i^k(t_1, \ldots, t_k) \in \mathcal{F}$,
- $\alpha, \beta \in \mathcal{F} \Rightarrow (\alpha \wedge \beta) \in \mathcal{F}$ und $(\alpha \vee \beta) \in \mathcal{F}$
- $\alpha \in \mathcal{F} \Rightarrow \neg\alpha \in \mathcal{F}$ und
- $x \in X, \alpha \in \mathcal{F} \Rightarrow (\exists x\alpha) \in \mathcal{F}$ und $(\forall x\alpha) \in \mathcal{F}$.

Formeln der Form $P_i^k(t_1, \ldots, t_k)$ heißen *Atome* oder *atomare Formeln*.

□

Die Begriffe *Literal*, *Klausel* sowie die Mengen $\mathbb{A}$ und $\mathbb{C}$ sind in der Prädikatenlogik genauso definiert wie in der Aussagenlogik (vgl. Def. 26 und 29).

Nullstellige Prädikate der Form $P_i^0()$ können auch als P_i^0 oder P_i notiert werden und haben so die Form eines Atoms der Aussagenlogik. Einem solchen Prädikat wird von der noch zu definierenden Interpretation direkt ein Wert aus der Menge $\{wahr, falsch\}$ zugewiesen.

Definition 38 Eine Variable x einer Formel α heißt *gebunden*, wenn sie in einer Subformel β von α der Form $\exists x\beta$ oder $\forall x\beta$ vorkommt. Ansonsten heißt x *frei*.

□

Jetzt kann definiert werden, was eine Interpretation ist.

Definition 39 Eine *Interpretation* ist ein Paar $I = (U_I, A_I)$, wobei

- U_I eine beliebige nichtleere Menge, genannt *Domain* oder *Universum* und
- A_I eine Abbildung ist, die
  - jedem k-ären Prädikatsymbol ein k-äres Prädikat aus U_I,
  - jedem k-ären Funktionssymbol eine k-äre Funktion aus U_I und
  - jeder Variablen ein Element der Domain zuordnet.

Wenn α eine Formel und $I = (U_I, A_I)$ eine Interpretation ist, dann heißt I eine Interpretation von α, gdw. A_I für jedes Prädikat- und Funktionssymbol und für jede freie Variable in α definiert ist.

□

Definition 40 (**Semantik der Prädikatenlogik**) Seien α und β prädikatenlogische Formeln und $I = (U_I, A_I)$ eine Interpretation von α. Für die Terme t, die aus den Symbolen von α gebildet werden können, ist der Wert von $I(t)$ gegeben durch

- $I(x) = x^I$ wenn $x \in \mathbb{A}(\alpha)$ und

- $I(f(t_1, \ldots, t_k)) = f^I(I(t_1), \ldots, I(t_k))$, wenn $t_1, \ldots, t_k$ Terme sind und f ein k-äres Funktionssymbol ist.

Der Wert I einer Formel wird bestimmt mit

- $I(p(t_1, \ldots, t_k)) = \begin{cases} wahr & \text{wenn } (I(t_1), \ldots, I(t_k)) \in p^I \\ falsch & \text{sonst,} \end{cases}$
- $I(\alpha \wedge \beta) = \begin{cases} wahr & \text{wenn } (I(\alpha) = wahr) \wedge (I(\beta) = wahr) \\ falsch & \text{sonst,} \end{cases}$
- $I(\alpha \vee \beta) = \begin{cases} wahr & \text{wenn } (I(\alpha) = wahr) \vee (I(\beta) = wahr) \\ falsch & \text{sonst,} \end{cases}$
- $I(\neg\alpha) = \begin{cases} wahr & \text{wenn } I(\alpha) = falsch \\ falsch & \text{sonst,} \end{cases}$
- $I(\forall x\alpha) = \begin{cases} wahr & \text{wenn } \forall d \in U_I : I_{[x/d]}(\alpha) = wahr \\ falsch & \text{sonst,} \end{cases}$
- $I(\exists x\alpha) = \begin{cases} wahr & \text{wenn } \exists d \in U_I : I_{[x/d]}(\alpha) = wahr \\ falsch & \text{sonst,} \end{cases}$

wobei

$$f_{[x/d]}(y) = \begin{cases} f(y) & \text{wenn } y \neq x \text{ und} \\ d & \text{sonst} \end{cases}$$

ist.

□

4.2.2 Transformation der Prädikatenlogik in pr/t-Netze

Nach Einführung der wesentlichen Begriffe folgt nun auch für die Prädikatenlogik ein Weg, Formeln als pr/t-Netz darzustellen. Sei also α eine prädikatenlogische Formel erster Ordnung in KNF. $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha, L_\alpha)$ ist das zu konstruierende pr/t-Netz, das die Formel $\alpha_\mathcal{N}$ implementiert. Außerdem bezeichne $\mathbb{A}(\alpha_\mathcal{N})$ wieder die Menge der Atome und $\mathbb{C}(\alpha_\mathcal{N})$ die Menge der Klauseln, die in $\alpha_\mathcal{N}$ vorkommen.

Mit dieser Definition für Atome und Klauseln in der Prädikatenlogik kann nun das Netz genauso erstellt werden, wie wir es im Abschnitt 4.1 getan haben. Dabei sei $P_i^k()$ eine Abkürzung für das Atom $P_i^k(te_1, \ldots, te_k)$. So kann bereits festgehalten werden, dass

- $S_\alpha = \mathbb{A}(\alpha)$ und
- $T_\alpha = \mathbb{C}(\alpha)$.

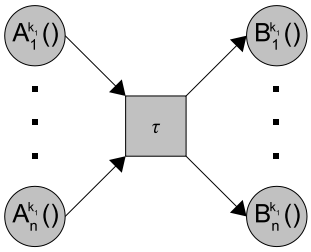


Abb. 4.2: Klausel τ im pr/t-Netz

Ist $\tau = (\neg A_1^{k_1}() \vee \cdots \vee \neg A_m^{k_m}() \vee (B_1^{k_1}() \vee \cdots \vee B_n^{k_n}())$ eine Klausel in α, so wird, wie in Abb. 4.2 dargestellt, eine Transition τ erstellt mit dem Vorbereich ${}^\bullet\tau = \{A_1^{k_1}(), \ldots, A_m^{k_m}()\}$ und dem Nachbereich $\tau^\bullet = \{B_1^{k_1}(), \ldots, B_n^{k_n}()\}$.

Die Terme selbst werden nun in die Kantenlabels übertragen. Hat man nach obiger Vorgehensweise eine Stelle $P_i^k(te_1, \ldots, te_k)$ zu genau diesem Prädikat erstellt, wird nun wie in Abb. 4.3 gezeigt das Label an allen Eingangs- und Ausgangskanten der Stelle erzeugt als das Tupel $< te_1, \ldots, te_k >$.

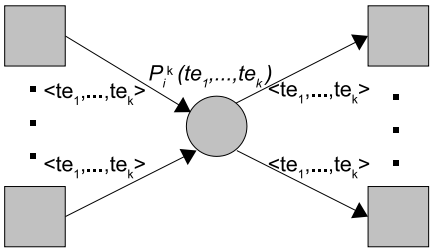


Abb. 4.3: Terme eines Prädikats $P_i^k(te_1, \ldots, te_k)$ im pr/t-Netz

Dies soll noch einmal zusammengefasst und in der folgenden Definition festgehalten werden.

Definition 41 Sei α eine prädikatenlogische Formel in KNF ohne Quantoren. Das pr/t-Netz $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha, L_\alpha)$ ist die *pr/t-Netz-Implementierung* von α gdw.

- $P_\alpha = \mathbb{A}(\alpha)$,
- $T_\alpha = \mathbb{C}(\alpha)$,
- $\forall\ \tau = (\neg A_1^{k_1}() \vee \cdots \vee \neg A_m^{k_m}() \vee (B_1^{k_1}() \vee \cdots \vee B_n^{k_n}()) \in \mathbb{C}(\alpha)$ mit $\{A_1^{k_1}(), \ldots, A_m^{k_m}(), (B_1^{k_1}(), \ldots, B_n^{k_n}()\} \subseteq \mathbb{A}(\alpha)$:
  F_α bestimmt wird durch ${}^\bullet\tau = \{(A_1^{k_1}(), \ldots, A_m^{k_m}()\}$ und $\tau^\bullet = \{(B_1^{k_1}(), \ldots, B_n^{k_n}()\}$ und
- $\forall P_i^k(te_1, \ldots, te_k) \in \mathbb{A}(\alpha) = P_\alpha$:
  $L_\alpha(x, y) = \{< te_1, \ldots, te_k >\}$ wobei $(x, y) \in F_\alpha$ und $x = P_i^k(te_1, \ldots, te_k)$ oder $y = P_i^k(te_1, \ldots, te_k)$ ist.

□

Nach dieser Definition einer Netz-Implementierung wird auch hier wieder eine Definition benötigt, die beschreibt, wie die Markierung des Netzes zu erzeugen ist.

Definition 42 Sei α eine prädikatenlogische Formel in KNF ohne Quantoren, $I = (U_I, A_I)$ eine Interpretation von α, $\mathcal{N}_\alpha = (P_\alpha, T_\alpha, F_\alpha, L, \alpha)$ die pr/t-Netz-Implementierung von α. Für jede Stelle $P_i^k(te_1, \ldots, te_k) \in P = \mathbb{A}(\alpha)$ ist die *Markierung* $M_I(P_i^k(te_1, \ldots, te_k))$ *von* $\mathcal{N}_\alpha$ bezogen auf die Interpretation I definiert als

$$M_I(p) := \begin{cases} \{< te_1, \ldots, te_k >\} & \text{wenn } I(p) = \textit{wahr} \text{ und} \\ \emptyset & \text{wenn } I(p) = \textit{falsch}. \end{cases}$$

□

4.2.3 Transformation von pr/t-Netzen in die Prädikatenlogik

Die umgekehrte Transformation, also die Implementierung eines gegebenen pr/t-Netzes $\mathcal{N} = (P, T, F, L')$ als prädikatenlogische Formel $\alpha_\mathcal{N}$, ist etwas einfacher. Auch diese ähnelt der Vorgehensweise aus dem vorigen Kapitel. Allerdings muss die Menge der Kantenlabels eingeschränkt werden auf $L' : F \to Y_n$. Jetzt wird zunächst wieder für jede Stelle des Netzes ein Atom erstellt:

$$\mathbb{A}(\alpha_\mathcal{N}) = P$$

Dabei werden die einzelnen Atome, und somit die Prädikate, wie folgt erstellt. Die Kantenlabels korrespondieren wie im vorigen Abschnitt mit den Termen der Prädikate (und somit der Stellen), die den Kanten auf einer Seite anhängen. Wenn z. B. zur in Abb. 4.4 dargestellten Stelle p ein Prädikat erstellt wurde, so ist dieses gleich $p(te_1, \ldots, te_k)$.[3]

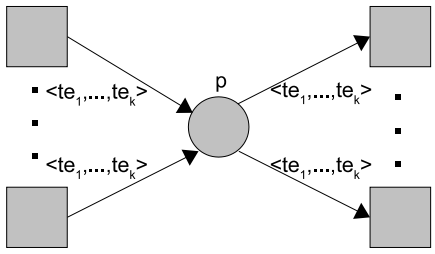


Abb. 4.4: Eine Beispiel-Transition τ_i

Die Klauselmenge ist wiederum gleich der Menge der Transitionen:

$$\mathbb{C}(\alpha_\mathcal{N}) = T$$

Die Übertragung der Kanten erfolgt wie bei der Implementierung einer aussagenlogischen Formel als p/t-Netz. Für jede Transition $t_i \in T$ sei die zugehörige Klausel $\tau_i \in \mathbb{C}(\alpha_\mathcal{N})$ wie folgt aufgebaut: Alle Stellen, die im Vorbereich der Transition t_i sind, werden negiert und durch die $\vee$-Verknüpfung aneinandergereiht in τ_i eingefügt. Nach einer weiteren $\vee$-Verknüpfung werden dann alle Stellen des Nachbereichs von t_i durch $\vee$ verbunden an τ_i angehängt.

[3] Die i-ten Tupelglieder müssen dabei nicht immer an jeder Kante den gleichen Namen haben. Ist dies der Fall, kann ein beliebiges dieser i-ten Tupelglieder ausgewählt werden und als Name des i-ten Terms des Prädikats dienen.

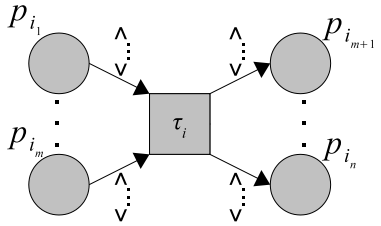


Abb. 4.5: Eine Beispiel-Transition τ_i

Ist z. B. wie in Abb. 4.5 gezeigt ${}^\bullet t_i = \{p_{i_1}, \ldots, p_{i_m}\}$ der Vorbereich von t_i und $t_i^\bullet = \{p_{i_{m+1}}, \ldots, p_{i_n}\}$ der Nachbereich von t_i, ergibt sich für die Klausel τ_i:

$$\tau_i = \neg p'_{i_1}() \vee \cdots \vee \neg p_{i_m}() \vee p_{i_{m+1}}() \vee \cdots \vee, p_{i_n}().$$

Die Argumente der einzelnen Prädikate werden hierbei direkt aus den Kantenlabels erstellt. Wenn z. B. $L(p_{i_k}, t_i) =< x_1, \ldots, x_l >$ $(L(t_i, p_{i_k}) =< x_1, \ldots, x_l >)$ ist, so ist das Prädikat in diesem Fall $p_{i_k}(x_1, \ldots, x_l)$ (ebenso).

Dies alles soll noch einmal formal in der folgenden Definition festgehalten werden.

Definition 43 Sei $\mathcal{N} = (P, T, F, L')$ ein pr/t-Netz. Die prädikatenlogische Formel $\alpha_\mathcal{N}$ ist die *prädikatenlogische Implementierung* von $\mathcal{N}$ gdw.

- $\mathbb{A}(\alpha_\mathcal{N}) = S$,
- $\mathbb{C}(\alpha_\mathcal{N}) = T$,
- $\forall t_i \in T : \tau_i = \neg p_{i_1}(Arg_{i_1}) \vee \cdots \vee \neg p_{i_m}(Arg_{i_m}) \vee p_{i_{m+1}}(Arg_{i_{m+1}}) \vee \cdots \vee, p_{i_n}(Arg_{i_n})$ mit $\{p_{i_1}, \ldots, p_{i_m}\} = {}^\bullet t_i, \{p_{i_{m+1}}, \ldots, p_{i_n}\} = t_i^\bullet$ und
  $Arg_{i_j} = x_1, \ldots, x_l$ wenn $< x_1, \ldots, x_l >= \begin{cases} L'(p_{i_j}, t_i) & \text{wenn } 1 \leq j \leq m \text{ und} \\ L'(t_i, p_{i_j}) & \text{sonst.} \end{cases}$

□

Zuletzt muss noch eine Übertragung der Markierung gefunden werden. Dazu wird jede Marke einer jeden Stelle als Fakt in die Formel eingefügt. Ist z. B. $M_\mathcal{N}(p) = \{< x_{1_1}, \ldots, x_{k_1} >, \ldots, < x_{1_m}, \ldots, x_{k_m} >\}$ die Markierung des Netzes $\mathcal{N}$ an der Stelle p, so werden an die bis hierher implementierte Formel $\alpha_\mathcal{N}$ die folgenden Fakten nach einer UND-Verknüpfung angehängt:

$$p(x_{1_1}, \ldots, x_{1_k}) \wedge \cdots \wedge p(x_{1_m}, \ldots, x_{m_k})$$

Definition 44 Sei $\mathcal{N} = (P, T, F, L')$ ein pr/t-Netz, $P = \{p_1, \ldots, p_n\}$ und $M = (M(p_1), \ldots, M(p_n))$ mit $M(p_i) = \{< x_{1_{1_i}}, \ldots, x_{k_{1_i}} >, \ldots, < x_{1_{m_i}}, \ldots, x_{k_{m_i}} >\}$ sei eine Markierung von $\mathcal{N}$. Die prädikatenlogische Formal $\alpha_{(\mathcal{N},M)}$ implementiert das Netz $\mathcal{N}$ samt seiner Markierung M gdw.

- $\alpha_\mathcal{N}$ eine prädikatenlogische Implementierung von $\mathcal{N}$ ist und
- $\alpha_{(\mathcal{N},M)} = \alpha_\mathcal{N} \wedge p_1(x_{1_{1_1}}, \ldots, x_{k_{1_1}}) \wedge \cdots \wedge p_1(x_{1_{m_1}}, \ldots, x_{k_{m_1}}) \wedge \cdots \wedge p_n(x_{1_{1_n}}, \ldots, x_{k_{1_n}}) \wedge \cdots \wedge p_n(x_{1_{m_n}}, \ldots, x_{k_{m_n}})$.

□

Nachdem die Implementierung eines pr/t-Netzes als Formel nun allgemein formuliert wurde, soll das Ganze abschließend an dem folgenden aus [Laut02-2] entnommenen Beispiel exemplarisch durchgeführt werden.

Beispiel 7 Das erste Beispiel kommt aus der sogenannten *Blocks World*, in der ein Roboterarm die Aufgabe hat, Blöcke aufeinander zu stapeln. Die Blöcke liegen zu Beginn in bis zu drei Stapeln auf einem Tisch. Der Arm kann immer nur einen Block von einem Stapel aufnehmen und oben auf einen anderen Stapel setzen. Abb. 4.6 zeigt einen möglichen Zustand der *Blocks World*.

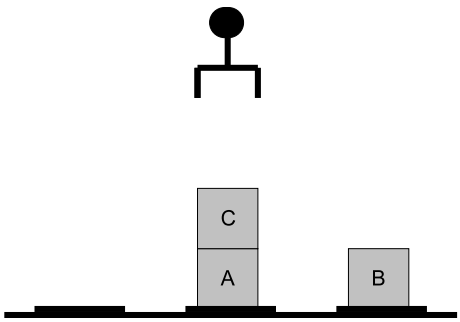


Abb. 4.6: Ein möglicher Zustand der *Blocks World*

Für diese *Blocks World* soll zunächst eine Darstellung als pr/t-Netz $\mathcal{N} = (P, T, F, L')$ gefunden werden. Dafür werden die Zustände bestimmt, die die Objekte der *Blocks World* einnehmen können. Die Objekte sind in diesem Fall der Roboterarm und die einzelnen Blöcke. Für den Roboterarm sollen zwei Stellen in das Netz eingefügt werden. Eine gibt an, ob der Greifhand des Roboters leer ist (HL als Abkürzung für „Hand ist leer“), mit der anderen wird festgehalten, welcher Block gerade vom Arm gehalten wird (H für „halten“). Für die Blöcke gibt es eine Stelle, die angibt welche Blöcke direkt auf dem Tisch liegen (AT für „auf dem Tisch“), eine, die angibt welche Blöcke oben auf einem Stapel liegen (O für „oben“) und eine, die angibt welche Blöcke direkt aufeinander liegen (A für „auf“). Die Menge der Stellen ist somit

$$S = \{HL, H, AT, A, O\}.$$

Die Transitionen geben die möglichen Aktionen an, die in der *Blocks World* durchgeführt werden können. Der einzig Agierende der *Blocks World* ist der Roboterarm. Er kann Blöcke aufnehmen und ablegen, was in den Transitionen AN und AL dargestellt wird. Es macht aber für die zuvor beschriebenen Zustände einen Unterschied, ob ein aufgenommener Block zuvor auf dem Tisch oder auf einem anderen Block lag. Im ersten Fall liegt der Block anschließend nicht mehr auf dem Tisch, sodass das entsprechende Tupel von der Stelle AT genommen werden muss. Im zweiten Fall liegt der Block nicht mehr auf einem anderen Block, sodass das Tupel von Stelle A genommen wird und in die Stelle O eingefügt wird, sodass der andere Block jetzt oben auf einem Stapel liegt. Um dies unterscheiden zu können, wird für den zweiten Fall eine Transition VS (für „vom Stapel nehmen“) erstellt. Das Gleiche gilt für das Ablegen eines Blocks und führt so zu einer Transition S (für „stapeln“). Damit ist die Menge der Transitionen

$$T = \{AN, AL, VS, S\}.$$

Es fehlen noch die Kanten des Netzes. Die Kanten sollen am Beispiel der Transition AN (aufnehmen) etwas genauer erklärt werden und anschließend auch für die anderen Transitionen gegeben werden. Soll ein Block X vom Arm aufgenommen werden, muss vorher die Hand leer sein (d. h. das leere Tupel <> muss auf HL liegen), der Block muss oben auf einem Stapel liegen (d. h. das Tupel $< X >$ muss auf O liegen) und der Block muss direkt auf dem Tisch liegen (d. h. das Tupel $< X >$ muss auch auf AT liegen). Ist der Block aufgenommen, so ist die Hand nicht mehr leer und der Block liegt auch nicht mehr oben auf einem Stapel bzw. auf dem Tisch. Die Tupel <>, $< X >$ und $< X >$ werden also respektive von HL, O und AT entfernt. Dargestellt wird dies jeweils durch eine Kante von jeder der Stellen, jeweils mit den beschriebenen Tupeln als Kantenmarkierung zur Transition AN.
Zudem wird nach dem Aufnehmen X in der Hand gehalten. Also muss das Tupel $< X >$ auf die Stelle H gelegt werden. Dargestellt wird dies durch eine Kante von Transition AN zur Stelle H hin mit der Kantenmarkierung $< X >$. Das gerade Beschriebene wird in Abb. 4.7 abgebildet.

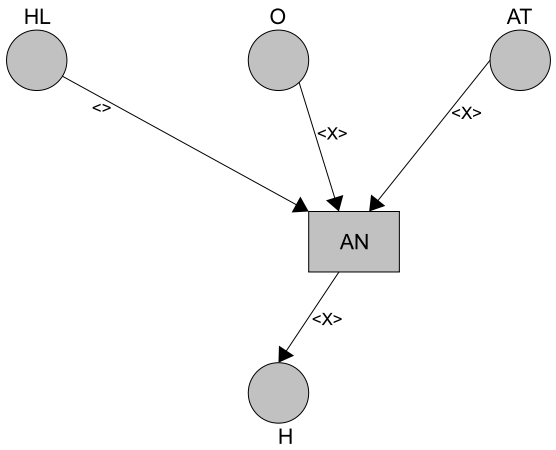


Abb. 4.7: Die Transition AN im pr/t-Netz

Ebenso werden die restlichen Kanten in das Netz eingefügt, sodass das Netz in Abb. 4.8 entsteht.

Um den in Abb. 4.6 gegebenen Zustand im gerade erstellten Netz zu erhalten, muss das Netz noch markiert werden.

Die einzelnen Zustände werden dazu als Tupel auf die Stellen gelegt. Die leere Hand führt zu einem leeren Tupel auf HL: $M_0(HL) =<>$. Die einzigen Blöcke, die aufeinander liegen, sind C und A. Das ergibt das Tupel $< C, A >$ auf Stelle A: $M_0(A) = \{< C, A >\}$. Direkt auf dem Tisch liegen die Blöcke A und B. Daraus ergibt sich $M_0(AT) = \{< A >, < B >\}$. Blöcke, die ganz oben auf einem Stapel liegen, sind B und C, so ist $M_0(O) = \{< B >, < C >\}$. Die letzte zu markierende Stelle ist H. Da aber kein Block vom Roboterarm

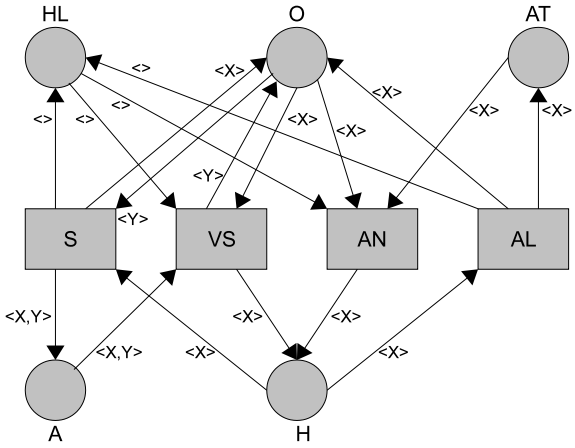


Abb. 4.8: Die *Blocks World* als pr/t-Netz

gehalten wird, bleibt H zunächst unmarkiert: $M_0(H) = \emptyset$. Die Anfangsmarkierung ist also

$$M_0 = \begin{pmatrix} <> \\ < C, A > \\ < A >, < B > \\ < B >, < C > \\ \emptyset \end{pmatrix}.$$

Das so markierte Netz steht in Abb. 4.9.

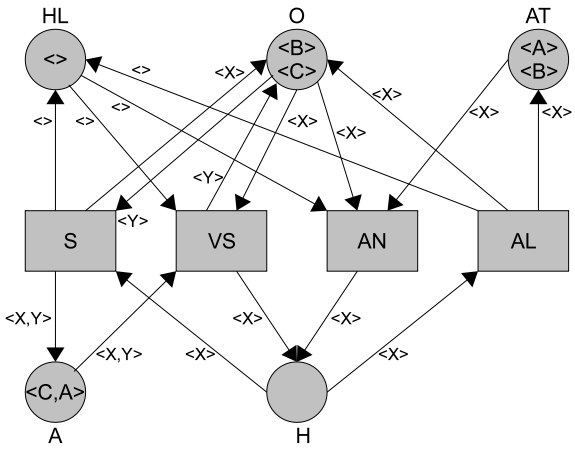


Abb. 4.9: Das markierte *Blocks World*-Netz

Jetzt kann mit der Übertragung des Netzes $\mathcal{N}$ in eine prädikatenlogische Formel $\alpha_{\mathcal{N}}$ begonnen werden. Zuerst wird die Menge der Atome und die Menge der Klauseln von $\alpha_{\mathcal{N}}$ bestimmt:

$$\mathbb{A}(\alpha_{\mathcal{N}}) = S = \{HL, O, AT, A, H\}$$

$$\mathbb{C}(\alpha_{\mathcal{N}}) = T = \{AN, AL, VS, S\}$$

Nun müssen die einzelnen Klauseln bestimmt werden. Exemplarisch wird dies für AN durchgeführt. Der Vorbereich von AN ist ${}^\bullet AN = \{HL, O, AT\}$, der Nachbereich ist

$AN^\bullet = \{H\}$. Durch Negation der Elemente des Vorbereichs und anschließendes ODERn dieser Elemente mit dem Element des Nachbereichs erhält man als Klausel AN

$$AN = (\neg HL() \vee \neg O(X) \vee \neg AT(X) \vee H(X)).$$

Für die Klausel AL ergibt sich

$$AL = (\neg H(X) \vee HL() \vee O(X) \vee AT(X)).$$

Die Klausel VS ist

$$VS = (\neg HL() \vee \neg O(X) \vee \neg A(X,Y) \vee O(Y) \vee H(X)).$$

Und schließlich ist die Klausel S

$$S = (\neg O(Y) \vee \neg H(X) \vee HL() \vee O(X) \vee A(X,Y)).$$

□

4.3 Transaktionslogik

Die in 4.3.1 vorgestellte Transaktionslogik ($\mathcal{TR}$) baut auf der zuvor beschriebenen Prädikatenlogik auf. Sie eignet sich zur Beschreibung und auch zur Programmierung von Zustandsänderungen in der logischen Programmierung, der künstlichen Intelligenz und von Datenbanken. Dazu ist sie zusätzlich zu den Operatoren der Prädikatenlogik mit Operatoren ausgestattet, mit denen die Ausführungsreihenfolge einzelner Operationen festgelegt werden kann. Mit der $\mathcal{TR}$ werden deshalb im Gegensatz zur Aussagenlogik und Prädikatenlogik nicht nur Zustände beschrieben, sondern auch die Dynamik eines solchen Systems, d. h. die Änderung dieser Zustände. Dies wird im Folgenden genauer definiert.

4.3.1 Definitionen und wichtige Begriffe der Transaktionslogik

Als Erweiterung der Prädikatenlogik besteht das Alphabet der $\mathcal{TR}$ aus dem Alphabet der Prädikatenlogik und einigen zusätzlichen Zeichen.

Definition 45 Das *Alphabet der $\mathcal{TR}$* besteht aus dem Alphabet der Prädikatenlogik nach Def. 34 und zusätzlich

- den Konnektoren
  - $\otimes$ (serielles UND),
  - $\oplus$ (serielles ODER) und
  - $|$ (KOINZIDENZ),
- dem einstelligen Operator
  - $\odot$ (ATOMIZITÄT) und
- den Hilfssymbolen '{', '}', '$\langle$' und '$\rangle$'.

□

Für die Begriffe der *Menge der Terme* und *Konstanten* gelten respektive die Definitionen 35 und 36 aus der Prädikatenlogik. Auch in der $\mathcal{TR}$ wird mit Hilfe der Terme die Menge der $\mathcal{TR}$-Formeln $\mathcal{F}_{\mathcal{TR}}$ definiert.

Definition 46 Sei $P = \{p_i^k | i, k \in \mathbb{N}\}$ eine zählbare Menge von Prädikatsymbolen, $\mathcal{T}$ eine Menge von Termen und X eine Menge von Variablen. Die *Menge der $\mathcal{TR}$-Formeln* $\mathcal{F}_{\mathcal{TR}}$ ist wie folgt induktiv definiert:

- $t_1, \ldots, t_k \in \mathcal{T}, p_i^k \in P \Rightarrow p_i^k(t_1, \ldots, t_k) \in \mathcal{F}_{\mathcal{TR}}$,
- $\alpha, \beta \in \mathcal{F}_{\mathcal{TR}} \Rightarrow (\alpha \wedge \beta) \in \mathcal{F}_{\mathcal{TR}}$, $(\alpha \vee \beta) \in \mathcal{F}_{\mathcal{TR}}$, $(\alpha \otimes \beta) \in \mathcal{F}_{\mathcal{TR}}$, $(\alpha \oplus \beta) \in \mathcal{F}_{\mathcal{TR}}$ und $(\alpha | \beta) \in \mathcal{F}_{\mathcal{TR}}$,
- $\alpha \in \mathcal{F}_{\mathcal{TR}} \Rightarrow \neg\alpha \in \mathcal{F}_{\mathcal{TR}}$,
- $\alpha \in \mathcal{F}_{\mathcal{TR}} \Rightarrow \odot\alpha \in \mathcal{F}_{\mathcal{TR}}$ und
- $x \in X, \alpha \in \mathcal{F}_{\mathcal{TR}} \Rightarrow (\exists x \alpha) \in \mathcal{F}_{\mathcal{TR}}$ und $(\forall x \alpha) \in \mathcal{F}_{\mathcal{TR}}$.

Formeln der Form $P_i^k(t_1, \ldots, t_k)$ heißen auch hier wieder *Atome* oder *atomare Formeln*. Zudem werden Formeln im Folgenden auch synonym als *Transaktionen* bezeichnet.

□

Wie bereits erwähnt, ist die $\mathcal{TR}$ eine Erweiterung der Prädikatenlogik, und so kann auch die auf S. 34 in Def. 40 gegebene Semantik hier übernommen werden. Einzig die neuen Symbole $\otimes$, $\oplus$, $|$ und $\odot$ müssen in ihrer Bedeutung noch definiert werden.

Definition 47 (**Semantik der $\mathcal{TR}$**) Wenn α und β $\mathcal{TR}$-Formeln sind, gilt für die $\mathcal{TR}$ zusätlich zu Def. 40:

- $I(\alpha \otimes \beta) = \begin{cases} wahr & \text{wenn zuerst } I(\alpha) = wahr \\ & \text{und dann } I(\beta) = wahr, \\ falsch & \text{sonst,} \end{cases}$
- $I(\alpha \oplus \beta) = \begin{cases} wahr & \text{wenn zuerst } I(\alpha) = wahr \\ & \text{oder später } I(\beta) = wahr, \\ falsch & \text{sonst,} \end{cases}$
- $I(\alpha|\beta) = \begin{cases} wahr & \text{wenn } I(\alpha) = wahr \\ & \text{und zeitgleich } I(\beta) = wahr, \\ falsch & \text{sonst und} \end{cases}$
- $I(\odot\alpha) = I(\alpha)$.

□

Regeln haben in der $\mathcal{TR}$ die Form

$$p(x_1, \ldots, x_n) \leftarrow \alpha,$$

wobei $p(x_1, \ldots x_n)$ ein Atom und α eine beliebige $\mathcal{TR}$-Formel ist. Wie in der klassischen Logik ist dies auch hier eine Abkürzung für $p(x_1, \ldots, x_n) \vee \neg\alpha$. Zusätzlich gibt es in der $\mathcal{TR}$ aber noch eine *prozedurale Interpretation*, die (informell ausgedrückt) besagt, dass es genügt, α auszuführen, um $p(x_1, \ldots, x_n)$ auszuführen. p ist dann der Name der Prozedur, $(x_1, \ldots, x_n)$ sind die zu übergebenden Parameter und α ist der Rumpf der Prozedur. Dies ermöglicht es, komplexe Formeln übersichtlicher zu notieren. Ist zum Beispiel

$$p(X, Y, Z) \leftarrow a(X) \wedge b(X, Y) \vee c(Z)$$

eine $\mathcal{TR}$-Formel, so kann diese in die folgende äquivalente Darstellung überführt werden:

$$p(X, Y, Z) \leftarrow ab(X, Y) \vee c(Z)$$

$$ab(X, Y) \leftarrow a(X) \wedge b(X, Y)$$

Soll also $p(X, Y, Z)$ berechnet werden, wird stattdessen der Wert von $ab(X, Y) \vee c(Z)$ bestimmt. Um wiederum den Wert von $ab(X, Y)$ zu erhalten, muss zuvor $a(X) \wedge b(X, Y)$ berechnet werden. Aus diesen Regeln wird in der $\mathcal{TR}$ die Transaktionsbasis P zusammengesetzt.

Definition 48 Die *Transaktionsbasis* P ist in der $\mathcal{TR}$ eine Konjunktion von Regeln.

□

Zusätzlich zur Menge der Formeln, die einfache Transaktionen zu komplexen zusammenfügen, gibt es in der $\mathcal{TR}$ eine *Transitionsbasis* $\mathcal{B}$, die aus elementaren Zustands-Transitionen (im weiteren Verlauf kurz als „elementare Transitionen“ bezeichnet) besteht und in der die eigentlichen Operationen auf der Datenbasis definiert werden. Dieser Begriff der Datenbasis wird jetzt definiert, um mit Hilfe dieses Begriffs die Transitionsbasis beschreiben zu können.

Definition 49 Eine *Datenbasis* D ist eine Menge von prädikatenlogischen Formeln erster Ordnung.

□

Die Datenbasis enthält im weiteren Verlauf dieser Arbeit allerdings nur atomare Formeln. Ist ein Atom in der Datenbasis vorhanden bedeutet dies, dass das Atom zum aktuellen Zeitpunkt wahr ist. In der Transitionsbasis wird durch die elementaren Transitionen festgehalten, wie die Datenbasis manipuliert werden kann, also wie Daten hinzugefügt bzw. entfernt werden können.

Definition 50 Sind D_1 und D_2 Datenbasen und u eine atomare Formel, so ist $\langle D_1, D_2 \rangle u$ eine *elementare Transition.*

□

$\langle D_1, D_2 \rangle u$ bedeutet, dass u die aktuelle Datenbasis D_1 in die Datenbasis D_2 überführt. Zum Beispiel wird durch $\langle D_1, D_1 \cup \{b\} \rangle$*ins:b* beschrieben, dass *ins:b* zur aktuellen Datenbasis D_1 das Atom b hinzufügt. Analog dazu beschreibt $\langle D_1 \cup \{b\}, D_1 \rangle$*del:b* bzw. $\langle D_1, D_1 - \{b\} \rangle$*del:b*, dass das Atom b aus der Datenbasis durch *del:b* entfernt wird. Zu beachten ist hierbei, dass weder „ins:“ noch „del:“ in der $\mathcal{TR}$ eine besondere Bedeutung haben. Es hätte auch jedes andere Wort anstatt der beiden Begriffe verwendet werden können. Die Semantik ist allein durch die Verwendung in einer elementaren Transition gegeben. Im weiteren Verlauf dieser Arbeit soll allerdings die Konvention gelten, dass *ins:x* das Hinzufügen eines Datums b zur aktuellen Datenbasis und *del:b* das Entfernen eines Datums x aus der aktuellen Datenbasis beschreiben. So soll auch das Vorhandensein eines Datums b in einer Datenbasis D immer durch das Atom *get:x* beschrieben werden mit $\langle D \cup \{b\}, D \cup \{b\} \rangle$*get:b.*

Außerdem sei noch darauf hingewiesen, dass u als atomare Formel auch Teil einer komplexen $\mathcal{TR}$-Formel sein kann wie z. B. in einer Regel $p(x_1, \ldots, x_n) \leftarrow \alpha \boxdot u \boxdot \beta$, wobei $p(x_1, \ldots x_n)$ ein Atom ist, $\boxdot \in \{\wedge, \vee, \otimes, \oplus, |\}$ und α und β beliebige $\mathcal{TR}$-Formeln sind. So können einfache Datenbank-Operationen zu komplexen zusammengefügt werden.

Eine besondere elementare Transition ist ϵ, die in jeder Transitionsbasis implizit enthalten sein soll. Sie ist immer ausführbar und wird gebraucht, um eine evtl. benötigte leere Aktion auszudrücken.

Definition 51 Sei $\mathcal{B}$ eine beliebige Transitionsbasis, P eine beliebige Transaktionsbasis und D sei eine beliebige Datenbasis. Es gilt

$$\forall \mathcal{B} : (\langle D, D \rangle \epsilon) \in \mathcal{B}$$

und

$$\forall P : (\leftarrow \epsilon) \in P.$$

□

Ein aus [BoKi94] entnommene Beispiel soll helfen, das bisher Beschriebene besser zu verstehen.

Beispiel 8 Es soll dargestellt werden, was alles getan werden muss, um einen Betrag Amt von einem Konto *a1* auf ein Konto $Acct$ zu überweisen. Dazu muss zunächst der Betrag Amt vom Konto *a1* abgezogen und später auf dem Zielkonto hinzugefügt werden. Die folgende Regel beschreibt dies transaktionslogisch:

$$payTo(Amt, Acct) \leftarrow withdraw(Amt, a1) \otimes deposit(Amt, Acct)$$

Wie dieses Abziehen und Hinzufügen genauer aussieht, ist in den beiden folgenden Regeln dargestellt:

$$\begin{aligned} withdraw(Amt, Acct) \quad &\leftarrow \quad get{:}balance(Acct, B) \otimes B \geq Amt \otimes \\ &\qquad del{:}balance(Acct, B) \otimes \\ &\qquad ins{:}balance(Acct, B - Amt) \end{aligned}$$

$$\begin{aligned} deposit(Amt, Acct) \quad &\leftarrow \quad get{:}balance(Acct, B) \otimes \\ &\qquad del{:}balance(Acct, B) \otimes \\ &\qquad ins{:}balance(Acct, B + Amt) \end{aligned}$$

Um also einen Betrag von einem Konto abzuziehen, wird zunächst der aktuelle Kontostand benötigt. Anschließend wird überprüft, ob genügend Geld auf dem Konto vorhanden ist. Ist dies der Fall, wird der alte Kontostand gelöscht und der neue Kontostand, der sich aus dem alten abzüglich der zu überweisenden Summe ergibt, wird in die Datenbank eingefügt. Soll ein Betrag zu einem Konto hinzugefügt werden, wird ebenfalls zuerst der aktuelle Kontostand benötigt. Danach kann dieser gelöscht werden und der neue Kontostand, der sich aus der Summe des alten und dem hinzuzufügenden Betrag ergibt, kann in die Datenbank geschrieben werden.

Diese drei Regeln sind die Transaktionsbasis P dieses Beispiels. In der bereits erwähnten Datenbank sollen die Konten mit dem zugehörigen Guthaben gespeichert sein. Für jedes Konto muss also ein Paar aus Kontonummer und Kontostand gespeichert werden. Für dieses Beispiel sollen zwei Konten *a1* und *a2* mit einem Guthaben von 90 bzw. 50 EUR genügen. Somit ist die Datenbank

$$D = \{balance(a1, 90), balance(a2, 50)\}.$$

Zuletzt müssen noch die elementaren Datenbankoperationen definiert werden. Wie oben festgelegt, beginnen diese auch hier mit *del:*, *ins:* bzw. *get:*. Damit sind die elementaren Transitionen hier

$$\langle D \cup \{balance(Acct, B), D\}\rangle\, del{:}balance(Acct, B)$$

,

$$\langle D, D \cup \{balance(Acct, B)\}\rangle\, ins{:}balance(Acct, B)$$

und

$$\langle D \cup \{balance(Acct, B), D \cup \{balance(Acct, B)\}\rangle\, get{:}balance(Acct, B).$$

□

Die bisher beschriebenen Mengen werden nun zu einem Tripel zusammengefasst, das als $\mathcal{TR}$-Programm bezeichnet wird.

Definition 52 Sei $\mathcal{B}$ eine Transitionsbasis, P eine Transaktionsbasis und D eine Datenbasis, dann ist das Tripel $Prog_{\mathcal{TR}} = (\mathcal{B}, P, D)$ ein *$\mathcal{TR}$-Programm.*

□

Im folgenden und insbesondere bei der Inferenz wird immer wieder eine besondere Notation verwendet, das sog. *executional entailment*:

$$(\mathcal{B}, P, D_1, \ldots, D_n) \vdash \alpha$$

Hierbei ist $\mathcal{B}$ eine Transitionsbasis, P eine Transaktionsbasis, $D_1, \ldots, D_n$ eine Sequenz von Datenbankzuständen und α eine auszuführende $\mathcal{TR}$-Formel. Der gesamte Ausdruck besagt nun, dass man von D_1 über beliebig viele Zwischenzustände nach D_n kommt wenn man α ausführt. Zusätzlich gibt es zu dieser Notation noch zwei weitere, ähnliche Abkürzungen:

$$(\mathcal{B}, P, D—) \vdash \alpha$$

besagt, dass im Zustand D α ausführt werden kann. Die Nachfolgezustände sind hierbei nicht von Interesse. Analog dazu bedeutet

$$(\mathcal{B}, P, —D) \vdash \alpha,$$

dass nach Ausführen von α der Zustand D erreicht wird.

Durch den Konnektor | kann es vorkommen, dass zu einem bestimmten Zeitpunkt mehrere Subtransaktionen gleichzeitig ausführbar sind. Diese ausführbaren Transaktionen werden in einer speziellen Menge *hot* festgehalten, die bei der Inferenz in der $\mathcal{TR}$ Verwendung findet. Die folgende Definition von *hot* ist [BoKi96, S. 10] entnommen und wurde um die dort fehlenden Operatoren $\vee$, $\wedge$ und $\oplus$ erweitert.

Definition 53 Seien ϕ_1 und ϕ_2 konkurrente serielle Ziele. Die Menge der *heißen Komponenten* (engl. hot component) $hot(\phi_i)$ von ϕ_i ist rekursiv definiert als

- $hot(()) = \{\}$, wobei () das leere Ziel ist,
- $hot(b) = \{b\}$, für ein Atom b,
- $hot(\phi_1 \wedge \phi_2) = hot(\phi_1) \cap hot(\phi_2)$,
- $hot(\phi_1 \vee \phi_2) = hot(\phi_1) \cup hot(\phi_2)$,
- $hot(\phi_1 \otimes \phi_2) = hot(\phi_1)$,
- $hot(\phi_1 \oplus \phi_2) = hot(\phi_1) \cup hot(\phi_2)$,
- $hot(\phi_1 | \phi_2) = hot(\phi_1) \cup hot(\phi_2)$ und
- $hot(\odot\phi_1) = \{\odot\phi_1\}$.

□

Außer der Funktion *hot* werden in der folgenden Inferenz zwei Relationen, die als Orakel bezeichnet werden, benutzt. Das Datenorakel $\mathcal{O}^d$ liefert zu einem Zustand einer Datenbasis alle $\mathcal{TR}$-Formeln, die in diesem Zustand wahr sind und das Transitionsorakel $\mathcal{O}^t$ liefert zu zwei (nicht notwendigerweise verschiedenen) Zuständen von Datenbasen alle $\mathcal{TR}$-Formeln, die einen Übergang vom ersten Zustand der Datenbasis in den zweiten formulieren. Die folgende Definition hält dies formal fest.

Definition 54 Seien $Prog_{\mathcal{TR}} = (\mathcal{B}, P, D)$ ein $\mathcal{TR}$-Programm und D_1, D_2 zwei beliebige Zustände der Datenbasis von $Prog_{\mathcal{TR}}$. Außerdem sei I eine Interpretation. Das *Datenorakel* $\mathcal{O}^d(D_1)$ ist eine Relation

$$\mathcal{O}^d(D_1) = \{b | (\langle D_1, D_2 \rangle b) \in \mathcal{B}\}$$

und das *Transitionsorakel* $\mathcal{O}^t(D_1, D_2)$ ist eine Relation

$$\mathcal{O}^t(D_1, D_2) = \{b | (\langle D_1, D_2 \rangle b) \in \mathcal{B}\}$$

□

Definition 55 Für die *Inferenz* gelten in der $\mathcal{TR}$ die folgenden Regeln. Dabei sei $\mathcal{B}$ eine Transitionsbasis, P eine Transaktionsbasis, $b \leftarrow \beta \in P$ eine Regel, D eine beliebige Datenbasis und ψ ein konkurrentes serielles Ziel. Außerdem sei $\boxdot \in \{\wedge, \vee, \otimes, \oplus, |\}$.

- **Axiome**: $(\mathcal{B}, P, D—) \vdash (), \forall \mathcal{B}, D, P$
- **Inferenzregeln**:
  1. (*Anwendung von Transaktionsdefinitionen*)

     $$\frac{(\mathcal{B}, P, D—) \vdash \psi}{(\mathcal{B}, P, D—) \vdash \psi'}$$

     wobei in $\psi = \gamma_1 \boxdot \cdots \boxdot \gamma_i \boxdot b \boxdot \gamma_{i+1} \boxdot \cdots \boxdot \gamma_n$ ein heißes Vorkommen von b durch β ersetzt wird und so $\psi' = \gamma_1 \boxdot \cdots \boxdot \gamma_i \boxdot \beta \boxdot \gamma_{i+1} \boxdot \cdots \boxdot \gamma_n$ ist.
     Zum Beispiel wird $\psi = b \otimes c \wedge d$ zu $\psi' = \beta \otimes c \wedge d$.
  2. (*Datenbankanfragen*) Wenn $b \in \mathcal{O}^d(D)$:

     $$\frac{(\mathcal{B}, P, D—) \vdash \psi}{P, D— \vdash \psi'}$$

     wobei in $\psi = \gamma_1 \boxdot \cdots \boxdot \gamma_i \boxdot b \boxdot \gamma_{i+1} \boxdot \cdots \boxdot \gamma_n$ ein heißes Vorkommen von b entfernt wird und so $\psi' = \gamma_1 \boxdot \cdots \boxdot \gamma_n$ ist.
     Zum Beispiel wird $\psi = b \otimes c \wedge d$ zu $\psi' = c \wedge d$.
  3. (*Datenbankupdates*) Wenn $b \in \mathcal{O}^t(D_1, D_2)$:

     $$\frac{(\mathcal{B}, P, D_1—) \vdash \psi}{(\mathcal{B}, P, D_2—) \vdash \psi'}$$

     wobei in $\psi = \gamma_1 \boxdot \cdots \boxdot \gamma_i \boxdot b \boxdot \gamma_{i+1} \boxdot \cdots \boxdot \gamma_n$ ein heißes Vorkommen von b entfernt wird und so $\psi' = \gamma_1 \boxdot \cdots \boxdot \gamma_n$ ist.
     Zum Beispiel wird $\psi = b \otimes c \wedge d$ zu $\psi' = c \wedge d$.

4. (*Ausführen atomarer Transaktionen*) Wenn $\odot\alpha \in hot(\psi)$:

$$\frac{(\mathcal{B}, P, D—) \vdash \psi}{(\mathcal{B}, P, D—) \vdash (\alpha \otimes \psi')}$$

wobei in $\psi = \gamma_1|\ldots|\gamma_i|(\odot\alpha)|\gamma_{i+1}|\ldots|\gamma_n$ ein heißes Vorkommen von $\odot\alpha$ entfernt wird und so $\psi' = \gamma_1|\ldots|\gamma_n$ ist.
Zum Beispiel wird $\psi = b|(\odot\alpha)|d$ zu $\psi' = c|d$.

[BoKi96]

Diese Regeln werden im folgenden Beispiel (zumindest zum Teil) angewendet, um zu überprüfen ob ein gegebenes Ziel zum aktuellen Zeitpunkt erreicht werden kann. Dazu wird auf das Beispiel 8 von S. 46 zurückgegriffen. $\mathcal{B}$, P und D sind bereits dort vorgegeben.

Beispiel 9 Es sollen 30 EUR von Konto *a1* auf das Konto *a2* überwiesen werden. Formal bedeutet dies, dass unser Ziel $\alpha = payTo(30, a2)$ ist. Da in diesem Beispiel der |-Operator nicht vorkommt, kann auf die Berechnung der *hot*-Mengen verzichtet werden. Es gilt also zu beweisen, dass

$$(\mathcal{B}, P, \{balance(a1, 90), balance(a2, 50)\}) \vdash payTo(30, a2).$$

Unter Verwendung der oben gegebenen Inferenzregeln wird jetzt versucht, daraus ein Axiom (in diesem Fall $(\mathcal{B}, P, D) \vdash ()$) herzuleiten. Die verwendeten Regeln sind in Klammern angegeben.

$(\mathcal{B}, P, \{balance(a1, 90) balance(a2, 50)\})$	$\vdash$	$payTo(30, a2)$	(1)
$(\mathcal{B}, P, \{balance(a1, 90), balance(a2, 50)\})$	$\vdash$	$withdraw(30, a1) \otimes$ $deposit(30, a2)$	(1)
$(\mathcal{B}, P, \{balance(a1, 90), balance(a2, 50)\})$	$\vdash$	$get{:}balance(a1, B) \otimes$ $B \geq 60 \otimes$ $del{:}balance(a1, B) \otimes$ $ins{:}balance(a1, B - 30) \otimes$ $deposit(30, a2)$	(2)
$(\mathcal{B}, P, \{balance(a1, 90), balance(a2, 50)\})$	$\vdash$	$90 \geq 30 \otimes$ $del{:}balance(a1, 90) \otimes$ $ins{:}balance(a1, 90 - 30) \otimes$ $deposit(30, a2)$	(*)
$(\mathcal{B}, P, \{balance(a1, 90), balance(a2, 50)\})$	$\vdash$	$del{:}balance(a1, 90) \otimes$ $ins{:}balance(a1, 90 - 30) \otimes$ $deposit(30, a2)$	(3)
$(\mathcal{B}, P, \{balance(a2, 50)\})$	$\vdash$	$ins{:}balance(a1, 90 - 30) \otimes$ $deposit(30, a2)$	(3)
$(\mathcal{B}, P, \{balance(a1, 60), balance(a2, 50))\}$	$\vdash$	$deposit(30, a2)$	(1)
$(\mathcal{B}, P, \{balance(a1, 60), balance(a2, 50))\}$	$\vdash$	$get{:}balance(a2, B) \otimes$ $del{:}balance(a2, B) \otimes$ $ins{:}balance(a2, B + 30)$	(2)
$(\mathcal{B}, P, \{balance(a1, 60), balance(a2, 50)\})$	$\vdash$	$del{:}balance(a2, 50) \otimes$ $ins{:}balance(a2, 50 + 30)$	(3)
$(\mathcal{B}, P, \{balance(a1, 60)\})$	$\vdash$	$ins{:}balance(a2, 50 + 30)$	(3)
$(\mathcal{B}, P, \{balance(a1, 60), balance(a2, 80)\})$	$\vdash$	$()$	

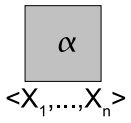
Zur Regel '*' muss noch etwas gesagt werden. Das Zeichen '$\geq$' kann der Einfachheit halber als Prädikat verstanden werden, das in einer hier nicht spezifizierten Datenbasis überprüft, ob ein Datum $\geq (90, 30)$ in dieser enthalten ist. Dies sei hier der Fall.

□

Nach diesem abschließenden Beispiel soll im nächsten Kapitel eine Methode beschrieben werden, die es ermöglicht $\mathcal{TR}$-Programme als Petri-Netze zu implementieren.

4.3.2 Transformation der Transaktionslogik in pr/t-Netze

Im Gegensatz zur Aussagen- und auch zur Prädikatenlogik stellen die atomaren Formeln in der $\mathcal{TR}$ keine Zustände sondern Aktionen dar. Dadurch werden diese hier nicht in äquivalente Stellen, sondern in äquivalente Transitionen übertragen. Somit wird für ein Atom $\alpha(X_1, \ldots, X_n)$ eine Transition t_α mit den freien Variablen $X_1, \ldots, X_n$ und somit Arität n erstellt:



Sei $p() = \alpha() \boxdot \beta()$ mit $\boxdot \in \{\wedge, \vee, \otimes, \oplus, |\}$ eine $\mathcal{TR}$-Formel. Die Netz-Implementierungen von p sind in den folgenden Abb. 4.10 bis 4.13 gegeben. Ist

$$p(X_1, \ldots, X_n, Y_1, \ldots, Y_m) = \alpha(X_1, \ldots, X_n) \wedge \beta(Y_1, \ldots, Y_m)$$

oder

$$p(X_1, \ldots, X_n, Y_1, \ldots, Y_m) = \alpha(X_1, \ldots, X_n) | \beta(Y_1, \ldots, Y_m),$$

wird p wie in Abb. 4.10 gezeigt implementiert. Wie zu sehen ist, wurde hierbei das Pattern des UND-Splits in Abb. 3.7 bzw. des UND-Joins in Abb. 3.8 verwendet.

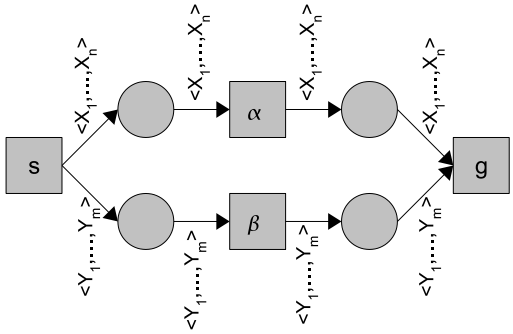


Abb. 4.10: pr/t-Netz-Implementierung von $\alpha \wedge \beta$ und $\alpha | \beta$

Ebenso wird

$$p(X_1, \ldots, X_n, Y_1, \ldots, Y_m) = \alpha(X_1, \ldots, X_n) \vee \beta(Y_1, \ldots, Y_m)$$

wie in Abb. 4.11 zu darstellt ist, dem ODER-Pattern aus Abb. 3.11 bzw. Abb. 3.12 folgend als pr/t-Netz implementiert.

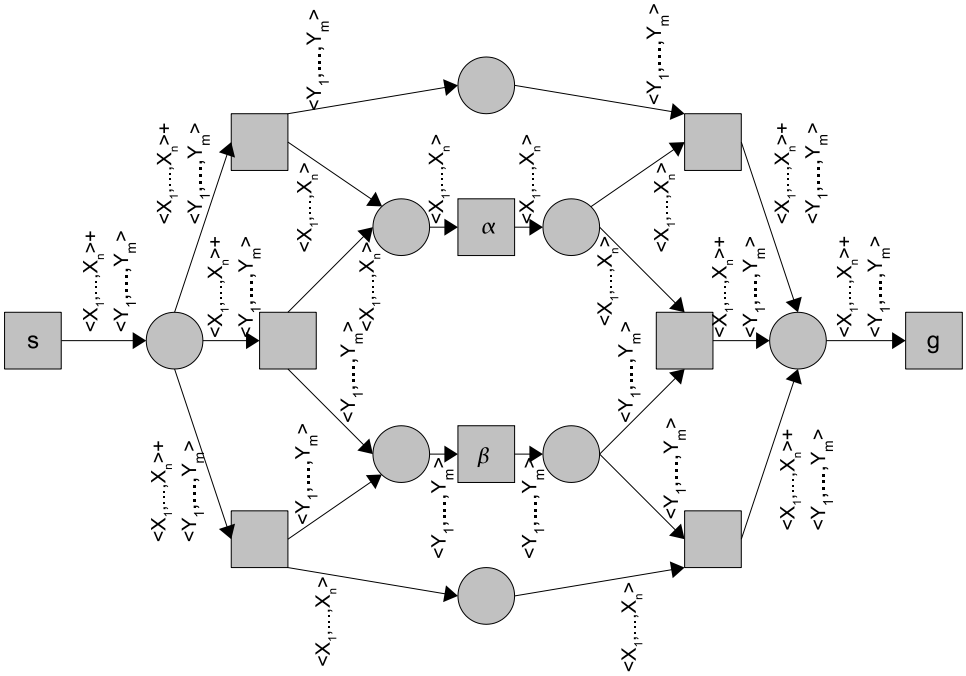


Abb. 4.11: pr/t-Netz-Implementierung von $\alpha \vee \beta$

Um Formeln, die durch die beiden Konnektoren $\otimes$ und $\oplus$ verbunden sind zu implementieren, muss hingegen ein neues Pattern entworfen werden. Um für

$$pr_1(X_1, \ldots, X_n, Y_1, \ldots, Y_m, Z_1, \ldots, Z_o) = \alpha(X_1, \ldots, X_n, Z_1, \ldots, Z_o) \otimes \beta(Y_1, \ldots, Y_m, Z_1, \ldots, Z_o)$$

und

$$pr_2(X_1, \ldots, X_n, Y_1, \ldots, Y_m, Z_1, \ldots, Z_o) = \alpha(X_1, \ldots, X_n, Z_1, \ldots, Z_o) \oplus \beta(Y_1, \ldots, Y_m, Z_1, \ldots, Z_o)$$

α und β jeweils durch eine Transition α und β mit der Arität $n + o$ respektive $m + o$ implementieren zu können, werden Stellen benötigt, die die Argumente von α und β aufnehmen. Dies wird durch die drei Stellen $\alpha\backslash\beta$, $\alpha \cup \beta$ und $\beta\backslash\alpha$ realisiert. Dabei nimmt $\alpha\backslash\beta$ die Terme auf, die nur Argument von α sind (also $X_1, \ldots, X_n$), $\alpha \cup \beta$ nimmt die Terme auf, die Argumente beider Formeln sind (d. h. $Z_1, \ldots, Z_o$) und $\beta\backslash\alpha$ nimmt genau die Terme auf, die nur Argument von β sind ($Y_1, \ldots, Y_m$). Das leere Tupel $<>$ dient nur noch dem Kontrollfluss. Dies ist für $pr_1()$ und $pr_2()$ in Abb. 4.12 bzw. Abb. 4.13 abgebildet.

Eine eventuelle vorhandene Datenbank D wird in eine besondere Teilmenge P_D der Stellen P überführt. Für jedes Prädikat $pr(X_1, \ldots, X_n)$, das zu einem beliebigen Zeitpunkt in der Datenbank gespeichert ist wird eine Stelle pr erzeugt, auf der dann die Tupel $< X_1, \ldots, X_n >$ abgelegt werden. Elementare Transaktionen wie $\langle D \cup \{pr(X_1, \ldots, X_n)\}, D\rangle$ u_1, die ein Datum $pr(X_1, \ldots, X_n)$ aus der Datenbank entfernen, werden dargestellt als eine Transition u_1, die beim Feuern das Tupel $< X_1, \ldots, X_n >$ von der Stelle pr herunternimmt:

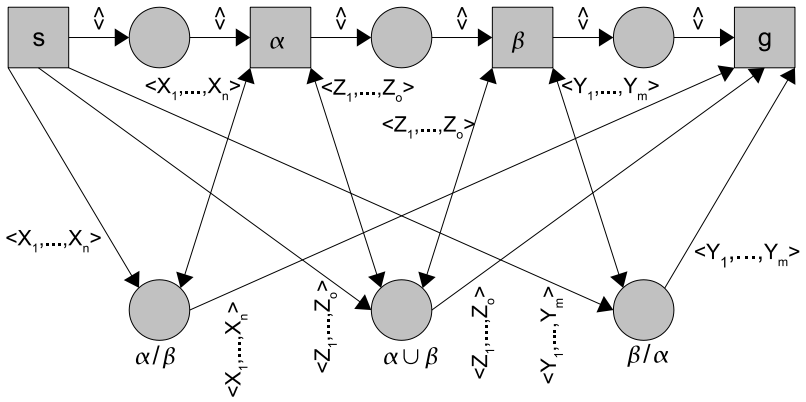


Abb. 4.12: pr/t-Netz-Implementierung von $\alpha \otimes \beta$

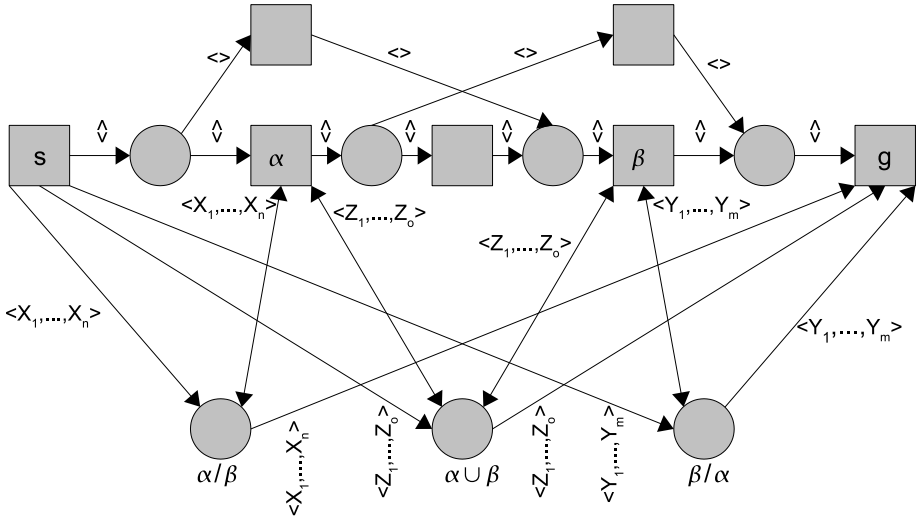
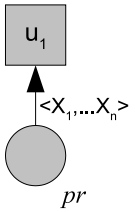
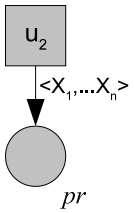


Abb. 4.13: pr/t-Netz-Implementierung von $\alpha \oplus \beta$

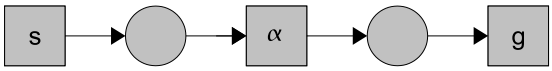


Wird durch eine elementare Transaktion $< D, D \cup \{pr(X_1, \ldots, X_n)\} > u_2$ ein Datum zur Datenbank hinzugefügt, so wird, analog zum vorhin beschriebenen Entfernen eines Datums, eine Transition u_2 im Netz erstellt, die beim Feuern ein Tupel $< X_1, \ldots, X_n >$ auf die Stelle *pr* legt.

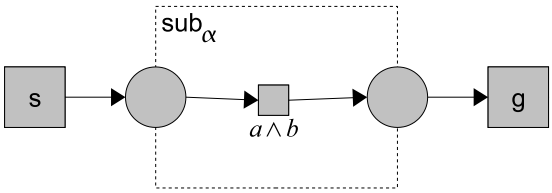


Mit Hilfe der in Abb. 4.10 bis 4.13 gegebenen Pattern für die fünf Konnektoren $\wedge$, $\vee$, $\otimes$, $\oplus$ und $|$ soll nun erklärt werden, wie evtl. vorkommende Subformeln behandelt werden

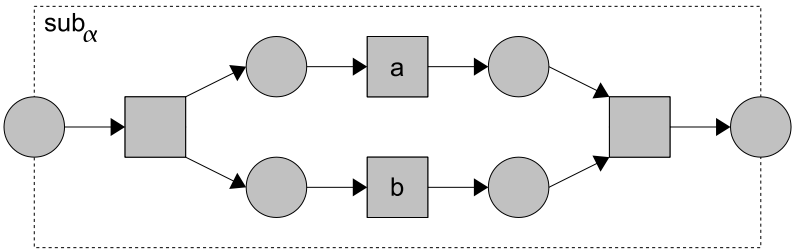
sollen. Ist zum Beispiel $\alpha \leftarrow a \wedge b$ eine $\mathcal{TR}$-Regel, besagt die prozedurale Interpretation, dass man statt α auch alternativ $a \wedge b$ ausführen kann. Für das zu erstellende Petri-Netz bedeutet dies, dass das Feuern der Transition t_α gleichbedeutend ist mit dem Feuern der beiden Transitionen a und b. Mit Hilfe von Subnetzen lässt sich dies wie folgt umsetzen. Zunächst wird die Transition t_α erstellt und von einer Starttransition s und einer Zieltransition g eingerahmt:



Da α aber kein Atom ist, sondern, wie aus seiner Rolle als Klauselkopf ersichtlich ist, eine zusammengesetzte Formel, wird α zum Namen eines Subnetzes:



α wird sozusagen zum Kopf des Subnetzes, während der Rumpf des Subnetzes sub_α die Formel $a \wedge b$ umsetzt:



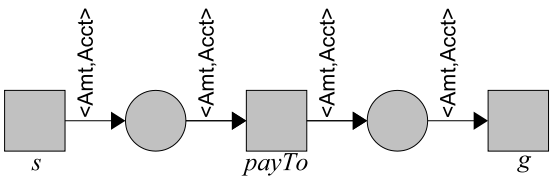
Also ist der Rumpf des Subnetzes gleich dem in der Abb. 4.10 angegebenen Pattern zur Implementierung des Konnektors $\wedge$. Wird nun α wiederum im Rumpf einer anderen Klausel verwendet, wird bei der Übertragung der $\mathcal{TR}$-Formel in eine äquivalente Netzdarstellung dort immer statt der Transition t_α das Subnetz sub_α eingesetzt.

Dies soll nun auf das obige Überweisungsbeispiel aus Bsp. 8 angewendet werden. Das $\mathcal{TR}$-Programm des Beispiels soll jetzt als pr/t-Netz implementiert werden.

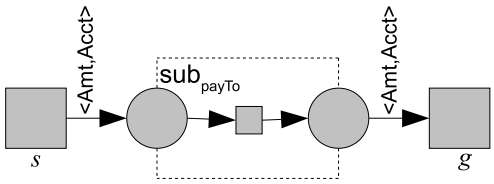
Beispiel 10 Beginnend mit der ersten Klausel

$$payTo(Amt, Acct) \leftarrow withdraw(Amt, a1) \otimes deposit(Amt, Acct)$$

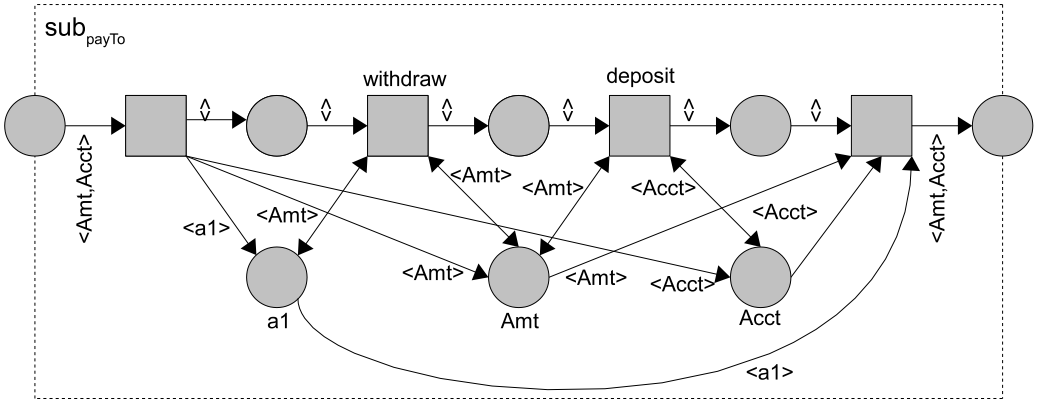
wird zunächst ein Netz erstellt, das die Formel $payTo$ darstellt.



In der Mitte befindet sich die Transtion $payTo$ an deren Ein- und Ausgangskanten sich je ein Tupel $< Amt, Acct >$ befindet. Also können die Stellen des Vor- und Nachbereichs von $payTo$ Tupel genau dieser Form aufnehmen. Eingerahmt wird das Ganze von der Starttransition s, die Tupel der Form $< Amt, Acct >$ erzeugt und von der Zieltransition g, die solche Tupel wieder entfernt. Aus der Klausel ist aber auch direkt zu ersehen, dass $payTo$ ein Prozedurkopf ist und somit durch ein Subnetz sub_{payTo} zu ersetzen ist.



Dieses Subnetz realisiert nun die Sequenz von *withdraw* und *deposit* nach Abb. 4.12, wie sie im Prozedurrumpf von *payTo* angegeben ist.



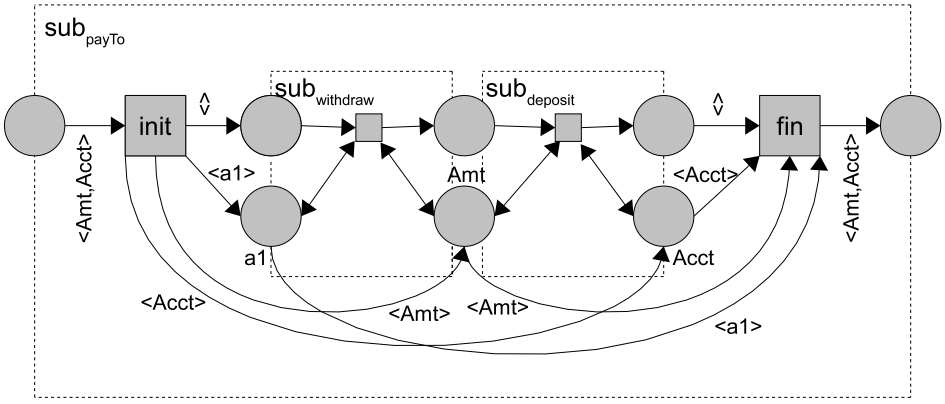
Die drei Stellen $a1$, *Amt* und *Acct* nehmen respektive die Terme (Argumente) auf, die nur in $withdraw(Amt, a1)$, in $withdraw(Amt, a1)$ und in $deposit(Amt, Acct)$ oder nur in $deposit(Amt, Acct)$ vorkommen. Wie aus den beiden Klauseln

$$\begin{aligned} withdraw(Amt, Acct) \quad \leftarrow \quad & get{:}balance(Acct, B) \otimes B \geq Amt \otimes \\ & del{:}balance(Acct, B) \otimes \\ & ins{:}balance(Acct, B - Amt) \end{aligned}$$

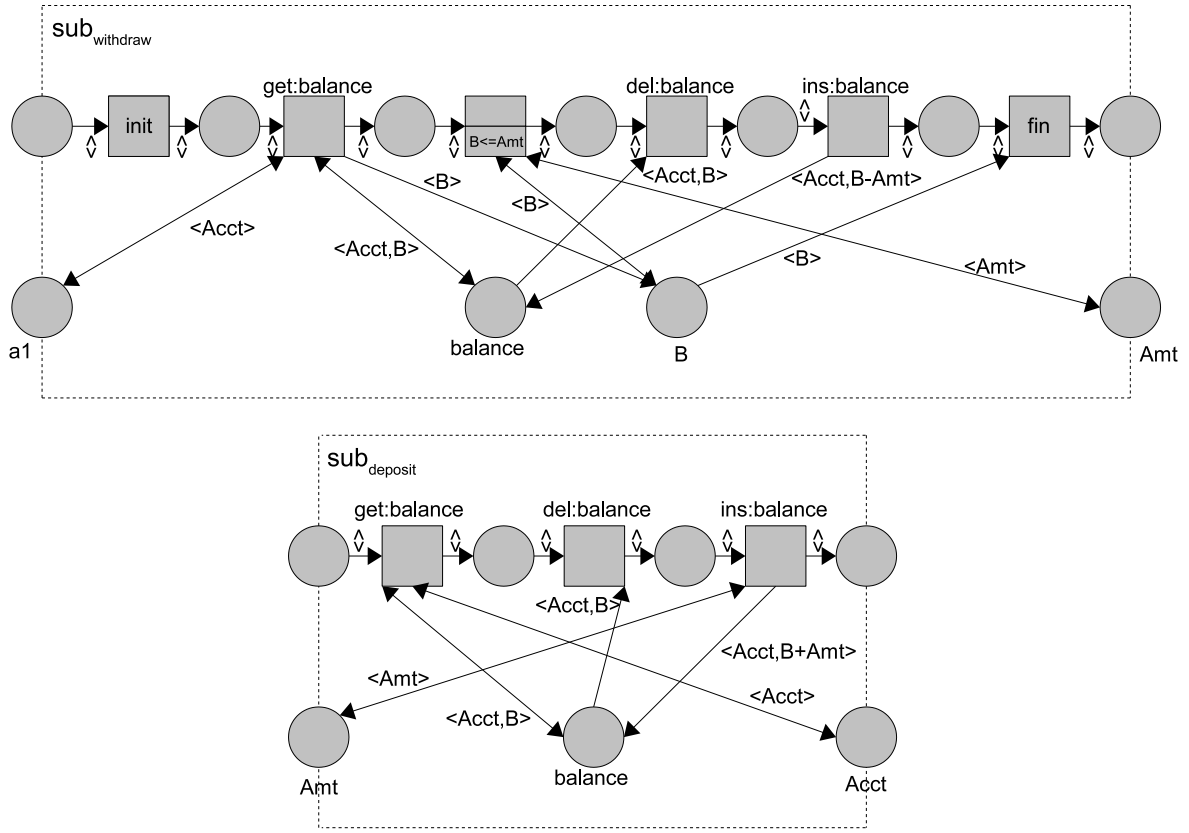
und

$$\begin{aligned} deposit(Amt, Acct) \quad \leftarrow \quad & get{:}balance(Acct, B) \otimes \\ & del{:}balance(Acct, B) \otimes \\ & ins{:}balance(Acct, B + Amt) \end{aligned}$$

ersichtlich ist, sind sowohl $withdraw(Amt, a1)$ als auch $deposit(Amt, Acct)$ zusammengesetzte Formeln. Also werden die beiden Transitionen jeweils durch die Subnetze $sub_{withdraw}$ und $sub_{deposit}$ ersetzt.

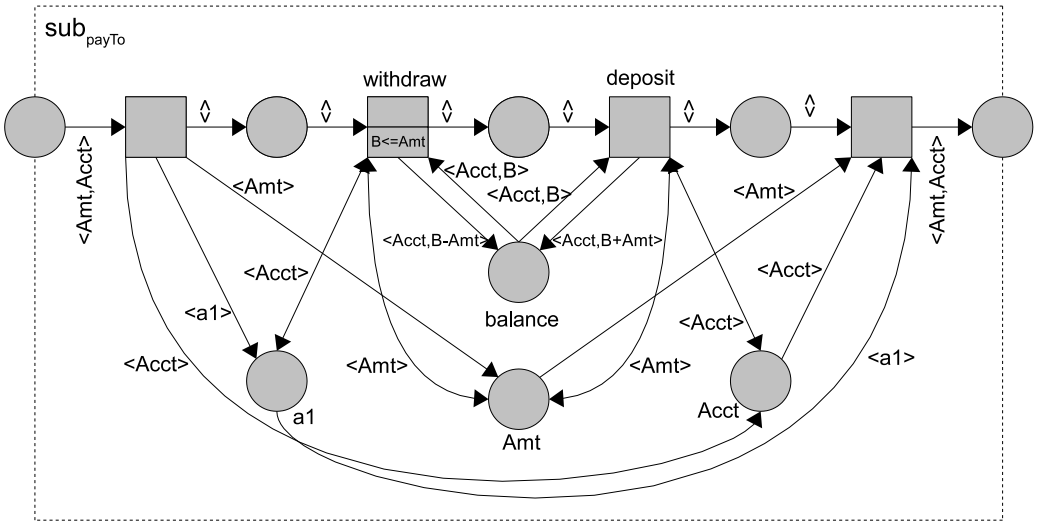


In diesen beiden Subnetzen wird wiederum die Sequenz von atomaren Formeln der Klauselrümpfe von $withdraw(Amt, Acct)$ und $deposit(Amt, Acct)$ umgesetzt.



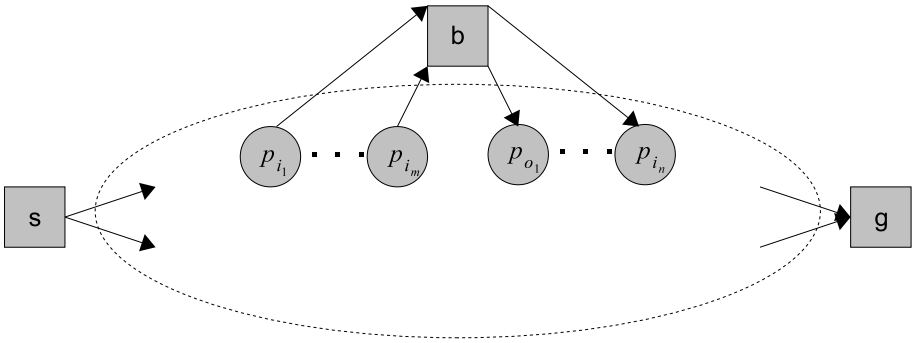
□

Es ist noch anzumerken, dass die Formeln $withdraw(Amt, a1)$ und $deposit(Amt, Acct)$ im pr/t-Netz auch atomar, d. h. aus einer einzigen Transaktion bestehend, dargestellt werden könnten. Zum einen wird sowohl das Entfernen als auch das Hinzufügen eines Datums aus bzw. zu der Datenbank implizit durch die eingehenden und ausgehenden Kanten einer Transition umgesetzt. Zum anderen können zusätzliche Hürden zum Ausführen einer Transaktion als Threshold in die Transition mit aufgenommen werden. So können die Transitionen der beiden Subnetze zu jeweils einer einzigen Transition „verschmolzen" werden. Die so atomarisierten Formeln benötigen auf Grund ihrer Atomizität auch keine Subnetze mehr, sodass nur noch ein einziges Subnetz sub_{payTo} übrig bleibt:



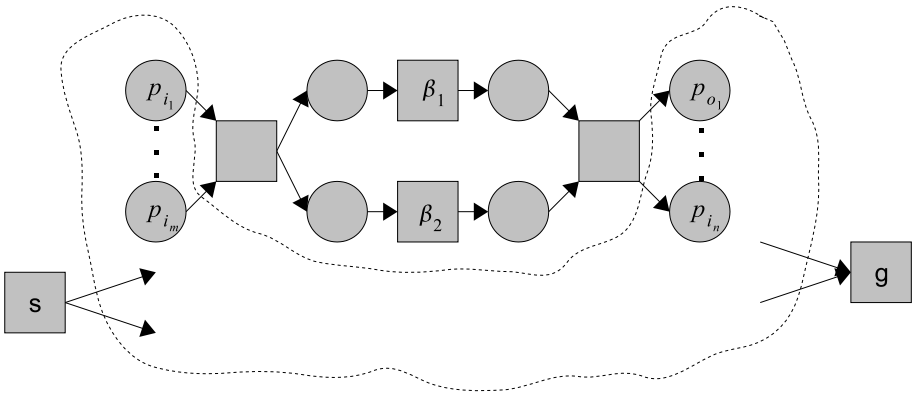
Die Struktur einer beliebigen transaktionslogischen Formel ohne Quantoren kann nun als pr/t-Netz implementiert werden. Es fehlt aber noch eine pr/t-Netz-Darstellung für die Inferenz der $\mathcal{TR}$. Dazu wird eine Übersetzung für die auf S. 48 gegebenen Inferenzregeln benötigt. Anschließend soll dies auf das obige Überweisungsbeispiel angewandt werden.

1. Die erste Regel ist die *Anwendung von Transaktionsdefinitionen*. Sei hierzu die Klausel $b \leftarrow \beta$ die Klausel, auf die diese Regel angewandt werden soll. Genau wie bei der Anwendung dieser Inferenzregel ein Klauselkopf durch den zugehörigen Klauselrumpf ersetzt wird, wird auch in der Netz-Implementierung die Transition b, die dem Klauselkopf entspricht, durch das Subnetz sub_b, das dem Klauselrumpf entspricht, ersetzt. Diese Ersetzungen werden nun für die einzelnen Konnektoren $\wedge$, $\vee$, $\otimes$, $\oplus$ und $|$ unter Verwendung der Pattern aus Abb. 4.10 bis 4.13 gegeben. Um die Netze übersichtlicher zu halten, wird dies hier nur für nullstellige Prädikate gezeigt. Bei mehrstelligen Prädikaten müssen die Netze entsprechend der obigen Pattern erweitert werden. Sei also das Netz $\mathcal{N}_\psi$

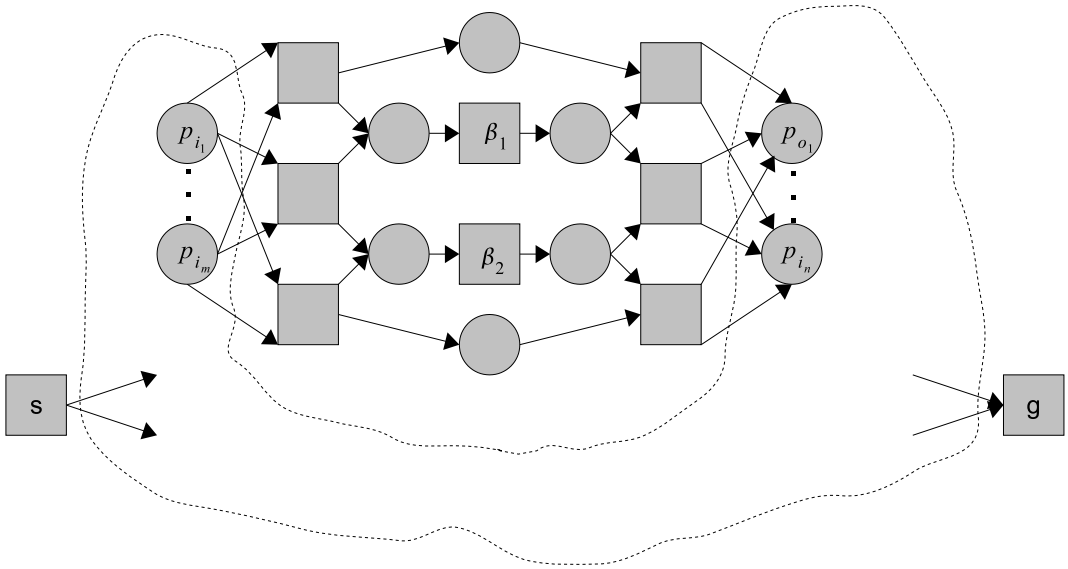


die pr/t-Netz-Implementierung der $\mathcal{TR}$-Formel ψ.

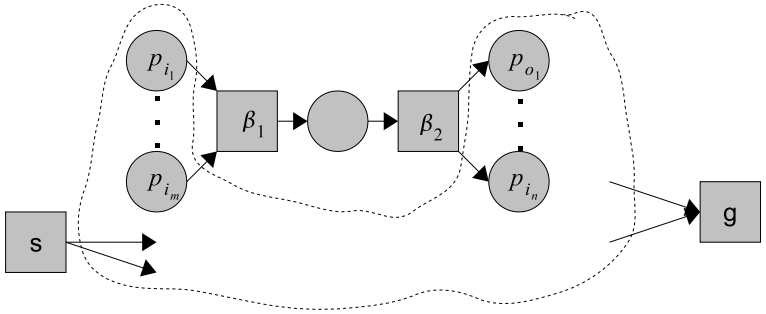
a) Ist $\beta = \beta_1 \boxdot \beta_2$ mit $\boxdot \in \{\wedge, |\}$ so ist das Netz $\mathcal{N}_{\psi'}$:



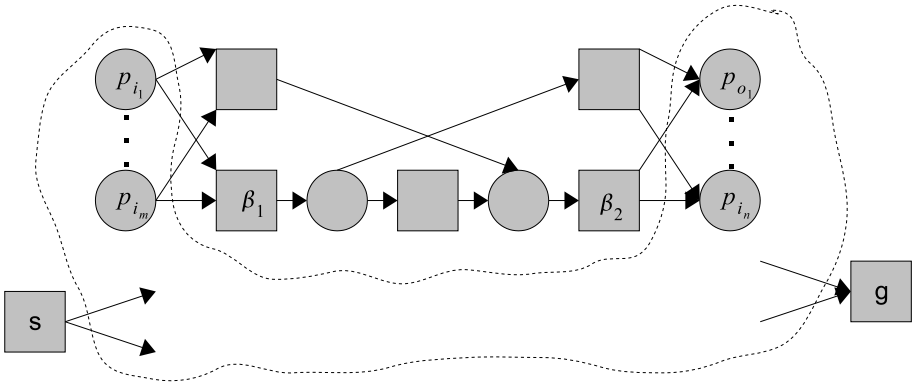
b) Ist $\beta = \beta_1 \vee \beta_2$, so ist das Netz $\mathcal{N}_{\psi'}$:



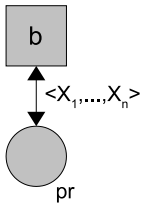
c) Ist $\beta = \beta_1 \otimes \beta_2$, so ist das Netz $\mathcal{N}_{\psi'}$:



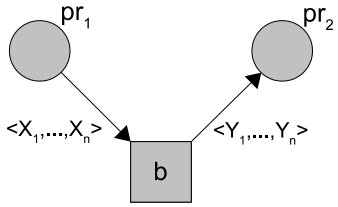
d) Ist $\beta = \beta_1 \oplus \beta_2$, so ist das Netz $\mathcal{N}_{\psi'}$:



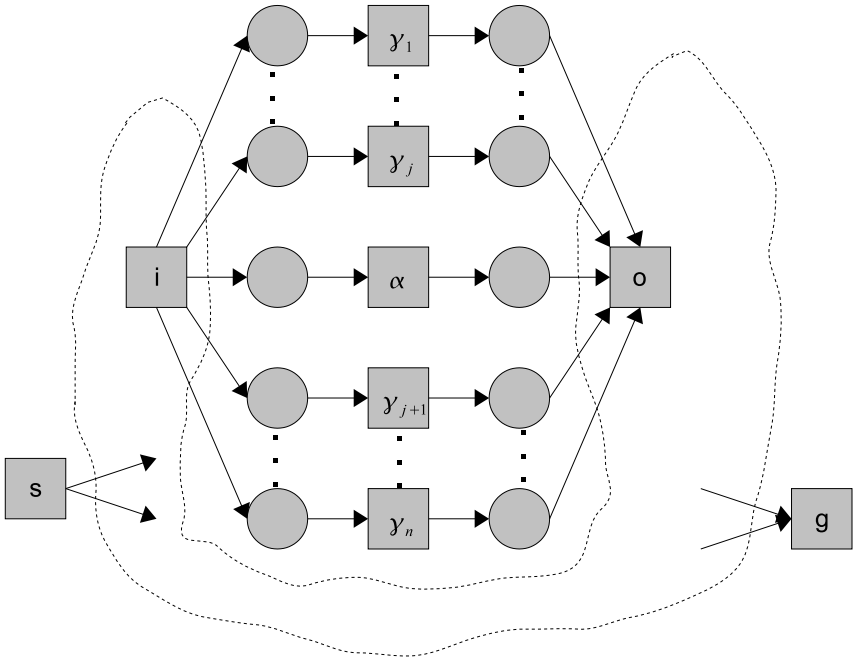
2. Während bei der Umsetzung der ersten Inferenzregel das pr/t-Netz geändert werden musste, ist dies bei *Datenbankanfragen* nicht nötig. Wenn b eine Datenbankanfrage der Form $< D \cup \{pr(X_1, \ldots, X_n)\}, D \cup \{pr(X_1, \ldots, X_n)\} > b$ ist, ändert sich das Netz nicht, d. h. $\mathcal{N}_\psi = \mathcal{N}_{\psi'}$. Die Datenbankanfrage b wird im pr/t-Netz durch das Feuern der Transition b umgesetzt. Dabei wird das Tupel $< X_1, \ldots, X_n >$ von der Stelle *pr* entfernt und sofort wieder dorthin zurückgelegt, d. h. die Markierung der Stelle *pr* ändert sich nicht. Eventuell können im pr/t-Netz aber Tupel (auch $< X_1, \ldots, X_n >$) aus dem Vorbereich der Transition b entfernt und/oder im Nachbereich der Transition erzeugt werden, da in der Netz-Implementierung Stellen und Transitionen vorkommen können, die in der $\mathcal{TR}$-Formel nur indirekt (z. B. beim ODER-Pattern) vorhanden sind. Dies ist dann in der $\mathcal{TR}$-Formel nicht zu sehen. Also ist die Übertragung in ein pr/t-Netz wie folgt:



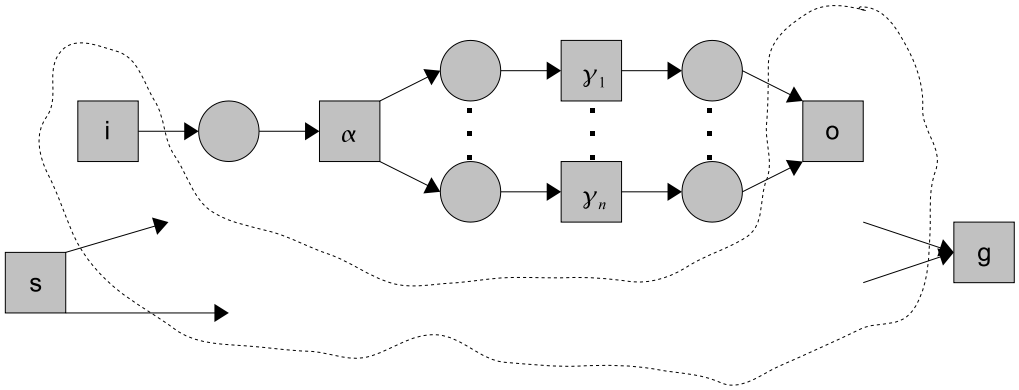
3. *Datenbankupdates* werden ähnlich wie Datenbankanfragen behandelt. Ist b ein Datenbankupdate der Form $< D \cup \{pr_1(X_1, \ldots, X_n)\}, D \cup \{pr_2(Y_1, \ldots, Y_m)\} > b$, so wird dieses Update im pr/t-Netz durch Feuern der Transition b durchgeführt. Dabei wird das Tupel $< X_1, \ldots, X_n >$ von der Stelle pr_1 entfernt und das Tupel $< Y_1, \ldots, Y_m >$ auf die Stelle pr_2 gelegt. Die Netzstruktur selbst ändert sich auch hier nicht.



4. Das *Ausführen atomarer Transaktionen* nach der vierten Inferenzregel erfordert allerdings wieder eine Änderung des Netzes selbst. Sei das folgenden Netz $\mathcal{N}_\psi$



die pr/t-Netz-Implementierung von ψ. Es ist ersichtlich, dass alle Transitionen $\gamma_1, \ldots, \gamma_n$ und α nach Feuern der Transition i konkurrent feuerbar sind. Nach Anwendung der Inferenzregel entsteht das folgende Netz $\mathcal{N}_{\psi'}$,



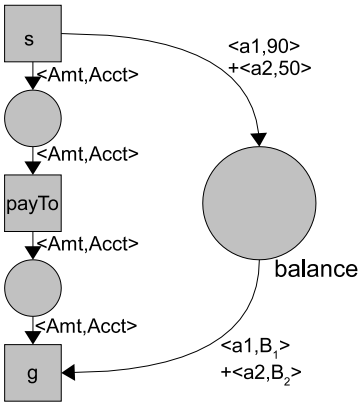
in dem die Transitionen $\gamma_1, \ldots, \gamma_n$ wie in der Inferenzregel vorgegeben erst feuerbar sind, wenn die Transition α gefeuert wurde.

Zusätzlich zu diesen Implementierungsregeln soll die Regelung vereinbart werden, dass nicht explizit benamte Transitionen, d.h. Transitionen, die keinem Atom der implementierten $\mathcal{TR}$-Formel entsprechen, möglichst früh feuern. Dies wird im folgenden Beispiel beachtet.

Beispiel 11 Mit dem gerade Formulierten soll jetzt gezeigt werden, wie der Beweis von

$$(B, P, \{balance(a1, 90), balance(a2, 50)\}) \vdash payTo(30, a2)$$

aus Bsp. 9 in der pr/t-Netz-Implementierung $\mathcal{N}_\psi$ nachvollzogen wird. Das initiale Netz besteht aus der Transition *payTo*, die von einer Starttransition s und einer Zieltransition g umgeben ist. Außerdem gibt es noch eine Stelle *balance*, die die Datenbasis der $\mathcal{TR}$-Formel implementiert. s hat die Aufgabe, die Stelle *balance* mit den Tupeln $< a1, 90 >$ und $< a2, 50 >$ zu befüllen. Außerdem wird durch das Legen des Tupels $< Amt, Acct >$ auf die Stelle zwischen s und *payTo* die Inferenz von $payTo(Amt, Acct)$ angestoßen. Die Aufgabe von g ist es, das Netz durch einmaliges Feuern am Ende wieder zu leeren. Wenn dies gelingt, wurde $payTo(Amt, Acct)$ erfolgreich durchgeführt.[4]

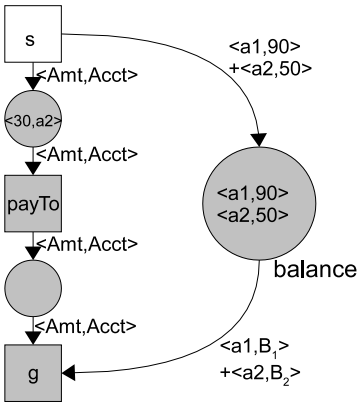


Nach Feuern von s[5] ist die Datenbasis wie gerade beschrieben befüllt und die Anfrage $payTo(30, a2)$ ist an das System gestellt. Dies entspricht dem executional entailment

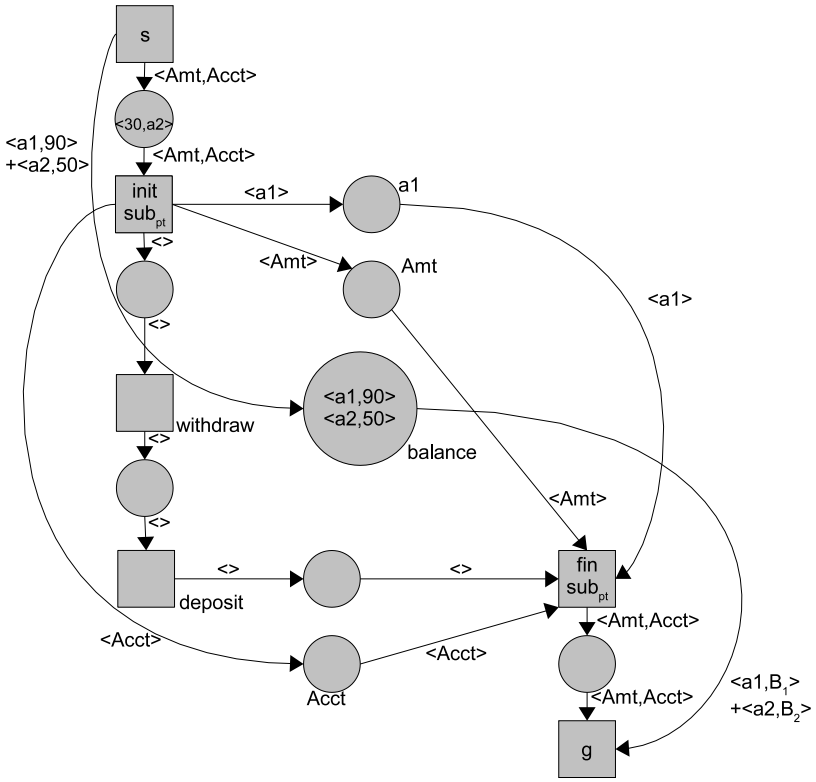
$$(B, P, \{balance(a1, 90) balance(a2, 50)\}) \vdash payTo(30, a2).$$

[4] Vgl. S. 14 Def. 19.

[5] Transitionen, die gerade gefeuert haben, sind im Netz weiß markiert.



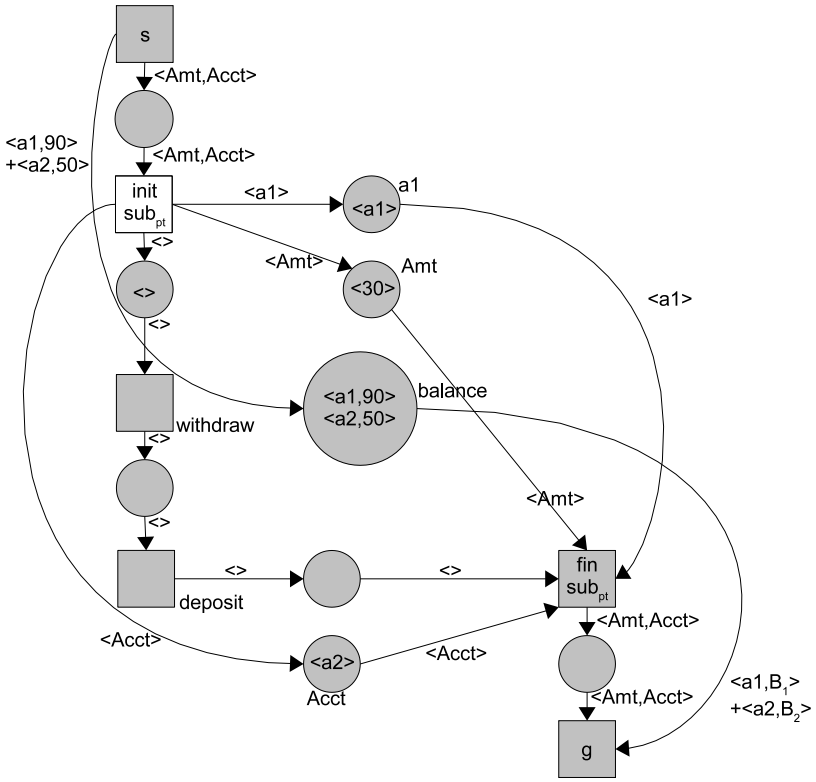
Als nächstes wird die Transition *payTo*, die schließlich einen Klauselkopf implementiert, durch die Implementierung des Klauselrumpfs ersetzt, womit das Netz



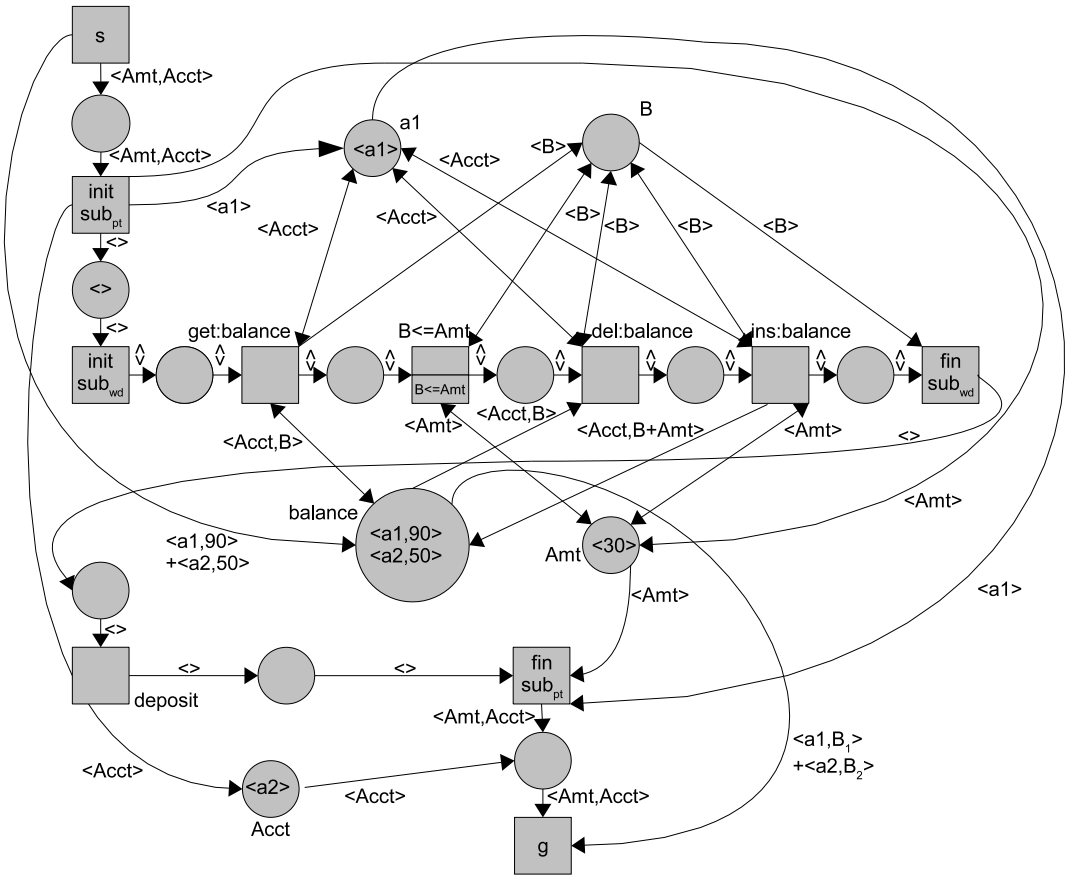
dem Ausdruck

$$(B, P, \{balance(a1, 90), balance(a2, 50)\}) \vdash withdraw(30, a1) \otimes deposit(30, a2)$$

entspricht. Nach Feuern der Transition $initsub_{pt}$,



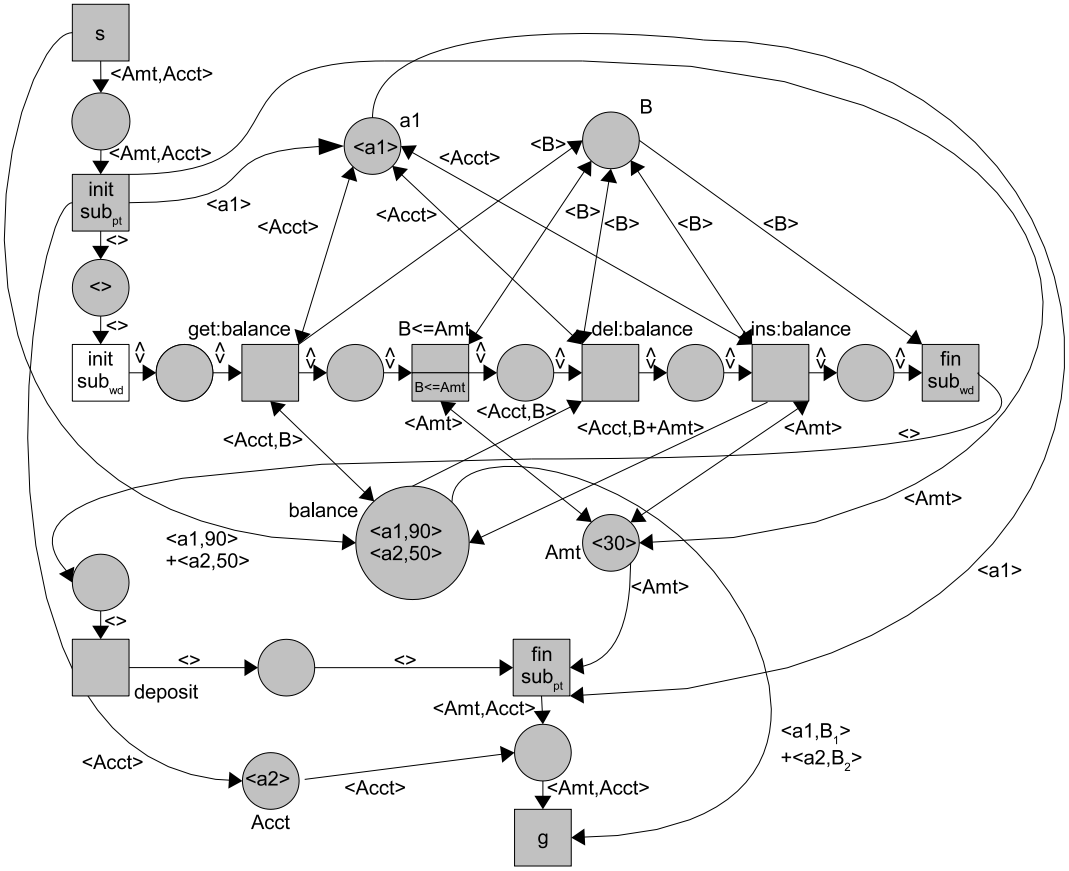
wird die Transition *withdraw* durch den Rumpf des Subnetzes ersetzt.



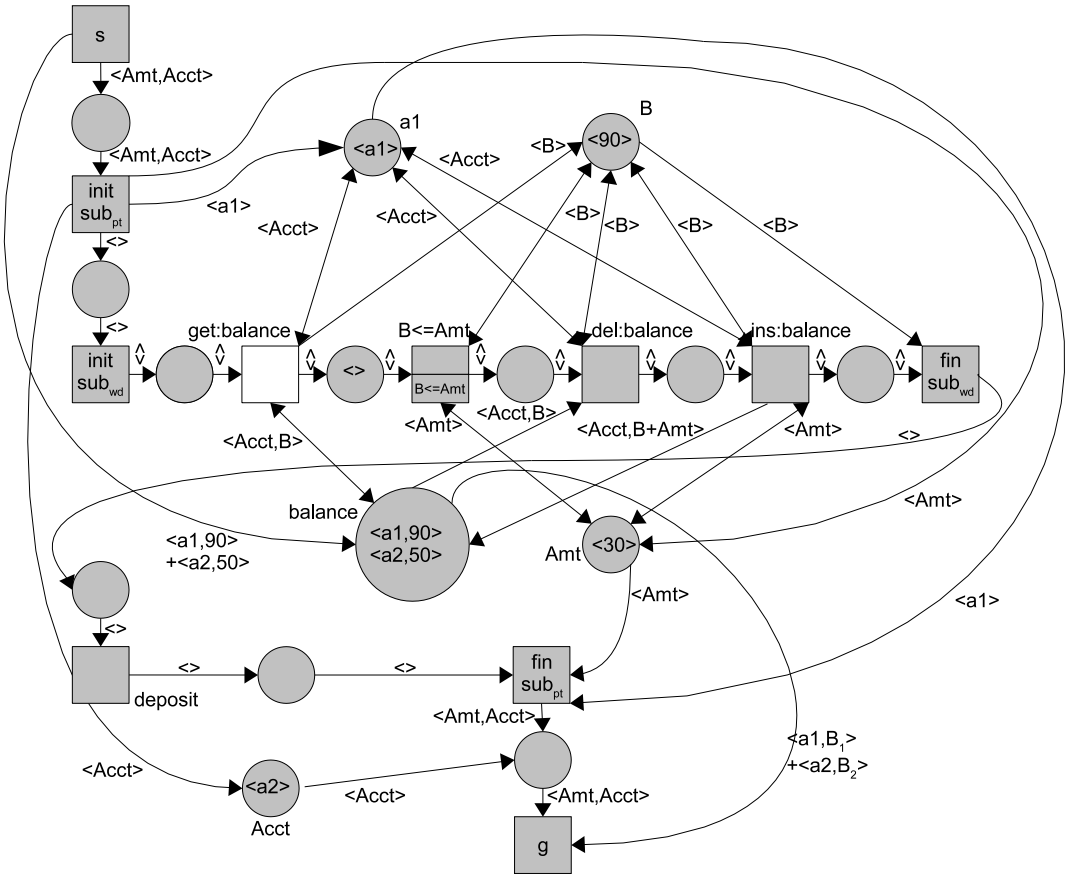
Dies entspricht dem Zustand

$$(B, P, \{balance(a1, 90), balance(a2, 50)\}) \vdash \begin{array}{l} get{:}balance(a1, B) \otimes \\ B \geq 60 \otimes \\ del{:}balance(a1, B) \otimes \\ ins{:}balance(a1, B - 30) \otimes \\ deposit(30, a2). \end{array}$$

Wenn die Transition $initsub_{wd}$ gefeuert hat,



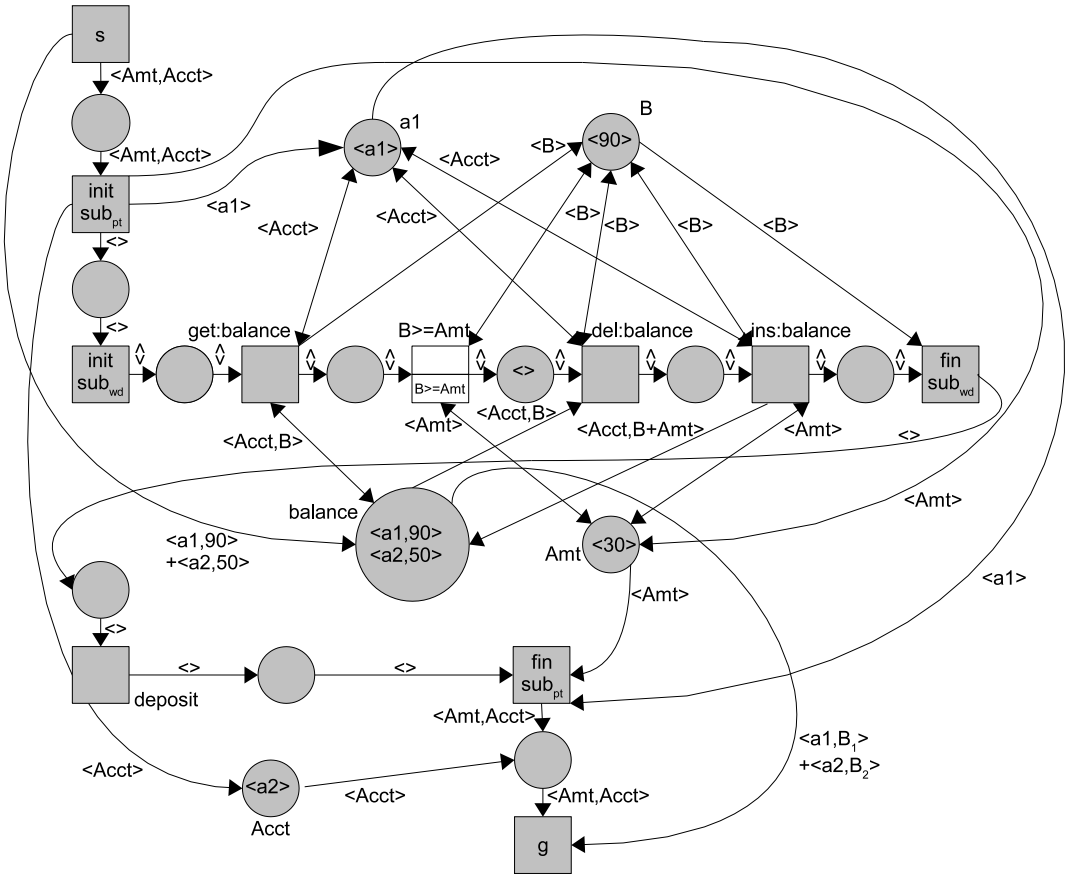
kann die Transition *get:balance* feuern



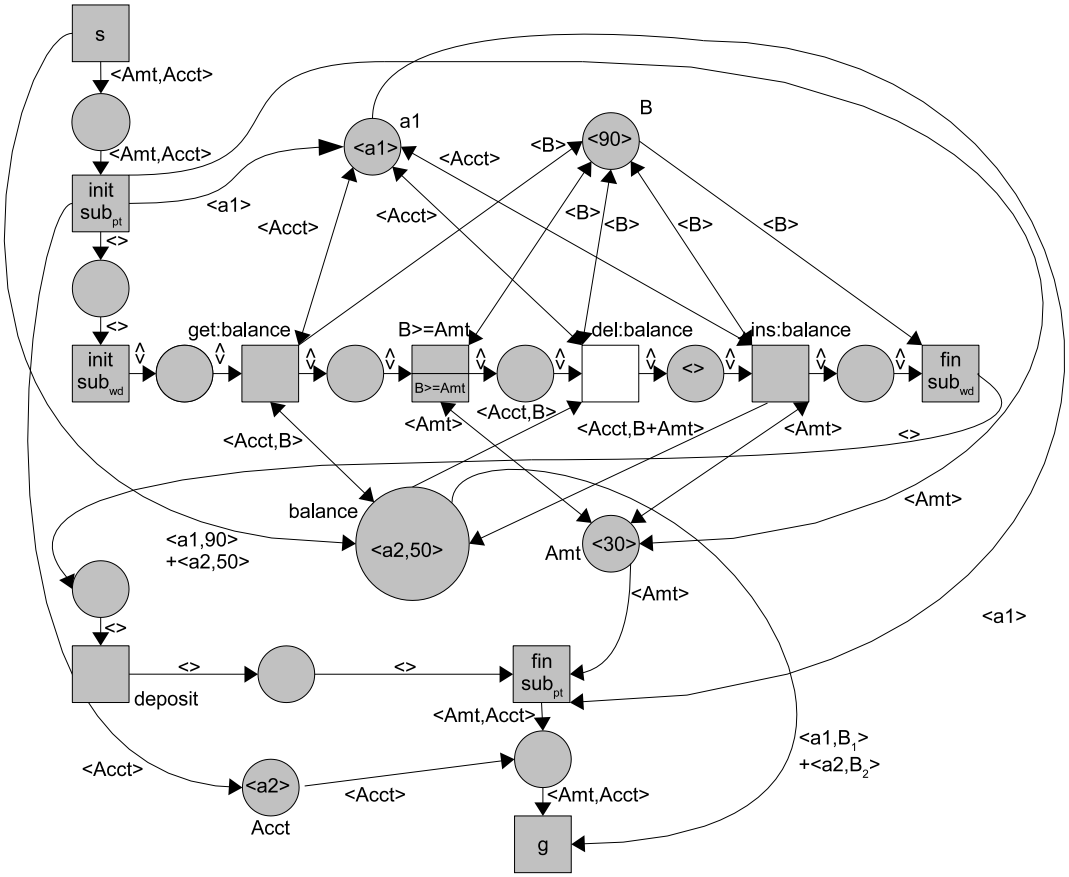
und das Netz erreicht den Zustand, der zu

$$(B, P, \{balance(a1, 90), balance(a2, 50)\}) \vdash (90 \geq 30) \otimes \\ del{:}balance(a1, 90) \otimes \\ ins{:}balance(a1, 90 - 30) \otimes \\ deposit(30, a2)$$

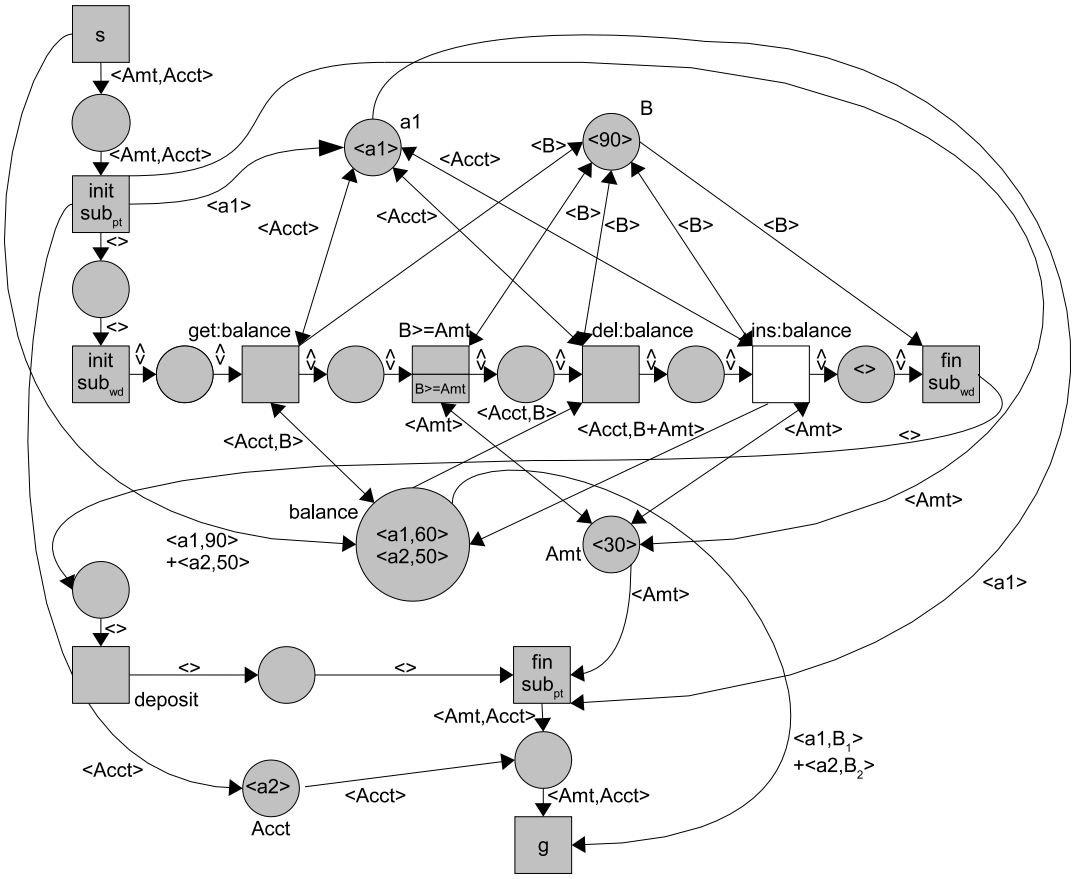
äquivalent ist. Anschließend feuern die Transitionen $B{>}{=}Amt$,



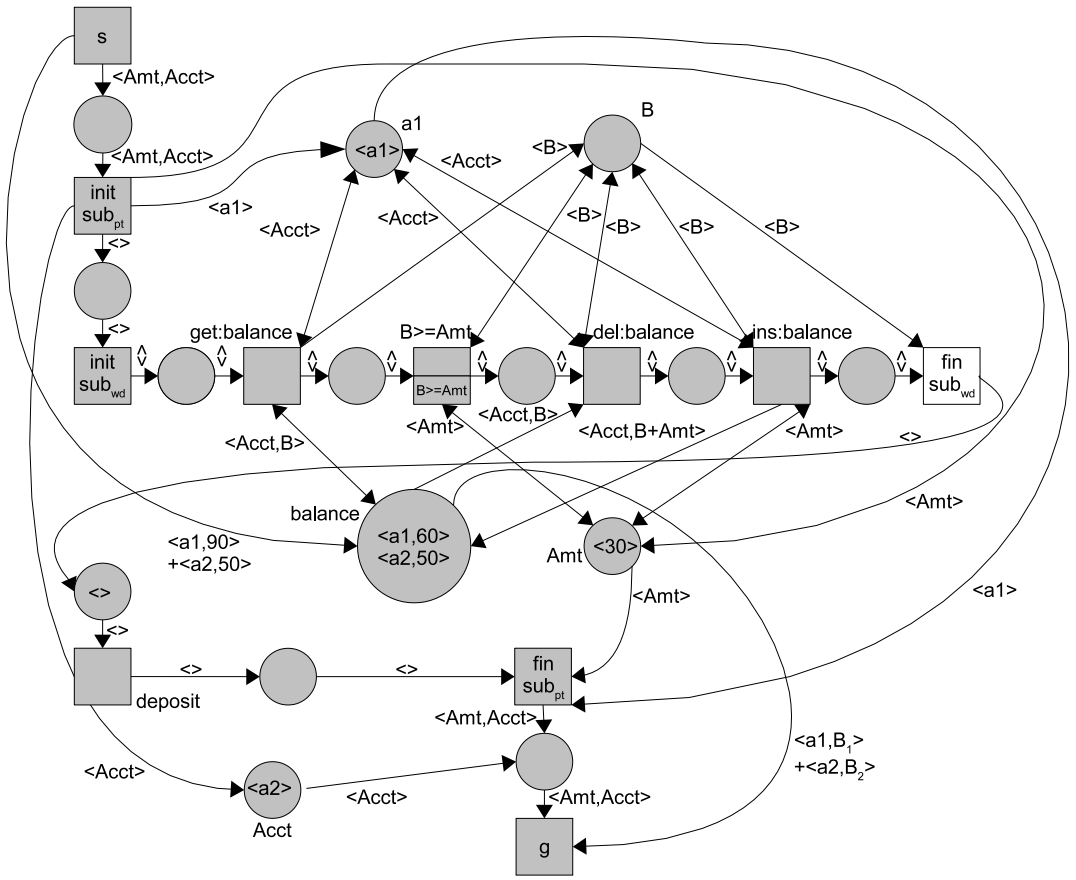
del:balance



und *ins:balance*,



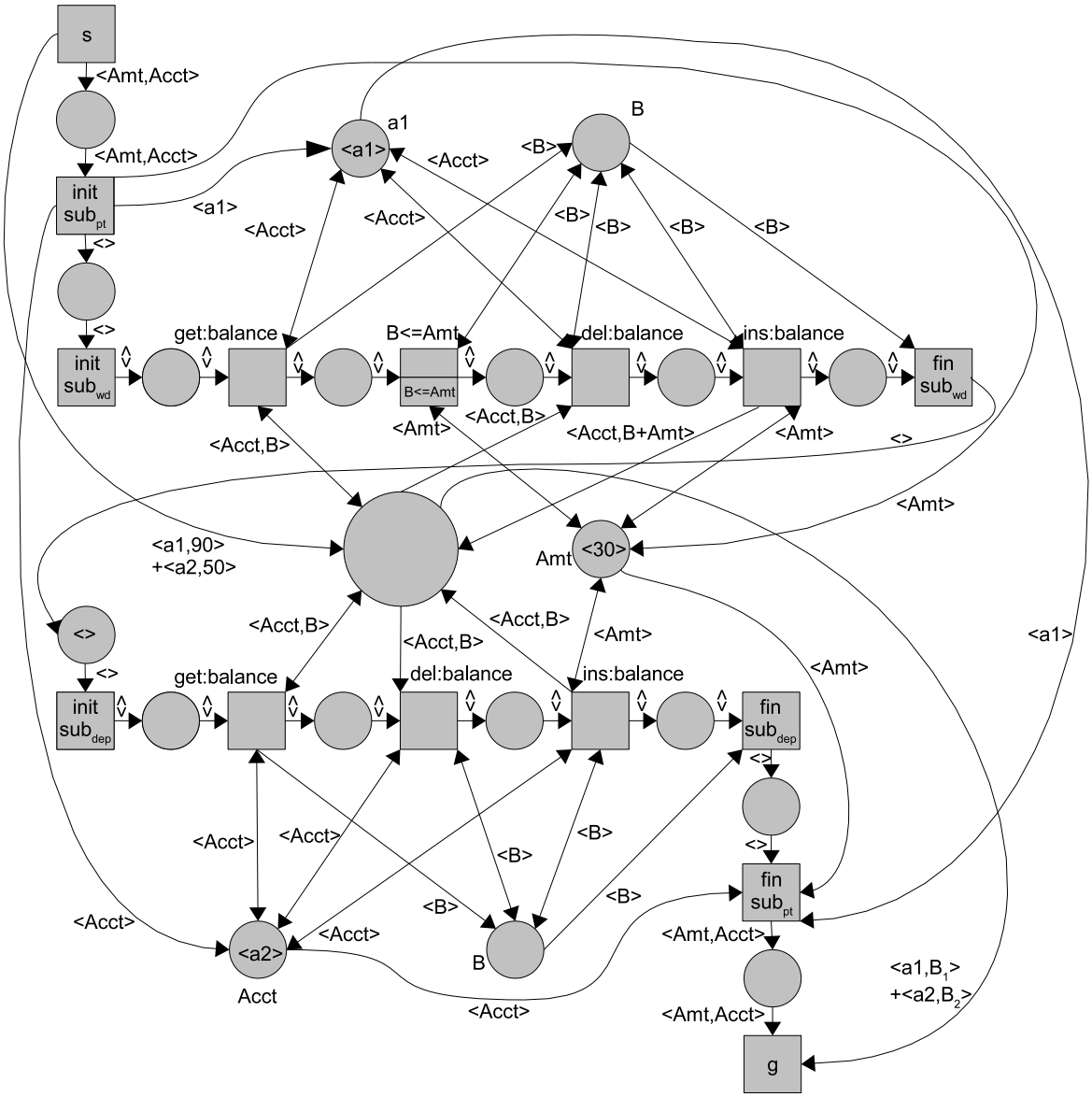
sodass nach Feuern von $finsub_{wd}$



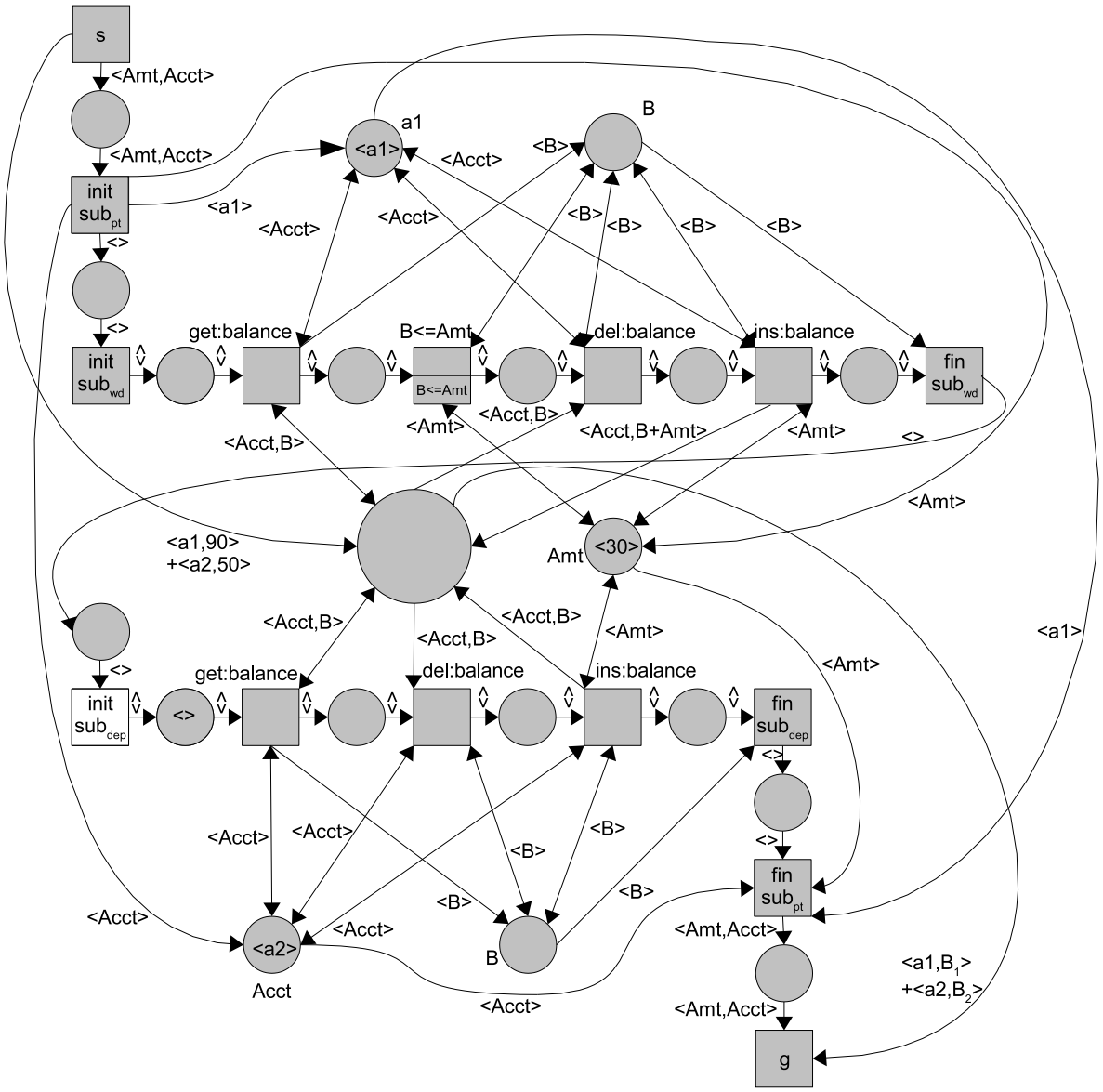
das Netz in einem Zustand ist, der

$$(B, P, \{balance(a1, 40), balance(a2, 50)\}) \vdash deposit(30, a2)$$

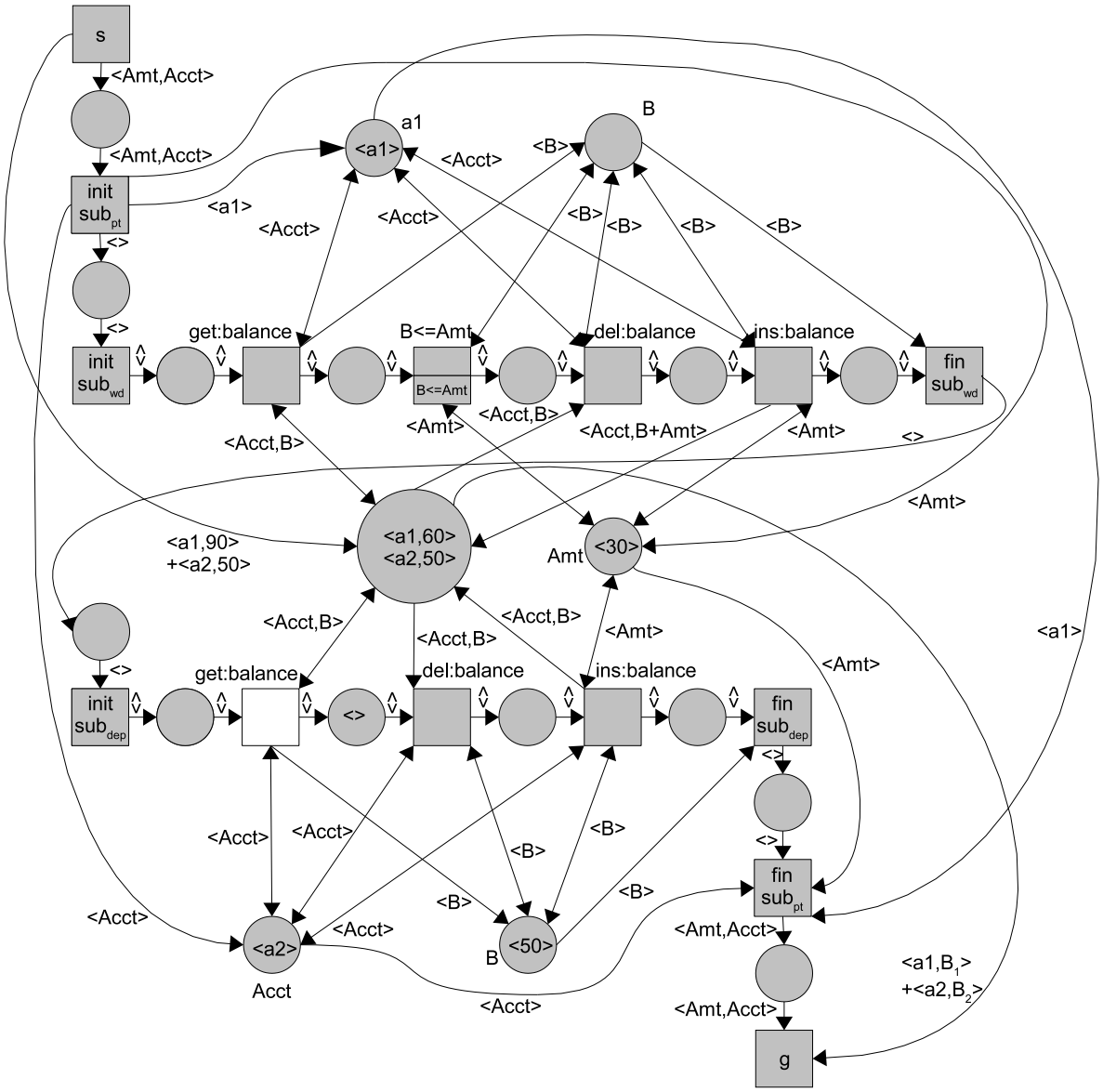
entspricht.
Jetzt wird die Transition *deposit* durch das korrespondierende Subnetz ersetzt.



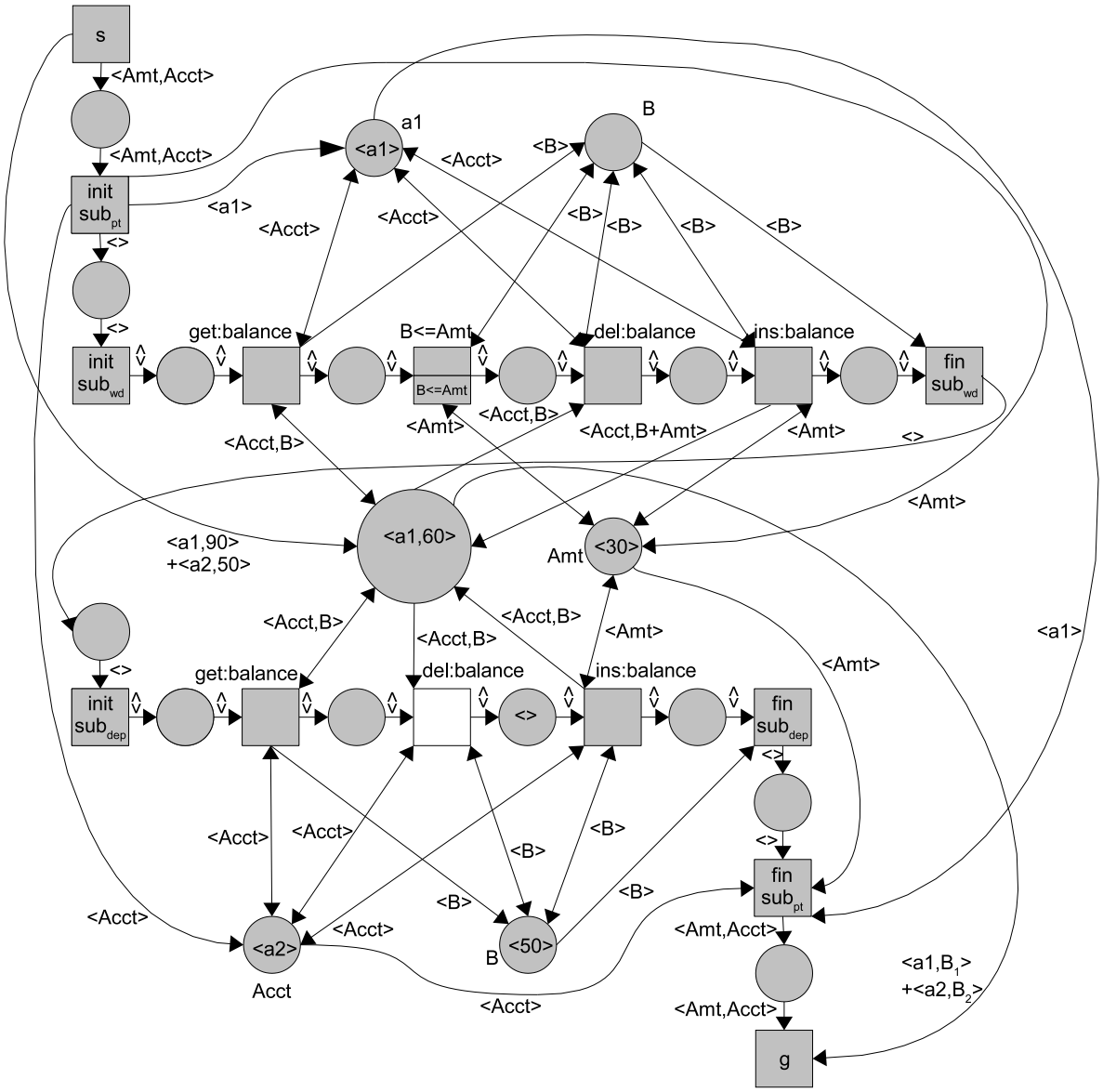
Nun können die Transitionen $initsub_{dep}$,


s
<Amt,Acct>
<Amt,Acct>
init sub_pt
<a1>
a1
B
<Acct>
<B>
<a1>
<Acct>
<Acct>
<B>
<B>
<B>
<B>
<>
<>
init sub_wd
get:balance
B<=Amt
del:balance
ins:balance
fin sub_wd
B<=Amt
<Acct,B>
<Amt>
<Acct,B+Amt>
<Amt>
<>
<Acct,B>
<Amt>
<a1,90>
+<a2,50>
Amt
<30>
<Acct,B>
<Acct,B>
<Acct,B>
<Amt>
<a1>
<Amt>
<>
init sub_dep
<>
get:balance
del:balance
ins:balance
fin sub_dep
<>
<Acct>
<Acct>
<B>
<B>
<>
fin sub_pt
<Acct>
<B>
<B>
<Acct>
<a2>
Acct
<Acct>
B
<Amt,Acct>
<a1,B1>
+<a2,B2>
<Acct>
<Amt,Acct>
g

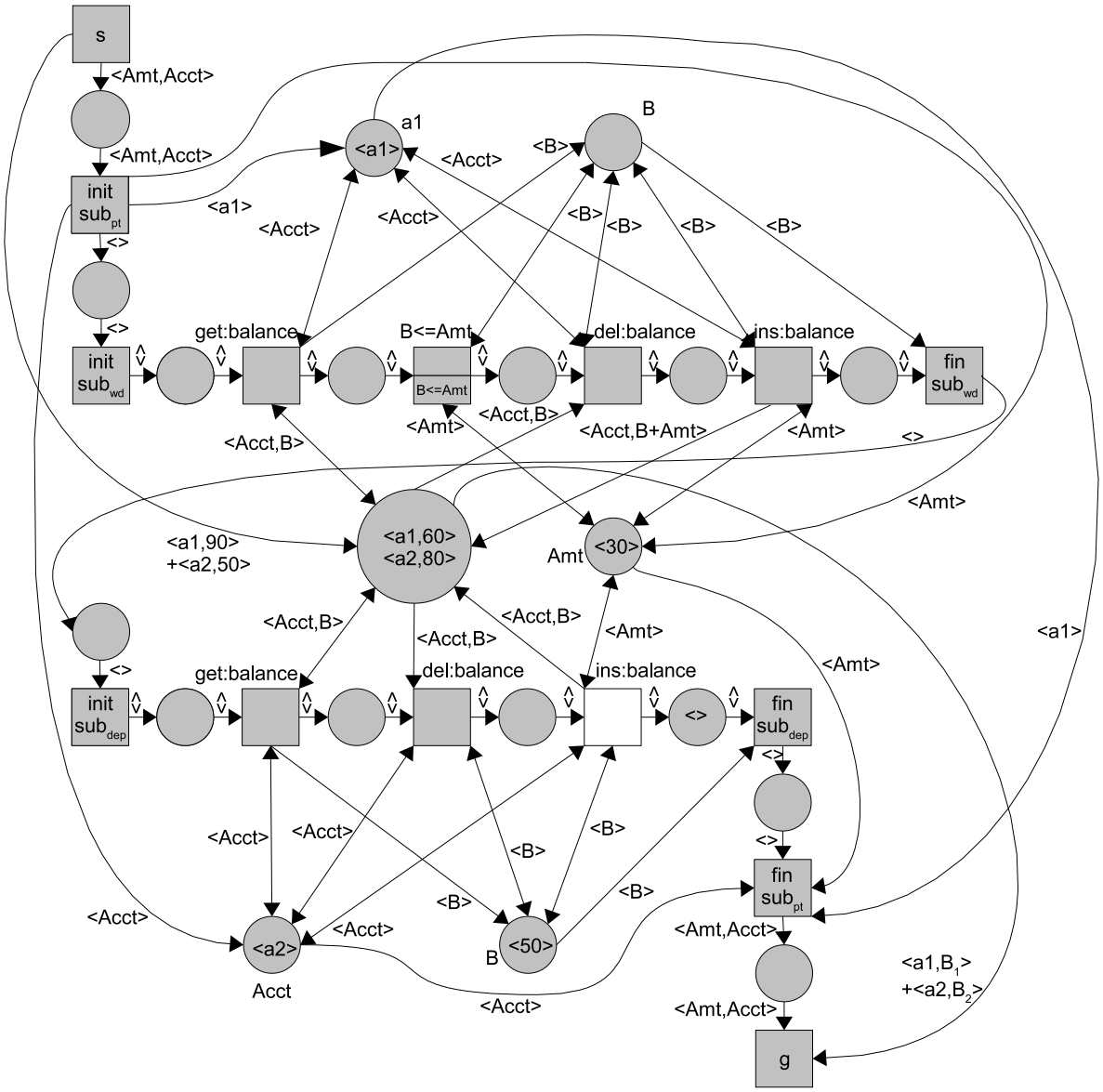
get:balance,


s
<Amt,Acct>
<Amt,Acct>
init sub_pt
<a1>
a1
B
<Acct>
<B>
<a1>
<Acct>
<Acct>
<B>
<B>
<B>
<B>
<>
<>
init sub_wd
get:balance
B<=Amt
del:balance
ins:balance
fin sub_wd
B<=Amt
<Acct,B>
<Amt>
<Acct,B+Amt>
<Amt>
<>
<Acct,B>
<Amt>
<a1,90>
+<a2,50>
<a1,60>
<a2,50>
Amt
<30>
<Acct,B>
<Acct,B>
<Acct,B>
<Amt>
<a1>
<Amt>
<>
init sub_dep
get:balance
del:balance
ins:balance
fin sub_dep
<>
<>
<Acct>
<Acct>
<B>
<B>
<>
fin sub_pt
<B>
<B>
<Acct>
<Acct>
<a2>
Acct
<Acct>
B
<50>
<Amt,Acct>
<a1,B_1>
+<a2,B_2>
<Acct>
<Amt,Acct>
g

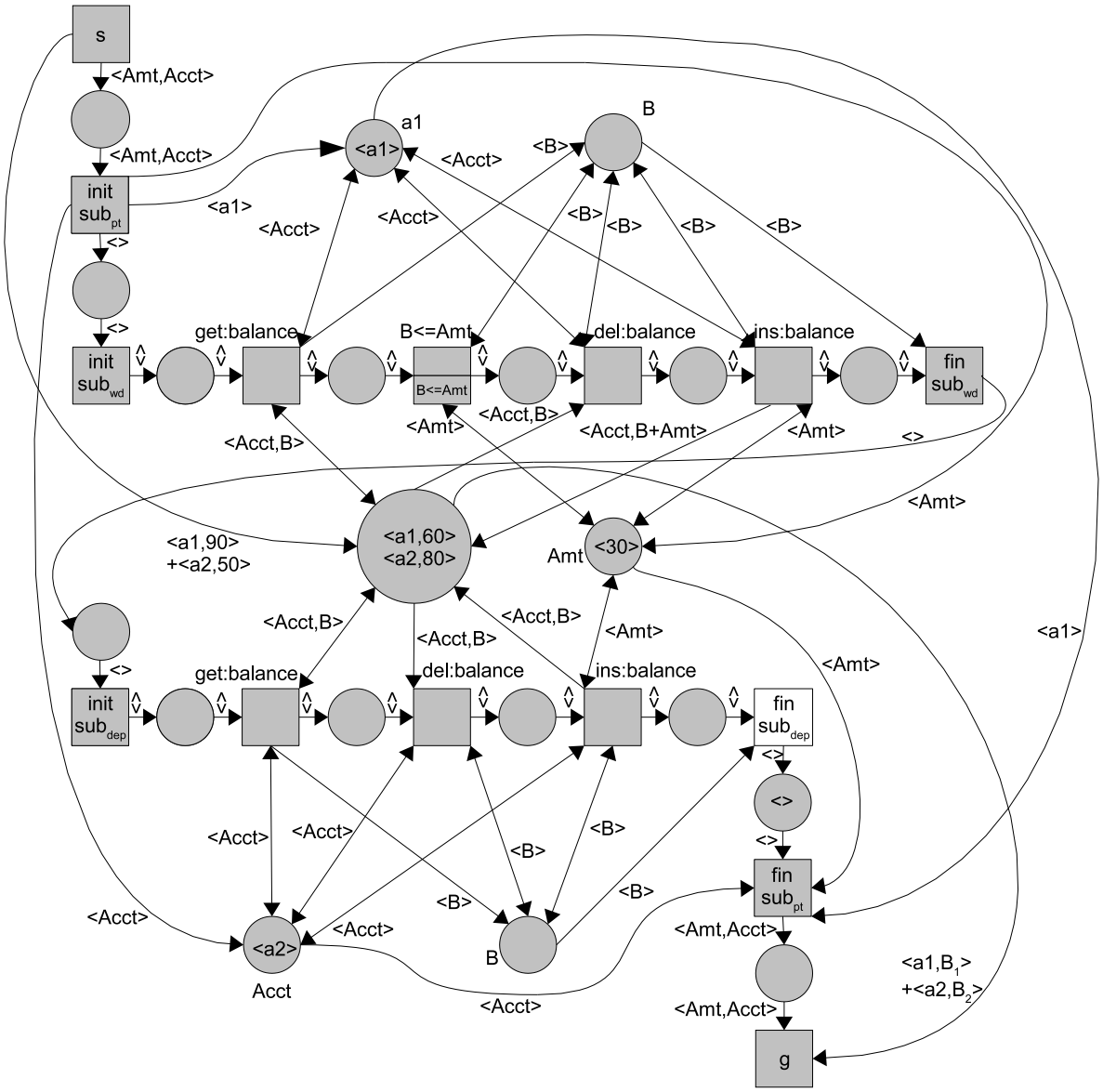
del:balance,


s
<Amt,Acct>
<Amt,Acct>
init sub_pt
<a1>
a1
B
<Acct>
<B>
<a1>
<Acct>
<Acct>
<B>
<B>
<B>
<B>
<>
<>
init sub_wd
get:balance
B<=Amt
del:balance
ins:balance
fin sub_wd
B<=Amt
<Acct,B>
<Amt>
<Acct,B+Amt>
<Amt>
<>
<Acct,B>
<Amt>
<a1,90>
+<a2,50>
<a1,60>
Amt
<30>
<Acct,B>
<Acct,B>
<Acct,B>
<Amt>
<a1>
<Amt>
<>
init sub_dep
get:balance
del:balance
<>
ins:balance
fin sub_dep
<>
<Acct>
<Acct>
<B>
<B>
<>
<B>
<B>
fin sub_pt
<Acct>
<a2>
<Acct>
B
<50>
<Amt,Acct>
<a1,B_1>
+<a2,B_2>
Acct
<Acct>
<Amt,Acct>
g

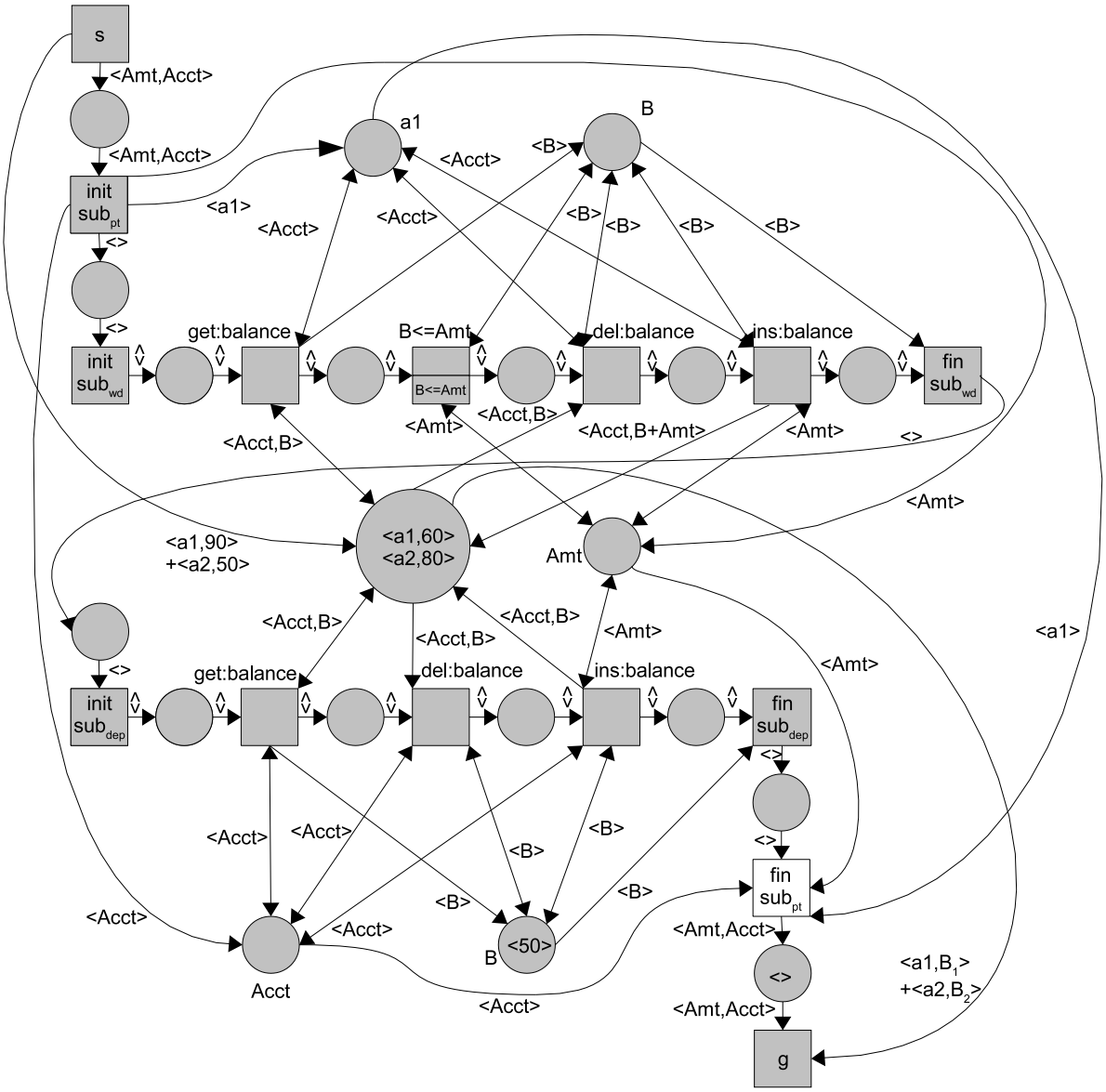
ins:balance



und $finsub_{dep}$ feuern:



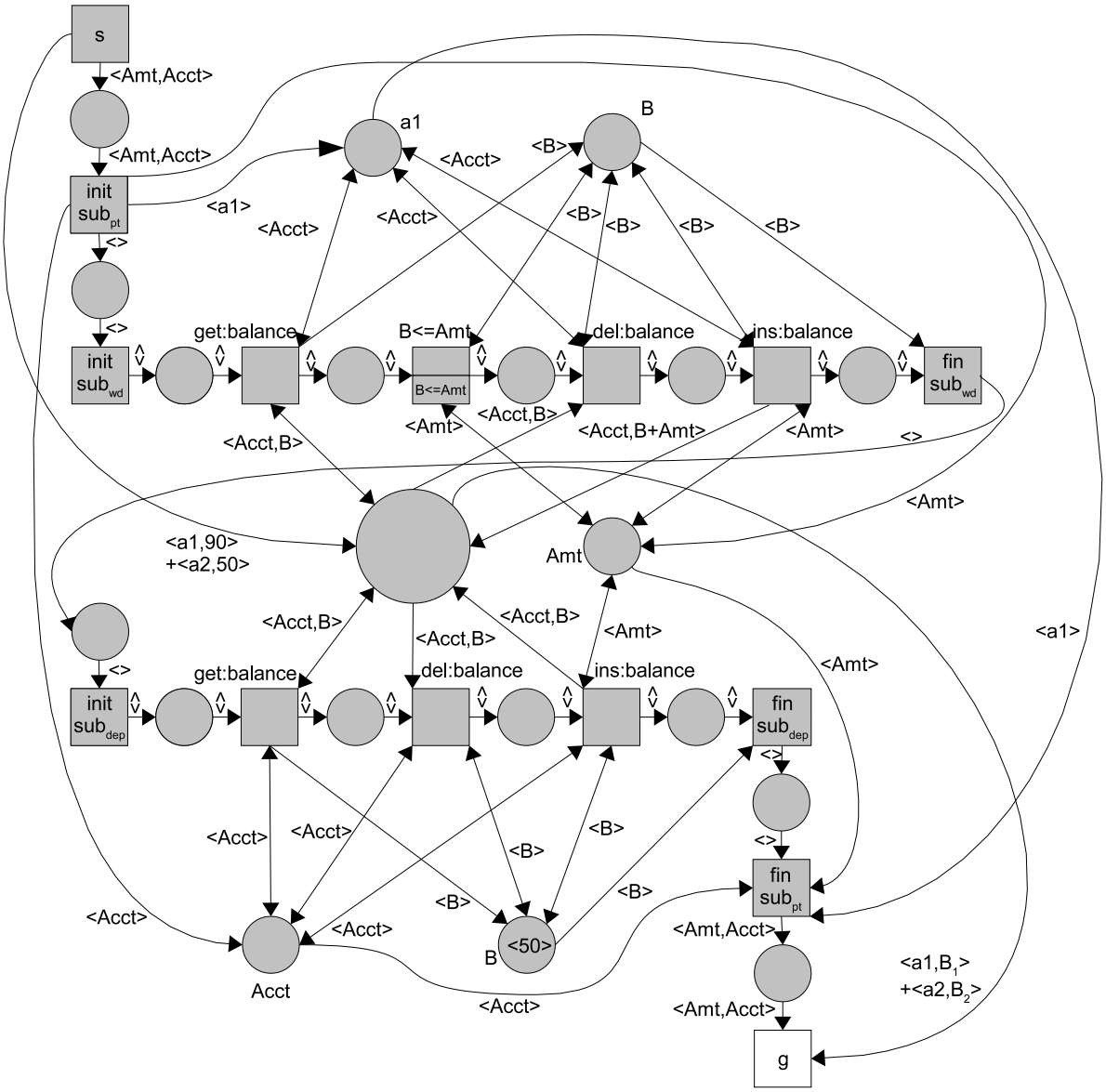
Nach Feuern der Transition $finsub_{pt}$



ist das Netz in einem zu

$$(B, P, \{balance(a1, 40), balance(a2, 80)\}) \vdash ()$$

äquivalenten Zustand. Durch das abschließende Feuern der Transition g



ist das Netz wieder leer und es wurde auch im Netz bewiesen, dass

$$(B, P, \{balance(a1, 90)balance(a2, 50)\}) \vdash payTo(30, a2)$$

gilt.

□

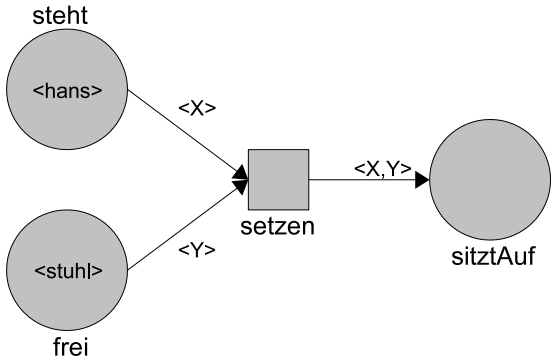
4.3.3 Transformation von Workflow-Netzen in die Transaktionslogik

Der umgekehrte Weg, nämlich ein Workflow-Netz $\mathcal{N} = (P,\ T,\ F, L')$[6] als $\mathcal{TR}$-Programm $Prog_{\mathcal{N}}(\mathcal{B}_{\mathcal{N}}, P_{\mathcal{N}}, D_{\mathcal{N}})$ zu implementieren, gestaltet sich erheblich einfacher. Für jede Transition $t \in T$ des Netzes wird eine elementare Transition $\langle D_1, D_2\rangle, t$ in die Transitionsbasis $\mathcal{B}_{\mathcal{N}}$ eingefügt. Ist ${}^\bullet t = \{p_{i_1}, \ldots, p_{i_n}\}$ und ist tu_{i_j} das Kantenlabel an der Kante von p_{i_j} nach t $\forall j \in \{1, \ldots, n\}$, so ist $D_1 = D \cup \{p_{i_1}(tu_{i_1}), \ldots, p_{i_n}(tu_{i_n})\}$. Und ist

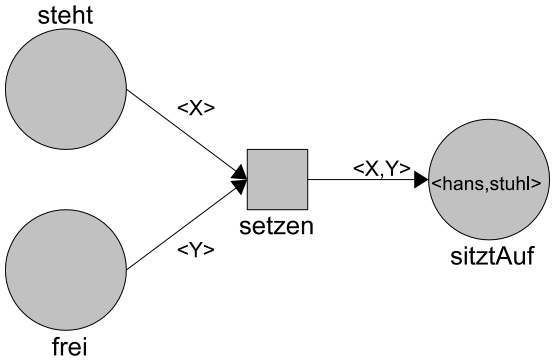
[6] vgl. L' in Abschnitt 4.2.3

$t^\bullet = \{p_{o_1}, \ldots, p_{o_m}\}$ und ist tu_{o_k} das Kantenlabel an Kante von p_{o_k} nach $t\ \forall k \in \{1, \ldots, m\}$, so ist $D_2 = D \cup \{p_{o_1}(tu_{o_1}), \ldots, p_{o_m}(tu_{o_m})\}$. Die Tupel an den Eingangskanten der Transition t werden also zu Daten, die durch Ausführen der elementaren Transition t aus der Datenbank entfernt werden, und die Tupel an den Ausgangskanten werden zu Daten, die in die Datenbank eingefügt werden. Ein Beispiel soll dies verdeutlichen.

Beispiel 12 Hierzu wird die folgende Beispieltransition *setzen* betrachtet.



Zunächst ist also $\langle D_1, D_2 \rangle setzen$ die zu erstellende elementare Transition. Durch Feuern der Transition *setzen* wird von der Stelle *frei* das Tupel $< stuhl >$ und von der Stelle *steht* das Tupel $< hans >$ entfernt. Sind keine entsprechenden Tupel auf der jeweiligen Stelle vorhanden, kann die Transition nicht feuern. Genauso müssen zum Ausführen der hierzu erstellten elementaren Transition die Prädikate $frei(Y)$ und $steht(X)$ in der Datenbank vorhanden sein. Also ist $D_1 = D \cup \{frei(Y), steht(X)\}$. Da das Entfernen der Tupel durch Entfernen der entsprechenden Daten aus der Datenbasis umgesetzt wird, kommen diese beiden Daten nach der Ausführung von *setzen* in der Datenbasis nicht mehr vor. Allerdings legt diese Beispieltransition hier ein Tupel $< hans, stuhl >$ auf die Stelle *sitztAuf*.



Dem entspricht in der elementaren Transition das Einfügen des Datums *sitztAuf*(*hans*, *stuhl*) in die Datenbank, sodass $D_2 = D \cup \{sitztAuf(Y, X)\}$ ist. Somit ergibt sich für diese elementare Transition

$$\langle D \cup \{frei(Y), steht(X)\}, D \cup \{sitztAuf(X, Y)\} \rangle setzen(X, Y).$$

□

Auf Basis dieser Transitionsbasis und Datenbasis ist es jetzt möglich, die Transaktionsbasis $P_\mathcal{N}$, die nur aus einer einzigen Transaktion besteht, wie folgt zu konstruieren. Sei $tr_\mathcal{N}$ diese Transaktion und somit $tr_\mathcal{N} \leftarrow \alpha$ die zu konstruierende Regel. Der Rumpf

der Regel besteht aus einem rekursiven Aufruf der Transaktion $tr_{\mathcal{N}}$, nachdem eine oder mehrere der elementaren Transitionen $s, g, t_1, \ldots, t_n$ ausgeführt wurden und dem durch ein ODER abgetrennten Rekursionsabruch ϵ:

$$\begin{aligned} P'_{\mathcal{N}} =& \{(tr_{\mathcal{N}} \leftarrow ((s(X_{s_1}, \ldots, X_{s_{m_s}}) \vee g(X_{g_1}, \ldots, X_{g_{m_g}}) \vee t_1(X_{t_{1_1}}, \ldots, X_{t_{1_{m_{t_1}}}}) \\ & \vee \cdots \vee t_n(X_{t_{n_1}}, \ldots, X_{t_{n_{m_{t_n}}}})) \otimes tr_{\mathcal{N}}) \vee \epsilon)\} \end{aligned}$$

Da in Workflow-Netzen nach Def. 17 und 19 immer s als erste und g als letze Transition feuern müssen, wird nicht diese, sondern die folgende Implementierung von $P_{\mathcal{N}}$ verwendet, die auch in der $\mathcal{TR}$-Implementierung sicherstellt, dass zuerst s und zuletzt g abgeleitet werden:

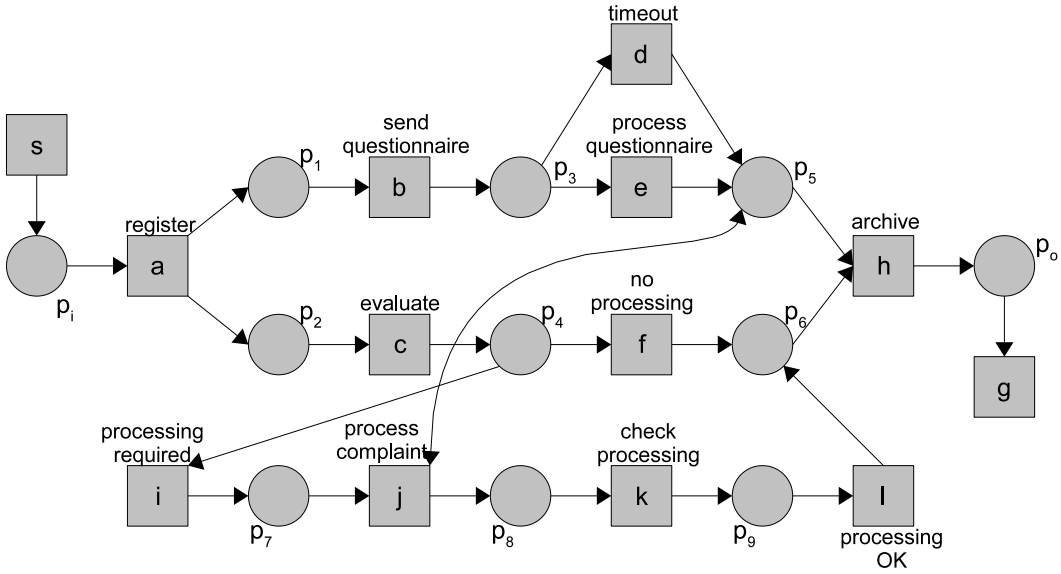
$$\begin{aligned} P_{\mathcal{N}} = \{&((\mathcal{N} \leftarrow ((s(X_{s_1}, \ldots, X_{s_{m_s}}) \otimes tr_{\mathcal{N}} \otimes g(X_{g_1}, \ldots, X_{g_{m_g}})), \\ & (tr_{\mathcal{N}} \leftarrow ((((t_1(X_{t_{1_1}}, \ldots, X_{t_{1_{m_{t_1}}}}) \vee \cdots \vee t_n(X_{t_{n_1}}, \ldots, X_{t_{n_{m_{t_n}}}})) \otimes tr_{\mathcal{N}}) \vee \epsilon))\} \end{aligned}$$

Welche der elementaren Transitionen aktuell ausgeführt werden können, wird hier allein durch die aktuelle Datenbasis bestimmt. Genau wie die aktuell feuerbaren Transitionen in $\mathcal{N}$ durch die aktuelle Markierung bestimmt werden. Daraus ergibt sich, dass (genau wie in $\mathcal{N}$) immer zuerst s und zuletzt g ausgeführt werden.

Wird ein Workflow-Netz auf diese Art und Weise als $\mathcal{TR}$-Programm implementiert, entspricht das Reproduzieren der leeren Datenbasis dem Reproduzieren der leeren Markierung.

Das folgende Beispiel soll helfen, das soeben Formulierte besser zu verstehen.

Beispiel 13 Hierzu wird noch einmal das von S. 4 bekannte Beispiel eines Reklamationsprozesses aufgegriffen. Zur besseren Übersicht wird das daraus erstellte p/t-Netz von S. 26 hier noch einmal dargestellt.



Aus diesem p/t-Netz mit $\mathcal{N} = (P, T, F)$ ergibt sich

- $P = \{p_i, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_o\}$
- $T = \{s, a, b, c, d, e, f, h, i, j, k, l, g\}$

- $F = \{(s, p_i), (p_i, a), (a, p_1), (p_1, b), (b, p_3), (p_3, d), (p_3, e),$
  $(d, p_5), (e, p_5), (p_5, h), (a, p_2), (p_2, c), (c, p_4), (p_4, f), (f, p_6),$
  $(p_6, h), (p_4, i), (i, p_7), (p_7, j), (j, p_8), (p_8, k), (k, p_9), (p_9, l),$
  $(l, p_6), (p_5, j), (j, p_5), (h, p_o), (p_o, g)\}$

Um dieses Netz als $\mathcal{TR}$-Programm $Prog_{\mathcal{N}} = (\mathcal{B}_{\mathcal{N}}, P_{\mathcal{N}}, D_{\mathcal{N}})$ zu implementieren, müssen $\mathcal{B}_{\mathcal{N}}$, $P_{\mathcal{N}}$ und $D_{\mathcal{N}}$ bestimmt werden. Beginnend mit $\mathcal{B}_{\mathcal{N}}$ wird für jede Transition $t \in T$ eine elementare Transition erstellt:

- $\langle D_{s_1}, D_{s_2} \rangle s$
- $\langle D_{a_1}, D_{a_2} \rangle a$
- $\langle D_{b_1}, D_{b_2} \rangle b$
- $\langle D_{c_1}, D_{c_2} \rangle c$
- $\langle D_{d_1}, D_{d_2} \rangle d$
- $\langle D_{e_1}, D_{e_2} \rangle e$
- $\langle D_{f_1}, D_{f_2} \rangle f$
- $\langle D_{h_1}, D_{h_2} \rangle h$
- $\langle D_{i_1}, D_{i_2} \rangle i$
- $\langle D_{j_1}, D_{j_2} \rangle j$
- $\langle D_{k_1}, D_{k_2} \rangle k$
- $\langle D_{l_1}, D_{l_2} \rangle l$
- $\langle D_{g_1}, D_{g_2} \rangle g$

Um die einzelnen D_{t_m} mit $m \in \{1, 2\}$ zu bestimmen, wird der Vorbereich und der Nachbereich der Transition t betrachtet. Zum Beispiel ist ${}^\bullet s = \{\}$ und $s^\bullet = \{p_i\}$ und somit ist $D_{s_1} = D$ und $D_{s_2} = D \cup \{p_i\}$, sodass sich für diese elementare Transition $\langle D, D \cup \{p_i\} \rangle s$ ergibt. Insgesamt erhält man so

$$\begin{aligned}\mathcal{B}_{\mathcal{N}} = \{&(\langle D, D \cup \{p_i\}\rangle s), (\langle D \cup \{p_i\}, D \cup \{p_1, p_2\}\rangle a), (\langle D \cup \{p_1\}, D \cup \{p_3\}\rangle b),\\ &(\langle D \cup \{p_2\}, D \cup \{p_4\}\rangle c), (\langle D \cup \{p_3\}, D \cup \{p_5\}\rangle d),\\ &(\langle D \cup \{p_3\}, D \cup \{p_5\}\rangle e), (\langle D \cup \{p_4\}, D \cup \{p_6\}\rangle f),\\ &(\langle D \cup \{p_5, p_6\}, D \cup \{p_o\}\rangle h), (\langle D \cup \{p_4\}, D \cup \{p_7\}\rangle i),\\ &(\langle D \cup \{p_5, p_7\}, D \cup \{p_5, p_8\}\rangle j), (\langle D \cup \{p_8\}, D \cup \{p_9\}\rangle k),\\ &(\langle D \cup \{p_9\}, D \cup \{p_6\}\rangle l), (\langle D \cup \{p_o\}, D \cup \{\}\rangle g)\}.\end{aligned}$$

Da $\mathcal{N}$ zu Beginn leer ist (d. h. $M_0 = \mathbf{0}$), ist auch $D_{\mathcal{N}}$ zunächst leer:

$$D_{\mathcal{N}_0} = \{\}$$

Die Transaktionsbasis $P_{\mathcal{N}}$ ergibt sich nach obiger Beschreibung als

$$\begin{aligned}P_{\mathcal{N}} = \quad \{ \quad &(\mathcal{N} \leftarrow (s \otimes tr_{\mathcal{N}} \otimes g)),\\ &(tr_{\mathcal{N}} \leftarrow (((a \vee b \vee c \vee d \vee e \vee f \vee g \vee h \vee i \vee j \vee k \vee l) \otimes tr_{\mathcal{N}}) \vee \epsilon))\}.\end{aligned}$$

□

Diese Art der Übertragung ist (auch wenn sie ihren Zweck erfüllt) sehr primitiv. Sie ist zudem insofern nicht zufriedenstellend, als dass sie die Transaktionsbasis fast vollkommen unberücksichtigt lässt. Im Laufe des folgenden Kapitels wird eine Methode vorgestellt, mit deren Hilfe man dieses Manko für die $\mathcal{TR}$-Implementierung beheben kann.

4.4 Aktionslogik

Die Aktionslogik ($\mathcal{LA}$) wurde in [Fide93] auf Basis von [Genr73] eingeführt. Sie wurde in [Laut97] modifiziert und schließlich in [Simo01] um den Zeitbegriff erweitert. Sie ist im Gegensatz zur Prädikatenlogik und Transaktionslogik keine Erweiterung der Aussagenlogik. Mit der Transaktionslogik hat sie gemein, dass sie zur Beschreibung und Überprüfung dynamischer Vorgänge dient. Solch ein Vorgang, im folgenden Prozess genannt, kann auch ein Computerprogramm sein. In [Fide93] wurde gezeigt, wie für ein solches Programm überprüft werden kann, ob es korrekt, d. h. der Spezifikation entsprechend, implementiert wurde. Dies wurde mit Hilfe von p/t-Netzen durchgeführt, sodass dort auch bereits gezeigt wurde, wie ein $\mathcal{LA}$-Modul (eine $\mathcal{LA}$-Formel) als p/t-Netz zu implementieren ist. Nach der Einführung zentraler Begriffe im folgenden Abschnitt wird eine solche Implementierung nach [Simo01] exemplarisch durchgeführt, bevor anschließend der umgekehrte Weg gezeigt wird. Im letzten Abschnitt wird dieser Weg dann auf die $\mathcal{TR}$ übertragen.

4.4.1 Definitionen und wichtige Begriffe der Aktionslogik

Auch für die $\mathcal{LA}$ wird als Erstes die Menge der Zeichen, das Alphabet, bestimmt.

Definition 56 Das *Alphabet der* $\mathcal{LA}$ besteht aus

- einer endlichen Menge von Buchstaben $\mathcal{A} = \{a_1, a_2, b_1, c, \dots\}$,
- den Konnektoren von Prozessen
  - $\bar{}$ (NEGATION),
  - $\oslash$ (serielles VOR),
  - $\oslash$ (serielles NACH) und
  - $\ominus$ (KOINZIDENZ),
- den Konnektoren von Modulen
  - $\bar{}$ (NEGATION),
  - $\boxminus$ (serielles VOR),
  - $\boxminus$ (serielles NACH),
  - $\boxminus$ (KOINZIDENZ),
  - $\boxwedge$ (logisches UND),
  - $\boxvee$ (logisches ODER) und
  - $\boxplus$ (logisches XODER),
- den Konstantensymbolen
  - $\bot$ (LEERES WORT),
  - $\top$ (Tautologie) und
  - $\top^*$ (iterative Tautologie) und
- den Hilfssymbolen '(', ')', '[', und ']'.

[Fide93, Laut97, Simo01]

Über diesem Alphabet werden zunächst die Wörter, die über diesem Alphabet gebildet werden können definiert.

Definition 57 Sei $\mathcal{A}$ eine Menge von Buchstaben. Die Menge der *synchronen Wörter* $S_\mathcal{A}$ über $\mathcal{A}$ ist die kleinste Menge mit

1. $(a) \in S_A$ *und* $\overline{a} \in S_\mathcal{A}$ gdw. $a \in A$ und
2. $(W_1 \circledcirc W_2) \in S_\mathcal{A}$ gdw. $W_1, W_2 \in S_A$.

Die Menge *Wörter* $W_\mathcal{A}$ über $\mathcal{A}$ ist die kleinste Menge mit

- $(\bot) \in W_\mathcal{A}$,
- $S_A \subseteq W_\mathcal{A}$ und
- $(W_1, W_2 \in W_\mathcal{A} - \{(\bot)\}) \Rightarrow (W_1 \circledleft W_2) \in W_\mathcal{A}$ und $(W_1 \circledright W_2) \in W_\mathcal{A}$.

[Fide93, Laut97, Simo01]

Zu diesen Wörtern wird auch hier wieder die Menge der Atome bestimmt. Außerdem wird die Menge der Literale eines Worts definiert, die im Gegensatz zur Atommenge zwischen negierten und nicht negierten Atomen unterscheidet.

Definition 58 Sei $\mathcal{A}$ ein Alphabet und $W \in W_\mathcal{A}$ ein Wort über $\mathcal{A}$. Die $W_\mathcal{A}$ $\mathbb{A}(W)$ von W ist

- $\mathbb{A}(W) := \emptyset$ gdw. $W = (\bot)$,
- $\mathbb{A}(W) := \{a\}$ gdw. für $a \in \mathcal{A} : W = (a)$, oder $W = \overline{a}$ und
- $\mathbb{A}(W) := \mathbb{A}(W_1) \cup \mathbb{A}(W_2)$ gdw. $W = (W_1 \boxdot W_2), W_1, W_2 \in W_\mathcal{A}$ Wörter über $\mathcal{A}$ sind und $\boxdot \in \{\circledleft, \circledright, \circledcirc\}$.

Die *Menge der Literale* von S ($L(W)$) ist

- $L(W) := \emptyset$ gdw. $W = \bot$,
- $L(W) := \{a\}$ gdw. für $a \in \mathcal{A} : W = (a)$,
- $L(W) := \{\overline{a}\}$ gdw. for $a \in \mathcal{A} : W = (\overline{a})$ und
- $L(W) := L(W_1) \cup L(W_2)$ gdw. $W = (W_1 \boxdot W_2), W_1, W_2 \in W_\mathcal{A}$ Wörter über $\mathcal{A}$ sind und $\boxdot \in \{\circledleft, \circledright, \circledcirc\}$.

[Fide93, Laut97, Simo01]

Prozesse werden nun als spezielle Wörter definiert.

Definition 59 Sei $\mathcal{A}$ ein Alphabet. Die *Menge der Prozesse* $P_\mathcal{A}$ über $\mathcal{A}$ ist die kleinste Menge mit

- $(\bot) \in P_\mathcal{A}$, [7]

[7] $(\bot)$ bezeichnet den leeren Prozess und wird manchmal gemäß der Aussagenlogik *Kontradiktion* oder *Falsum* genannt.

- $(a) \in P_{\mathcal{A}}$ gdw. $a \in \mathcal{A}$,
- $(\overline{a}) \in P_{\mathcal{A}}$ gdw. $a \in \mathcal{A}$, [8]
- $(W_1 \ominus W_2) \in P_{\mathcal{A}}$ gdw. $W_1, W_2 \in S_{\mathcal{A}}$ und $\nexists a \in \mathcal{A} : \{a, \overline{a}\} \subseteq L(W_1) \cup L(W_2)$ und
- $(W_1 \oslash W_2) \in P\mathcal{A}$ und $(W_1 \oslash W_2) \in P_A$ gdw. $W_1, W_2 \in P_{\mathcal{A}} - \{(\bot)\}$ und $A(W_1) \cap A(W_2) = \emptyset$.

[Fide93, Laut97, Simo01]

Definition 60 Sei $\mathcal{A}$ ein Menge von Aktionen. Die *Menge der Module* über $\mathcal{A}$ ($\mathcal{M}_{\mathcal{A}}$) ist die kleinste Menge mit

- $[\bot] \in \mathcal{M}_{\mathcal{A}}$ (leeres Modul),
- $a \in A \Rightarrow [a] \in \mathcal{M}_{\mathcal{A}}$,
- $M \in \mathcal{M}_{\mathcal{A}} \Rightarrow [\overline{M}] \in \mathcal{M}_{\mathcal{A}}$ (Negation),
- $M_1, M_2 \in \mathcal{M}_{\mathcal{A}} \Rightarrow [M_1 \boxleft M_2] \in \mathcal{M}_{\mathcal{A}}$ (VOR), $[M_1 \boxright M_2] \in \mathcal{M}_{\mathcal{A}}$ (NACH), und $[M_1 \boxminus M_2] \in \mathcal{M}_{\mathcal{A}}$ (KOINZIDENZ),
- $M_1, M_2 \in \mathcal{M}_{\mathcal{A}} \Rightarrow [M_1 \boxwedge M_2] \in \mathcal{M}_{\mathcal{A}}$ (UND), $[M_1 \boxvee M_2] \in \mathcal{M}_{\mathcal{A}}$ (ODER), und $[M_1 \boxplus M_2] \in \mathcal{M}_{\mathcal{A}}$ (exklusives ODER) und
- $M \in \mathcal{M}_{\mathcal{A}} \Rightarrow [M^n] \in \mathcal{M}_{\mathcal{A}}$ (n-fache Iteration), $[M^+] \in \mathcal{M}_{\mathcal{A}}$ (n-fache Iteration für $n \geq 1$), und $[M^*] \in \mathcal{M}_{\mathcal{A}}$ (n-fache Iteration für $n \geq 0$).

[Fide93, Laut97, Simo01]

$\boxleft$, $\boxright$ und $\boxminus$ heißen *elementare Operatoren*; alle anderen Operatoren heißen *nicht-elementare Operatoren*.

Definition 61 Sei $\mathcal{A}$ ein Alphabet und $M \in \mathcal{M}_{\mathcal{A}}$ ein Modul über $\mathcal{A}$. Die *Aktionsmenge* $A(M)$ in M ist

- $A(M) := \emptyset$ gdw $M = [\bot]$,
- $A(M) := a$ gdw. $a \in \mathcal{A}$ und $M = [a]$,
- $A(M) := A(M_1) \cup A(M_2)$ gdw. $M = (M_1 \boxdot M_2), M_1, M_2 \in \mathcal{M}_{\mathcal{A}}$ Module über $\mathcal{A}$ sind und $\boxdot \in \{\boxleft, \boxright, \boxminus, \boxwedge, \boxplus, \boxvee\}$ und
- $A(M) := A(M_1)$ gdw. $M = \{\overline{M_1}, M = [M_1^n], M = [M_1^+]\}$ oder $M = [M_1^*]$.

Die Menge der Literale in M ($A(M)$) ist

- $L(M) := \emptyset$ gdw $M = [\bot]$,
- $L(M) := a$ gdw. $a \in \mathcal{A}$ und $M = [a]$,

[8] Im weiteren Verlauf werden die Elemente von $\mathcal{A}$ *Aktionen* genannt. Der leere Prozess ($\bot$) wird als Unmöglichkeit von Aktionen interpretiert. (a) und $(\overline{a})$ werden *elementare Prozesse* genannt. In (a) kommt die Aktion a vor, in $(\overline{a})$ ist Aktion a verboten.

- $L(M) := \{a|\overline{a} \in L(M_1)\} \cup \{\overline{a}|a \in L(M_1)\}$ gdw. $M = [\overline{M_1}]$
- $L(M) := L(M_1) \cup L(M_2)$ gdw. $M = (M_1 \boxdot M_2), M_1, M_2 \in \mathcal{M}_\mathcal{A}$ Module über $\mathcal{A}$ sind und $\boxdot \in \{\boxleftarrow, \boxrightarrow, \boxminus, \boxslash, \boxplus, \boxtimes\}$ und
- $L(M) := L(M_1)$ gdw. $M = [M_1^n], M = [M_1^+]\}$ oder $M = [M_1^*]$.

[Fide93, Laut97, Simo01]

Definition 62 Sei $\mathcal{A}$ eine Menge von Aktionen, $a \in \mathcal{A}$ und $M_1, M_2 \in \mathcal{M}_\mathcal{A}$. Die *Prozessmenge* P der Module über $\mathcal{A}$ ist definiert als

- $P[\bot] := \emptyset$,
- $P[a] := \{(a)\}$,
- $P[\overline{a}] := \{(\overline{a})\}$,
- $P[M_1 \boxleftarrow M_2] := \{W|W = (W_1 \circledleftarrow W_2), W \in P_\mathcal{A}, W_1 \in P[M_1], W_2 \in P[M_2]\}$,
- $P[M_1 \boxrightarrow M_2] := \{W|W = (W_1 \circledrightarrow W_2), W \in P_A, W_1 \in P[M_1], W_2 \in P[M_2]\}$ und
- $P[M_1 \boxminus M_2] := \begin{cases} \{(W_1 \circleddash W_2)\} & \text{wenn } (W_1 \circleddash W_2) \in P_A, \\ & W_1 \in P[M_1], W_2 \in P[M_2]\} \\ \emptyset & \text{sonst.} \end{cases}$

[Fide93, Laut97, Simo01]

Definition 63 Seien $M_1, M_2 \in \mathcal{M}_\mathcal{A}$ zwei Module über $\mathcal{A}$. Dann ist

- $P[M_1 \boxplus M_2] := P[M_1] \uplus P[M_2] = (P[M_1] \cup P[M_2]) - (P[M_1] \cap P[M_2])$.

[Fide93, Laut97, Simo01]

Definition 64 Sei $\mathcal{L}$ eine endliche Menge von Literalen. Eine *Abhängigkeitsfunktion* $D : \mathcal{L} \times \mathcal{L} \to \{vo, na, gl\}$ beschreibt alle kausalen Abhängigkeiten aller Paare (l_1, l_2) mit $l_1, l_2 \in \mathcal{L}$ als:

$$D(l_1, l_2) := \begin{cases} vo & \text{gdw. } l_1 \textbf{ vor } l_2 \text{ vorkommt oder verboten ist,} \\ na & \text{gdw. } l_1 \textbf{ nach } l_2 \text{ vorkommt oder verboten ist und} \\ gl & \text{gdw. } l_1 \text{ und } l_2 \textbf{ gleichzeitig} \text{ vorkommen oder verboten sind.} \end{cases}$$

[Fide93, Laut97, Simo01]

Um zwei verschiedene Module miteinander vergleichbar zu machen, müssen diese jeweils um die Atome des anderen erweitert werden. Vor dieser gegenseitigen Komplettierung zweier Module wird noch die Komplettierung eines einzelnen Moduls definiert. Nach einer solchen Komplettierung ist in einem Modul jedes Atom sowohl negiert als auch nicht negiert vorhanden.

Definition 65 Komplettierung von Modulen Sei $M \in \mathcal{M}_{\mathcal{A}}$ ein Modul über $\mathcal{A}$, W ein Prozess aus M und $L \subseteq A \cup \{\overline{a} | a \in A\}$ wobei $L(W) \cap = \emptyset$. Das *opposite* der Literale aus M, die nicht in W vorkommen ist definiert durch

$$opp(W, M) := \{\overline{a} | a \in L(M), a \notin A(W)\} \cup \{a | \overline{a} \in L(M), a \notin A(W)\}.$$

$$merge(W, L) := \{W' | \quad W' \text{ist ein Prozess mit Abhängigkeitsfunktion} D_{W'} \text{ und} \forall l_1, l_2 \in L(W') = L(W) \cup L :$$

$$D_{W'}(l_1, l_2) \begin{cases} = D_W(l_1, l_2) & \text{wenn } l_1, l_2 \in L(W) \\ \in \{vo, na, gl\} & \text{sonst} \end{cases}$$

$$\}$$

Die Prozessmenge der *Komplettierung* $C[M]$ von M ist

$$P[C[M]] := \bigcup_{W \in P[M]} merge(W, opp(W, M)).$$

[Fide93, Laut97, Simo01]

Definition 66 Seien $M_1, M_2 \in \mathcal{M}_{\mathcal{A}}$ Module über $\mathcal{A}$ und $\{a_1, \ldots, a_k\} := A(M_2) - A(M_1)$. Die *gegenseitige Komplettierung* von M_1 relativ zu M_2 ($C_{M_2}[M_1]$) ist

$$C_{M_2}[M_1] := \begin{cases} [M_1 \boxtimes [[a_1 \boxplus \overline{a_1}] \boxtimes \ldots \boxtimes [a_k \boxtimes \overline{a_k}]]] & \text{wenn } A(M_2) - A(M_1) \neq \emptyset, \\ C[M_1] & \text{sonst.} \end{cases}$$

[Fide93, Laut97, Simo01]

Die gegenseitige Komplettierung erlaubt die Prozessmengen zweier Module zu bestimmen, die durch $\boxtimes$ bzw. $\boxdot$ miteinander verbunden sind.

Definition 67 Seien $M_1, M_2 \in \mathcal{M}_{\mathcal{A}}$ zwei Module über $\mathcal{A}$. Dann ist

- $P[M_1 \boxtimes M_2] := P[C_{M_2}[M_1]] \cup P[C_{M_1}[M_2]]$ und
- $P[M_1 \boxdot M_2] := P[M_1] \cup P[M_2] \cup P[M_1 \boxtimes M_2]$.

[Fide93, Laut97, Simo01]

Wie schon gesagt, werden durch die gegenseitig Komplettierung zwei Module miteinander vergleichbar gemacht. Dies ist notwendig, um überprüfen zu können, ob ein Modul eine ebenfalls als Modul gegebene Spezifikation erfüllt oder nicht.

Definition 68 Seien $M_1, M_2 \in \mathcal{M}_{\mathcal{A}}$ zwei Module über $\mathcal{A}$.

- M_1 *erfüllt* M_2 ($M_1 \twoheadrightarrow M_2$) *gdw.* $P[C_{M_2}]M_1]] \subseteq P[C_{M_1}[M_2]]$.
- M_1 *widerspricht* M_2 ($M_1 \rightsquigarrow M_2$) *gdw.* $P[C_{M_2}]M_1]] \cap P[C_{M_1}[M_2]] = \emptyset$.

[Fide93, Laut97, Simo01]

Damit sind alle für diese Arbeit wichtigen Begriffe der $\mathcal{LA}$ eingeführt. Bevor im nächsten Abschnitt gezeigt wird, wie ein $\mathcal{LA}$-Modul als p/t-Netz implementiert wird, soll an einem Beispiel demonstriert werden, wie überprüft wird, ob eine Realisierung eine Spezifikation erfüllt.

Beispiel 14 Unter Verwendung des Moduls $R = [a \boxleft [[b \boxplus c] \boxright d]]$ soll geprüft werden, ob diese Realisierung die Spezifikation $S = [[b \boxright c] \boxleft d]$ erfüllt. Dazu werden zunächst die beiden Module R und S gegenseitig komplettiert, wozu wiederum noch zuerst die Aktionsmengen von R und S bestimmt werden müssen:

$$\begin{aligned} A(R) =& \{a, b, c, d\} \\ A(S) =& \{b, c, d\} \end{aligned}$$

Somit ist $C_S[R] = C[R]$, da $\{b, c, d\} - \{a, b, c, d\} = \emptyset$ und $C_R[S] = [[[b \boxright c] \boxleft d] \boxleft [a \boxplus \overline{a}]]$ da $\{a, b, c, d\} - \{b, c, d\} = \{a\}$. Jetzt werden diese beiden Prozessmengen berechnet.

$$\begin{aligned} P[C_S[R]] = \{&(a \circledcirc b \circledcirc \overline{c} \circledcirc \overline{d}), (a \circledcirc \overline{c} \circledcirc b \circledcirc \overline{d}), (\overline{c} \circledcirc a \circledcirc b \circledcirc \overline{d}), (\overline{c} \circledcirc a \circledcirc \overline{d} \circledcirc b), (a \circledcirc b \circledcirc \overline{d} \circledcirc \overline{c}), \\ &(a \circledcirc \overline{d} \circledcirc b \circledcirc \overline{c}), (\overline{d} \circledcirc a \circledcirc b \circledcirc \overline{c}), (\overline{d} \circledcirc a \circledcirc \overline{c} \circledcirc b), (a \circledcirc c \circledcirc \overline{b} \circledcirc \overline{d}), (a \circledcirc \overline{b} \circledcirc c \circledcirc \overline{d}), (\overline{b} \circledcirc a \circledcirc c \circledcirc \overline{d}), \\ &(\overline{b} \circledcirc a \circledcirc \overline{d} \circledcirc c), (a \circledcirc c \circledcirc \overline{d} \circledcirc \overline{b}), (a \circledcirc \overline{d} \circledcirc c \circledcirc \overline{b}), (\overline{d} \circledcirc a \circledcirc c \circledcirc \overline{b}), (\overline{d} \circledcirc a \circledcirc \overline{b} \circledcirc c), (a \circledcirc d \circledcirc \overline{b} \circledcirc \overline{c}), \\ &(a \circledcirc \overline{b} \circledcirc d \circledcirc \overline{c}), (\overline{b} \circledcirc a \circledcirc d \circledcirc \overline{c}), (\overline{b} \circledcirc a \circledcirc \overline{c} \circledcirc d), (a \circledcirc d \circledcirc \overline{c} \circledcirc \overline{b}), (a \circledcirc \overline{c} \circledcirc d \circledcirc \overline{b}), (\overline{c} \circledcirc a \circledcirc d \circledcirc \overline{b}), \\ &(\overline{c} \circledcirc a \circledcirc \overline{b} \circledcirc d), (a \circledcirc b \circledcirc d \circledcirc \overline{c}), (a \circledcirc b \circledcirc \overline{c} \circledcirc d), (a \circledcirc \overline{c} \circledcirc b \circledcirc d), (\overline{c} \circledcirc a \circledcirc b \circledcirc d), (a \circledcirc d \circledcirc b \circledcirc \overline{c}), \\ &(a \circledcirc d \circledcirc \overline{c} \circledcirc b), (a \circledcirc \overline{c} \circledcirc d \circledcirc b), (\overline{c} \circledcirc a \circledcirc d \circledcirc b), (a \circledcirc c \circledcirc d \circledcirc \overline{b}), (a \circledcirc c \circledcirc \overline{b} \circledcirc d), (a \circledcirc \overline{b} \circledcirc c \circledcirc d), \\ &(\overline{b} \circledcirc a \circledcirc c \circledcirc d), (a \circledcirc d \circledcirc c \circledcirc \overline{b}), (a \circledcirc d \circledcirc \overline{b} \circledcirc c), (a \circledcirc \overline{b} \circledcirc d \circledcirc c), (\overline{b} \circledcirc a \circledcirc d \circledcirc c)\} \end{aligned}$$

$P[C_R[S]] = \{$
$(b \circledcirc d \circledcirc \overline{a} \circledcirc \overline{c}), (b \circledcirc \overline{a} \circledcirc d \circledcirc \overline{c}), (\overline{a} \circledcirc b \circledcirc d \circledcirc \overline{c}), (\overline{a} \circledcirc b \circledcirc \overline{c} \circledcirc d), (\overline{a} \circledcirc \overline{c} \circledcirc b \circledcirc d), (d \circledcirc \overline{a} \circledcirc \overline{c} \circledcirc d),$
$(b \circledcirc d \circledcirc \overline{c} \circledcirc \overline{a}), (b \circledcirc \overline{c} \circledcirc d \circledcirc \overline{a}), (\overline{c} \circledcirc b \circledcirc d \circledcirc \overline{a}), (\overline{c} \circledcirc b \circledcirc \overline{a} \circledcirc d), (\overline{c} \circledcirc \overline{a} \circledcirc b \circledcirc d), (d \circledcirc \overline{c} \circledcirc \overline{a} \circledcirc d),$
$(b \circledcirc d \circledcirc a \circledcirc \overline{c}), (b \circledcirc a \circledcirc d \circledcirc \overline{c}), (a \circledcirc b \circledcirc d \circledcirc \overline{c}), (a \circledcirc b \circledcirc \overline{c} \circledcirc d), (a \circledcirc \overline{c} \circledcirc b \circledcirc d), (d \circledcirc a \circledcirc \overline{c} \circledcirc d),$
$(b \circledcirc d \circledcirc \overline{c} \circledcirc a), (b \circledcirc \overline{c} \circledcirc d \circledcirc a), (\overline{c} \circledcirc b \circledcirc d \circledcirc a), (\overline{c} \circledcirc b \circledcirc a \circledcirc d), (\overline{c} \circledcirc a \circledcirc b \circledcirc d), (d \circledcirc \overline{c} \circledcirc a \circledcirc d),$
$(d \circledcirc b \circledcirc \overline{a} \circledcirc \overline{c}), (d \circledcirc \overline{a} \circledcirc b \circledcirc \overline{c}), (\overline{a} \circledcirc d \circledcirc b \circledcirc \overline{c}), (\overline{a} \circledcirc d \circledcirc \overline{c} \circledcirc b), (\overline{a} \circledcirc \overline{c} \circledcirc d \circledcirc b), (d \circledcirc \overline{a} \circledcirc \overline{c} \circledcirc b),$
$(d \circledcirc b \circledcirc \overline{c} \circledcirc \overline{a}), (d \circledcirc \overline{c} \circledcirc b \circledcirc \overline{a}), (\overline{c} \circledcirc d \circledcirc b \circledcirc \overline{a}), (\overline{c} \circledcirc d \circledcirc \overline{a} \circledcirc b), (\overline{c} \circledcirc \overline{a} \circledcirc d \circledcirc b), (d \circledcirc \overline{c} \circledcirc \overline{a} \circledcirc b),$
$(d \circledcirc b \circledcirc a \circledcirc \overline{c}), (d \circledcirc a \circledcirc b \circledcirc \overline{c}), (a \circledcirc d \circledcirc b \circledcirc \overline{c}), (a \circledcirc d \circledcirc \overline{c} \circledcirc b), (a \circledcirc \overline{c} \circledcirc d \circledcirc b), (d \circledcirc a \circledcirc \overline{c} \circledcirc b),$
$(d \circledcirc b \circledcirc \overline{c} \circledcirc a), (d \circledcirc \overline{c} \circledcirc b \circledcirc a), (\overline{c} \circledcirc d \circledcirc b \circledcirc a), (\overline{c} \circledcirc d \circledcirc a \circledcirc b), (\overline{c} \circledcirc a \circledcirc d \circledcirc b), (d \circledcirc \overline{c} \circledcirc a \circledcirc b),$
$(d \circledcirc c \circledcirc \overline{a} \circledcirc \overline{b}), (d \circledcirc \overline{a} \circledcirc c \circledcirc \overline{b}), (\overline{a} \circledcirc d \circledcirc c \circledcirc \overline{b}), (\overline{a} \circledcirc d \circledcirc \overline{b} \circledcirc c), (\overline{a} \circledcirc \overline{b} \circledcirc d \circledcirc c), (d \circledcirc \overline{a} \circledcirc \overline{b} \circledcirc c),$
$(d \circledcirc c \circledcirc \overline{b} \circledcirc \overline{a}), (d \circledcirc \overline{b} \circledcirc c \circledcirc \overline{a}), (\overline{b} \circledcirc d \circledcirc c \circledcirc \overline{a}), (\overline{b} \circledcirc d \circledcirc \overline{a} \circledcirc c), (\overline{b} \circledcirc \overline{a} \circledcirc d \circledcirc c), (d \circledcirc \overline{b} \circledcirc \overline{a} \circledcirc c),$
$(d \circledcirc c \circledcirc a \circledcirc \overline{b}), (d \circledcirc a \circledcirc c \circledcirc \overline{b}), (a \circledcirc d \circledcirc c \circledcirc \overline{b}), (a \circledcirc d \circledcirc \overline{b} \circledcirc c), (a \circledcirc \overline{b} \circledcirc d \circledcirc c), (d \circledcirc a \circledcirc \overline{b} \circledcirc c),$
$(d \circledcirc c \circledcirc \overline{b} \circledcirc a), (d \circledcirc \overline{b} \circledcirc c \circledcirc a), (\overline{b} \circledcirc d \circledcirc c \circledcirc a), (\overline{b} \circledcirc d \circledcirc a \circledcirc c), (\overline{b} \circledcirc a \circledcirc d \circledcirc c), (d \circledcirc \overline{b} \circledcirc a \circledcirc c),$
$(d \circledcirc c \circledcirc \overline{a} \circledcirc \overline{b}), (d \circledcirc \overline{a} \circledcirc c \circledcirc \overline{b}), (\overline{a} \circledcirc d \circledcirc c \circledcirc \overline{b}), (\overline{a} \circledcirc d \circledcirc \overline{b} \circledcirc c), (\overline{a} \circledcirc \overline{b} \circledcirc d \circledcirc c), (d \circledcirc \overline{a} \circledcirc \overline{b} \circledcirc c),$
$(d \circledcirc c \circledcirc \overline{b} \circledcirc \overline{a}), (d \circledcirc \overline{b} \circledcirc c \circledcirc \overline{a}), (\overline{b} \circledcirc d \circledcirc c \circledcirc \overline{a}), (\overline{b} \circledcirc d \circledcirc \overline{a} \circledcirc c), (\overline{b} \circledcirc \overline{a} \circledcirc d \circledcirc c), (d \circledcirc \overline{b} \circledcirc \overline{a} \circledcirc c),$
$(d \circledcirc c \circledcirc a \circledcirc \overline{b}), (d \circledcirc a \circledcirc c \circledcirc \overline{b}), (a \circledcirc d \circledcirc c \circledcirc \overline{b}), (a \circledcirc d \circledcirc \overline{b} \circledcirc c), (a \circledcirc \overline{b} \circledcirc d \circledcirc c), (d \circledcirc a \circledcirc \overline{b} \circledcirc c),$
$(d \circledcirc c \circledcirc \overline{b} \circledcirc a), (d \circledcirc \overline{b} \circledcirc c \circledcirc a), (\overline{b} \circledcirc d \circledcirc c \circledcirc a), (\overline{b} \circledcirc d \circledcirc a \circledcirc c), (\overline{b} \circledcirc a \circledcirc d \circledcirc c), (d \circledcirc \overline{b} \circledcirc a \circledcirc c),$
$(a \circledcirc b \circledcirc c \circledcirc d), (b \circledcirc a \circledcirc c \circledcirc d), (b \circledcirc c \circledcirc a \circledcirc d), (b \circledcirc c \circledcirc d \circledcirc a),$
$(\overline{a} \circledcirc b \circledcirc c \circledcirc d), (b \circledcirc \overline{a} \circledcirc c \circledcirc d), (b \circledcirc c \circledcirc \overline{a} \circledcirc d), (b \circledcirc c \circledcirc d \circledcirc \overline{a}),$
$(a \circledcirc b \circledcirc d \circledcirc c), (b \circledcirc a \circledcirc d \circledcirc c), (b \circledcirc d \circledcirc a \circledcirc c), (b \circledcirc d \circledcirc c \circledcirc a),$
$(\overline{a} \circledcirc b \circledcirc d \circledcirc c), (b \circledcirc \overline{a} \circledcirc d \circledcirc c), (b \circledcirc d \circledcirc \overline{a} \circledcirc c), (b \circledcirc d \circledcirc c \circledcirc \overline{a}),$
$(a \circledcirc d \circledcirc b \circledcirc c), (d \circledcirc a \circledcirc b \circledcirc c), (d \circledcirc b \circledcirc a \circledcirc c), (d \circledcirc b \circledcirc c \circledcirc a),$
$(\overline{a} \circledcirc d \circledcirc b \circledcirc c), (d \circledcirc \overline{a} \circledcirc b \circledcirc c), (d \circledcirc b \circledcirc \overline{a} \circledcirc c), (d \circledcirc b \circledcirc c \circledcirc \overline{a})\}$

Für diese beiden Prozessmengen gilt nun, dass $P[C_S[R]] \subseteq P[C_R[S]]$, und so erfüllt die Realisierung R die Spezifikation S: $R \twoheadrightarrow S$.

□

4.4.2 Transformation der Aktionslogik in p/t-Netze

In diesem Abschnitt wird aus Gründen der Vollständigkeit wiedergegeben, wie in [Simo01] $\mathcal{LA}$-Module als p/t-Netz implementiert werden. Dies wird im Anschluss wieder an einem Beispiel vorgeführt.

Definition 69 Sei $W \in P_A$ ein Prozess über $\mathcal{A}$, $\mathcal{N} = (P, T, F)$ ein p/t-Netz mit leerer Anfangsmarkierung $M_0 = \mathbf{0}$, Starttransition s und Zieltransition g und sei $\sigma : L(W) \to T$ eine Abbildung, die jedem Literal aus W eine Transition zuordnet. $(\mathcal{N}, \mathbf{0}$ *realisiert* W (W ist *realisierbar* in $(\mathcal{N}, \mathbf{0}))$ gdw. ein Petri-Netz-Prozess Ω von $(\mathcal{N}, \mathbf{0})$ existiert, der

- die leere Markierung reproduziert

und in dem

- Start- und Zieltransition genau einmal feuern,

- Transitionen, die Aktionen aus W repräsentieren genauso oft vorkommen, wie die zugehörigen Aktionen in W (einschließlich niemals) und
- Subprozesse als Sequenz aktiviert werden, wenn die zugehörigen Subprozesse in W eine solche Sequenz bezeichnen. d. h. wenn $W = (W_1 \circledcirc W_2)$ ist $\Omega = \Omega_{W_1}\Omega_{W_2}$, wobei Ω_{W_i} eine Restriktion von Ω auf die Transitionen, die Aktionen von $W_i, i \in \{1, 2\}$ betreffen.

Die Prozessmenge $P[(\mathcal{N}, \mathbf{0})]$ ist die Menge aller in $(\mathcal{N}, \mathbf{0})$ realisierbaren Prozesse W.

[Fide93, Laut97, Simo01]

Definition 70 Sei $M \in \mathcal{M}_\mathcal{A}$ ein Modul über $\mathcal{A}$. Eine *Implementierung* $(\mathcal{N}[M], \mathbf{0}, \sigma)$ von M ist ein p/t-Netz $\mathcal{N}[M] = (P, T, F, W)$ mit der Anfangsmarkierung $M_0 = \mathbf{0}$ und einer Abbildung $\sigma : L(M) \to T$, in der genau die Prozesse aus $P[M]$ realisiert sind und die den Literalen aus M Transitionen aus $\mathcal{N}[M]$ zuordnet. Keine anderen Prozesse sind realisierbar in $(\mathcal{N}[M], \mathbf{0}, \sigma)$.

[Fide93, Laut97, Simo01]

In Abb. 4.14 bis 4.16 sind die Implementierungen einfacher Module gegeben. Diese sind bis auf das Modul $N[a \boxtimes b]$ in Abb. 4.15(c) aus [Simo01], S. 54f entnommen. In Abb. 4.14 stehen zunächst die Implementierungen der elementaren Module. Die Implementierung

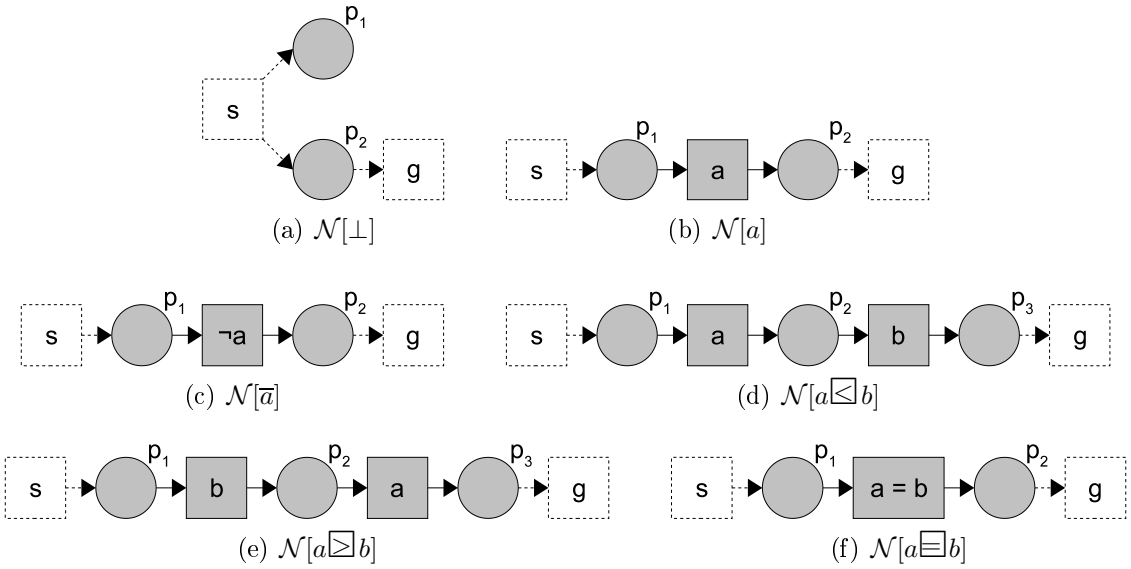
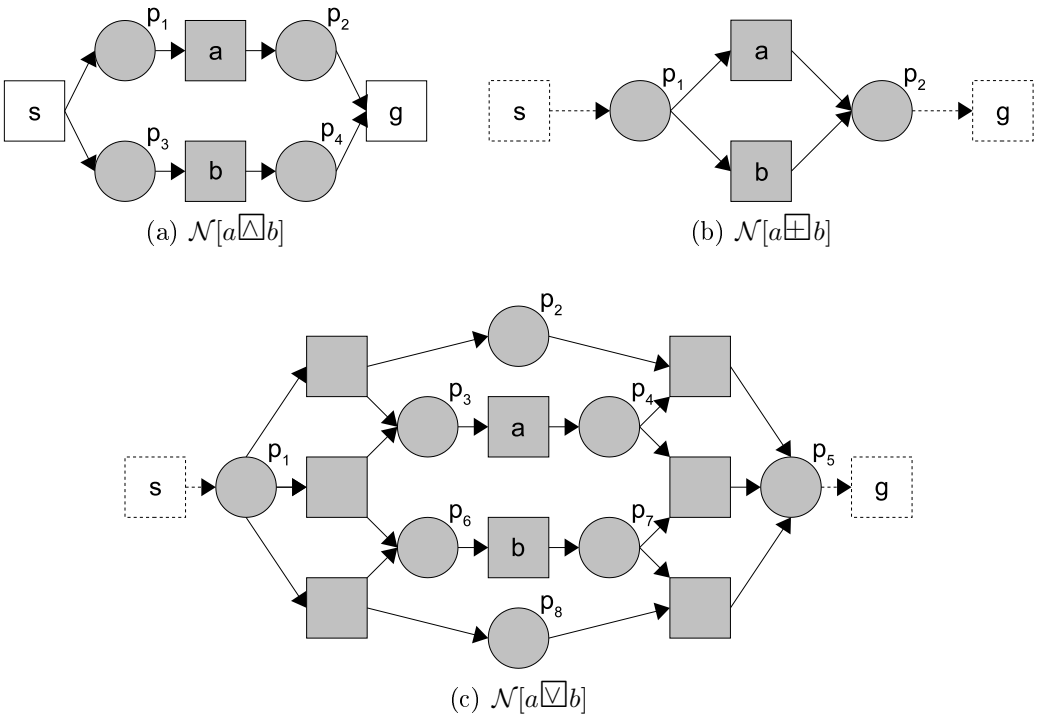


Abb. 4.14: Netzimplementierungen der elementaren Operatoren

von $[a \boxleft b]$ ist in dieser bzw. ähnlicher Form schon aus Abb. 3.6 und 4.12 bekannt. Ebenso sind die Netzimplementierungen der nicht-iterativen, nicht-elementaren Module in Abb. 4.15 bereits in ähnlicher Form in den vorigen Abschnitten verwendet worden. Die Netz-Implementierungen der Iterationen hingegen sind in dieser Form neu und stehen in Abb. 4.16.

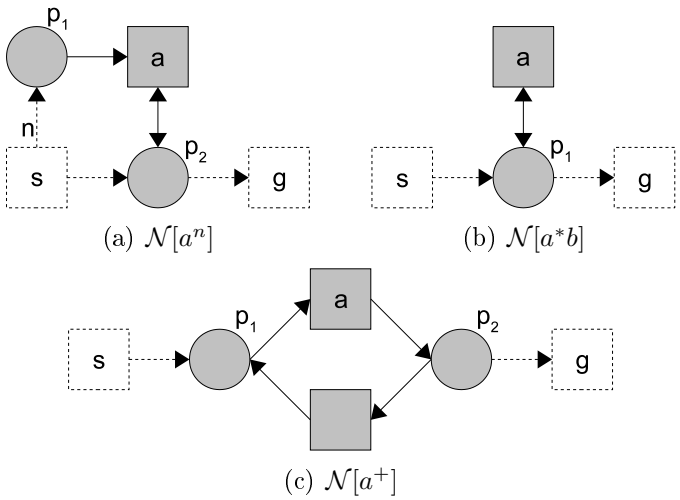
Im folgenden Beispiel werden genau diese Patterns auf ein $\mathcal{LA}$-Modul angewandt.



(a) $\mathcal{N}[a \boxtimes b]$ (b) $\mathcal{N}[a \boxplus b]$

(c) $\mathcal{N}[a \boxtimes b]$

Abb. 4.15: Netzimplementierungen der nicht-iterativen, nicht-elementaren Operatoren



(a) $\mathcal{N}[a^n]$ (b) $\mathcal{N}[a^*b]$

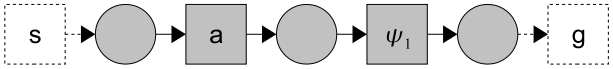
(c) $\mathcal{N}[a^+]$

Abb. 4.16: Netzimplementierungen der iterativen Operatoren

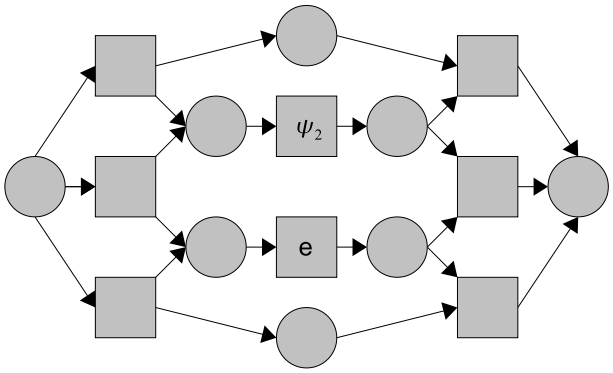
Beispiel 15 Sei

$$\psi = [a \boxleft [[b \boxplus [c \boxtriangle d]] \boxtriangledown e]]$$

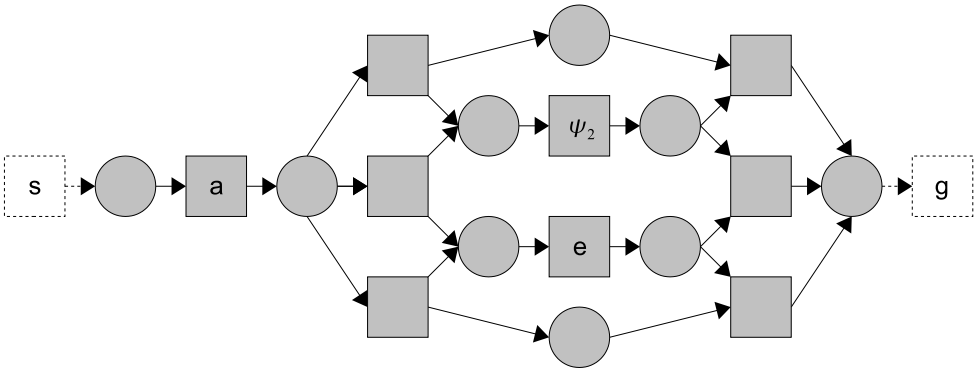
ein $\mathcal{LA}$-Modul. Zunächst soll die zugehörige Netz-Implementierung schrittweise erstellt werden. Dazu sei zunächst $\psi_1 = [[b \boxplus [c \boxtriangle d]] \boxtriangledown e]]$, sodass ψ_1 als eine einzige Aktion behandelt werden kann. Somit ist zu $\psi = [a \boxleft \psi_1]$ das Netz $\mathcal{N}[a \boxleft \psi_1]$:



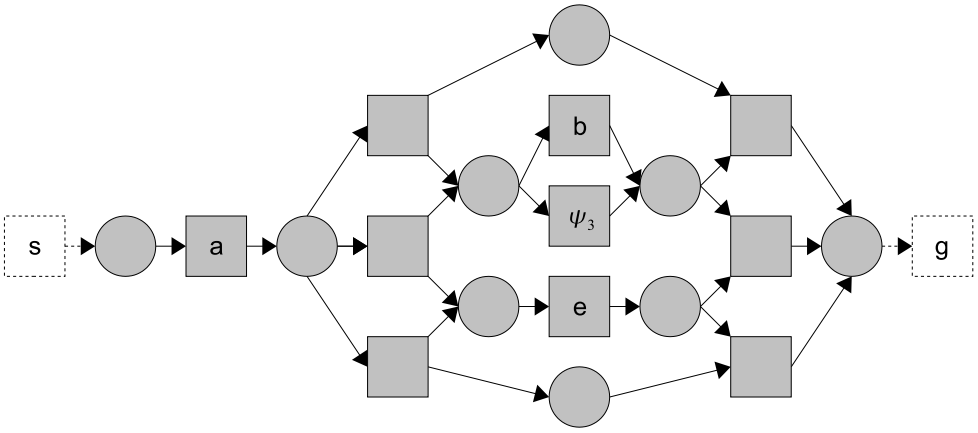
Als nächstes wird ψ_1 in $\psi_1 = [\psi_2 \boxtriangledown e]$ unterteilt und hierzu das Netz $\mathcal{N}[\psi_2 \boxtriangledown e]$ erstellt:



Dieses wird jetzt in das Netz $\mathcal{N}(a \boxleft \psi_1)$ eingesetzt und ersetzt dort die Transition ψ_1, sodass das Netz $\mathcal{N}[a \boxleft [\psi_2 \boxtriangledown e]$ entsteht:



So wird auch mit $\psi_2 = [b \boxplus \psi_3]$ und dem Netz $\mathcal{N}[a \boxleft [[b \boxplus \psi_3] \boxtriangledown e]]$ verfahren,



bis schließlich mit $\psi_3 = [c \boxtriangle d]$ das Netz $\mathcal{N}[a \boxleft [[b \boxplus [c \boxtriangle d]] \boxtriangledown e]]$ in Abb. 4.17 entsteht.

□

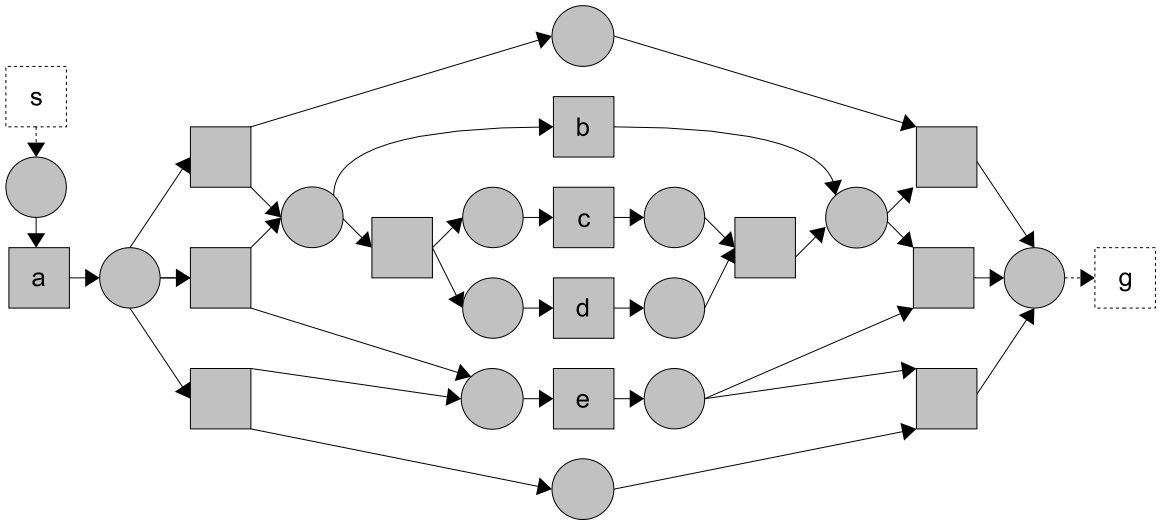


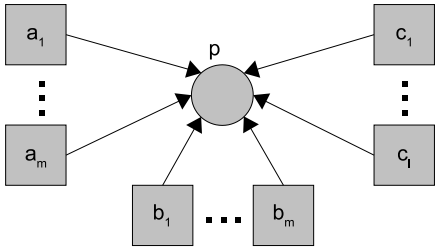
Abb. 4.17: Das Netz $N[\psi]$ aus Bsp. 15

4.4.3 Transformation von Workflow-Netzen in die Aktionslogik

Nachdem im vorigen Abschnitt ein Weg gezeigt wurde, wie ein $\mathcal{LA}$-Modul als p/t-Netz implementiert werden kann, wird in diesem Abschnitt, wie auch schon in den vorigen Kapiteln mit den dortigen Logiken durchgeführt, dargestellt, wie Workflow-Netze als $\mathcal{LA}$-Module implementiert werden können. Da die Aktionen der $\mathcal{LA}$ keine Argumente besitzen, beschränkt sich die folgende Konstruktion nur auf p/t-Netze, die Workflow-Netze sind. Außerdem sollen die Netze zykelfrei und vernünftig sein[9]. Zur Übersetzung von Workflow-Netzen gibt es zwei aufeinander aufbauende Möglichkeiten, die in den beiden folgenden Abschnitten erläutert werden. Bei der ersten werden alle Stellen des Netzes und deren Vor- und Nachbereiche betrachtet. Daraus werden zunächst Sequenzen aus jeweils zwei Modulen erstellt. Diese können dann noch mit Hilfe einiger Gesetze vereinfacht werden. So können auch die Pattern aus Abschnitt 4.4.2 und aus Kap. 3 aktionslogisch implementiert werden. Bei der zweiten Möglichkeit werden dann diese Pattern genutzt.

Elementare $\mathcal{LA}$-Implementierung

Sei $\mathcal{N} = (P, T, F)$ ein vernünftiges, zykelfreies Workflow-Netz. Um eine aktionslogische Implementierung $\mathcal{M}_\mathcal{N}$ dieses Netzes zu erhalten, wird bei der *elementaren aktionslogischen Implementierung* wie folgt vorgegangen. Für jede Stelle $p \in P$ wird der Vor- und Nachbereich bestimmt. Sei dazu $A = {}^\bullet p / p^\bullet = \{a_1, \ldots, a_m\}$ die Menge aller Transitionen, die nur im Vorbereich von p, $B = p^\bullet \cup^\bullet p = \{b_1, \ldots, b_n\}$ die Menge aller Transitionen, die sowohl im Vorbereich als auch im Nachbereich von p, und $C = p^\bullet / {}^\bullet p = \{c_1, \ldots, c_l\}$ die Menge aller Transitionen, die nur im Nachbereich von p liegen:



[9] Bei jedem $\mathcal{LA}$-Modul ist dies implizit der Fall. Deshalb muss dies auch bei den zu übertragenden Netzen gewährleistet sein.

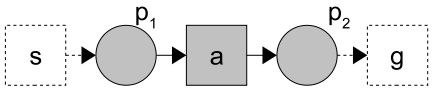
Die Idee ist nun, dass zuerst eine der Transitionen, die nur im Vorbereich der Stelle liegt feuern muss und so diese Stelle markiert, dann eine oder mehrere Transitionen, die im Vor- und Nachbereich von p liegen beliebig oft [10] feuern können und dann eine der Transitionen feuert, die sich nur im Nachbereich von p befindet und somit p wieder unmarkiert. So ergibt sich für diese Stelle das Modul

$$M_p = [a_1 \boxplus \ldots \boxplus a_m] \boxleft [[b_1 \boxdot \ldots \boxdot b_n]^*] \boxleft [c_1 \boxplus \ldots \boxplus c_l].$$

Ein solches Modul wird auch für alle anderen Stellen des Netzes erstellt und schließt diese durch $\boxtimes$ verbunden zu einem neuen Modul zusammen. Ist $\{p_1, \ldots, p_k\} = P$ so ergibt sich daraus für das gesamte Netz das Modul

$$\mathcal{M}_\mathcal{N} = M_{p_1} \boxtimes \ldots \boxtimes M_{p_k}.$$

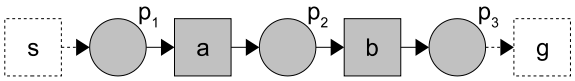
Beispiel 16 Dies soll beispielhaft an einigen der Pattern von S. 85 demonstriert werden. Für das Netz



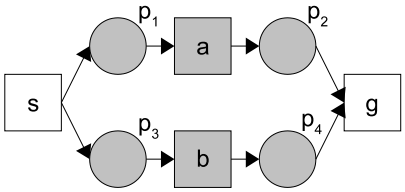
ist dann

$$\begin{aligned} \mathcal{M}_\mathcal{N} &= [M_{p_1} \boxtimes M_{p_2}] \\ &= [[s \boxleft a] \boxtimes [a \boxleft g]]. \end{aligned}$$

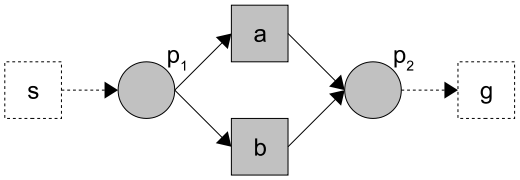
In der folgenden Tabelle stehen in der linken Spalte einige weitere Pattern. Jeweils rechts daneben steht die elementare aktionslogische Implementierung.



$[[s \boxleft a] \boxtimes [a \boxleft b] \boxtimes [b \boxleft g]]$



[[s⊴ a]⊠ [s⊴ b]⊠ [a⊴ g]⊠ [b⊴ g]



[[s⊴ [a ⊞ b]]⊠ [[a ⊞ b]⊴ g]

□

Es ist also auf diese Weise möglich, Workflow-Netze als $\mathcal{LA}$-Modul zu implementieren. Allerdings ist die Form des resultierenden Moduls unbefriedigend, da die Netzstruktur in ihr nur unzureichend widergegeben wird. Dies ist mit der folgenden Implementierung anders.

[10] Die Anzahl kann natürlich immer noch durch andere Teile des Netzes beschränkt sein. Dies kann auch 0-mal bedeuten.

Unter Anwendung der folgenden Gesetze ist es nämlich möglich, das Modul so umzuwandeln, dass es dem ursprünglichen Netz auch in seiner Struktur mehr ähnelt. Sind M_1, M_2, M_3 $\mathcal{LA}$-Module, so gilt:

(Assoziativität)

$[M_1 \boxleft [M_2 \boxleft M_3]] = [[M_1 \boxleft M_2] \boxleft M_3]$
$[M_1 \boxright [M_2 \boxright M_3]] = [[M_1 \boxright M_2] \boxright M_3]$
$[M_1 \boxminus [M_2 \boxminus M_3]] = [[M_1 \boxminus M_2] \boxminus M_3]$
$[M_1 \boxwedge [M_2 \boxwedge M_3]] = [[M_1 \boxwedge M_2] \boxwedge M_3]$
$[M_1 \boxvee [M_2 \boxvee M_3]] = [[M_1 \boxvee M_2] \boxvee M_3]$
$[M_1 \boxplus [M_2 \boxplus M_3]] = [[M_1 \boxplus M_2] \boxplus M_3]$

(Kommutativität)

$[M_1 \boxminus M_2] = [M_2 \boxminus M_1]$
$[M_1 \boxwedge M_2] = [M_2 \boxwedge M_1]$
$[M_1 \boxvee M_2] = [M_2 \boxvee M_1]$
$[M_1 \boxplus M_2] = [M_2 \boxplus M_1]$

(Distributivität)

$[M_1 \boxwedge [M_2 \boxvee M_3]] = [[M_1 \boxwedge M_2] \boxvee [M_1 \boxwedge M_3]]$
$[M_1 \boxvee [M_2 \boxwedge M_3]] = [[M_1 \boxvee M_2] \boxwedge [M_1 \boxvee M_3]]$
$[M_1 \boxleft [M_2 \boxwedge M_3]] = [[M_1 \boxleft M_2] \boxwedge [M_1 \boxleft M_3]]$
$[M_1 \boxleft [M_2 \boxvee M_3]] = [[M_1 \boxleft M_2] \boxvee [M_1 \boxleft M_3]]$
$[M_1 \boxleft [M_2 \boxplus M_3]] = [[M_1 \boxleft M_2] \boxplus [M_1 \boxleft M_3]]$
$[[M_1 \boxwedge M_2] \boxleft M_3] = [[M_1 \boxleft M_3] \boxwedge [M_2 \boxleft M_3]]$
$[[M_1 \boxvee M_2] \boxleft M_3] = [[M_1 \boxleft M_3] \boxvee [M_2 \boxleft M_3]]$
$[[M_1 \boxplus M_2] \boxleft M_3] = [[M_1 \boxleft M_3] \boxplus [M_2 \boxleft M_3]]$

(Transitivität)

$[[M_1 \boxleft M_2] \wedge [M_2 \boxleft M_3]] = [M_1 \boxleft M_2 \boxleft M_3]$
$[[M_1 \boxright M_2] \wedge [M_2 \boxright M_3]] = [M_1 \boxright M_2 \boxright M_3]$
$[[M_1 \boxminus M_2] \wedge [M_2 \boxminus M_3]] = [M_1 \boxminus M_2 \boxminus M_3]$
$[[M_1 \boxwedge M_2] \wedge [M_2 \boxwedge M_3]] = [M_1 \boxwedge M_2 \boxwedge M_3]$

Die Gesetze können alle über die Prozessmengen bewiesen werden. Bei allen Gesetzen ist jeweils die Prozessmenge der Module der linken Seite gleich der Prozessmenge der Module der rechten Seite.

Wurde zu einem Workflow-Netz $\mathcal{N}$ eine elementare $\mathcal{LA}$-Implementierung $\mathcal{M}_\mathcal{N}$ erstellt, werden diese Gesetze auf diese Implementierung angewendet. Ziel ist dabei, das Modul so umzuwandeln, dass jede Aktion x der Implementierung nur noch einmal im gesamten Modul vorkommt. Dieses Modul $\mathcal{M}'_\mathcal{N}$ wird dann als *aktionslogische Implementierung* des Workflow-Netzes $\mathcal{N}$ bezeichnet.

Beispiel 17 Die vier Gesetze werden nun auf die vier bereits erstellten elementaren $\mathcal{LA}$-Implementierungen aus Bsp. 16 angewendet. Für das erste der Module, $[[s \boxleft a] \boxwedge [a \boxleft g]]$, ergibt sich

$$\begin{aligned} & [[s \boxleft a] \boxwedge [a \boxleft g]] \\ =& [s \boxleft a \boxleft g]. && (Transitivität) \end{aligned}$$

So wie dieses lassen sich auch die anderen drei Module umformen.

$$\begin{aligned} & [[s \boxleft a] \boxwedge [a \boxleft b] \boxwedge [b \boxleft g]] \\ =& [[s \boxleft a \boxleft b] \boxwedge [b \boxleft g]] && (Transitivität) \\ =& [s \boxleft a \boxleft b \boxleft g] && (Transitivität) \end{aligned}$$

$$
\begin{aligned}
&[[s \trianglelefteq a] \triangle [s \trianglelefteq b]] \triangle [a \trianglelefteq g] \triangle [b \trianglelefteq g] \\
=&[[s \trianglelefteq [a \triangle b]] \triangle [a \trianglelefteq g] \triangle [b \trianglelefteq g] && (Distributivität) \\
=&[[s \trianglelefteq [a \triangle b]] \triangle [a \triangle b] \trianglelefteq g] && (Distributivität) \\
=&[[s \trianglelefteq [a \triangle b] \trianglelefteq g] && (Transitivität)
\end{aligned}
$$

$$
\begin{aligned}
&[[s \trianglelefteq [a \boxplus b]] \triangle [[a \boxplus b] \trianglelefteq g] \\
=&[s \trianglelefteq [a \boxplus b] \trianglelefteq g] && (Transitivität)
\end{aligned}
$$

Es ist zu sehen, dass nun alle Module mit „$s \trianglelefteq$“ beginnen und mit „$\trianglelefteq g$“ enden. Diese beiden Aktionen entsprechen genau der Start- und der Zieltransition, die in den Netzimplementierungen immer hinzugefügt wurden. Da in diesen vier Modulen bei der Netzimplementierung ansonsten keine zusätzliche Transitionen hinzugefügt wurde (wie dies bei $\triangledown$ der Fall ist), kommt man nach Weglassen von $s \trianglelefteq$ und $\trianglelefteq g$ wieder genau zu den ursprünglichen Modulen:

$$
\begin{aligned}
[a] &\overset{Netzimplementierung}{\rightarrow} & N[a] &\overset{\mathcal{LA}-Implementierung}{\rightarrow} & [a] \\
[a \trianglelefteq b] &\overset{Netzimplementierung}{\rightarrow} & N[a \trianglelefteq b] &\overset{\mathcal{LA}-Implementierung}{\rightarrow} & [a \trianglelefteq b] \\
[a \triangle b] &\overset{Netzimplementierung}{\rightarrow} & N[a \triangle b] &\overset{\mathcal{LA}-Implementierung}{\rightarrow} & [a \triangle b] \\
[a \boxplus b] &\overset{Netzimplementierung}{\rightarrow} & N[a \boxplus b] &\overset{\mathcal{LA}-Implementierung}{\rightarrow} & [a \boxplus b]
\end{aligned}
$$

□

Um im folgenden Abschnitt die Workflow-Pattern aus Kap. 3 zur $\mathcal{LA}$-Implementierung verwenden zu können, müssen zuerst noch die einzelnen Pattern mit Hilfe der gerade beschriebenen Methode als $\mathcal{LA}$-Modul implementiert werden. Die so erstellten Module stehen in der folgenden Liste. Dabei wurden jeweils die Start- und Zieltransition (die nur eingefügt wurden, um die einzelnen Pattern vernünftig zu machen), außer acht gelassen.

$$
\begin{aligned}
M[Sequenz] &= [a \trianglelefteq b] \\
M[UND\text{-}Split] &= [a \trianglelefteq [b \triangle c]] \\
M[UND\text{-}Join] &= [[a \triangle b] \trianglelefteq c] \\
M[XODER\text{-}Split] &= [a \trianglelefteq [b \boxplus c]] \\
M[XODER\text{-}Join] &= [[a \triangledown b] \trianglelefteq c] \\
M[ODER\text{-}Split] &= [[a \trianglelefteq [b \boxplus c]]] \\
M[ODER\text{-}Join] &= [[[a \triangledown b] \trianglelefteq c]] \\
M[Multipler\text{-}Join] &= [[[a \triangle b] \trianglelefteq c] \boxplus [[a \boxplus b] \trianglelefteq c^2]] \\
M[1\ aus\ m\ Diskriminator] &= [[a \triangle b] \trianglelefteq c] \boxplus [[a \boxplus b] \trianglelefteq c] \\
M[n\ aus\ m\ Diskriminator] &= [[a \triangle b] \trianglelefteq c] \boxplus [[a \boxplus b] \trianglelefteq c] \boxplus \\
&\quad [a \trianglelefteq c \trianglelefteq b \trianglelefteq c] \boxplus [b \trianglelefteq c \trianglelefteq b \trianglelefteq c] \\
M[Multiple\ Inst.\ ohne\ Synchronisierung] &= [a \trianglelefteq b^*] \\
M[Multiple\ Inst.\ mit\ Entwurfszeitwissen] &= [a \trianglelefteq b^n \trianglelefteq c] \\
M[Multiple\ Inst.\ mit\ Laufzeitwissen] &= [[a \trianglelefteq b^* \trianglelefteq c]] \\
M[Multiple\ Inst.\ ohne\ Entwurfszeitwissen] &= [[a \trianglelefteq b^* \trianglelefteq c]] \\
M[Verzögerter\ Split] &= [a \trianglelefteq [b \boxplus c]] \\
M[Verschachtelt\ paralleles\ Routing] &= [[[a_1 \triangle \ldots \triangle a_n] \triangle [\overline{[a_1 \boxminus a_2]} \triangle \ldots \\
&\quad [\overline{[a_1 \boxminus a_2]} \triangle \ldots \\
&\quad \triangle \overline{[a_1 \boxminus a_n]} \triangle \ldots \triangle \overline{[a_{n-1} \boxminus a_n]}]] \trianglelefteq b]
\end{aligned}
$$

$M[\textit{Meilenstein}] = [a \boxleft c \boxleft b]$

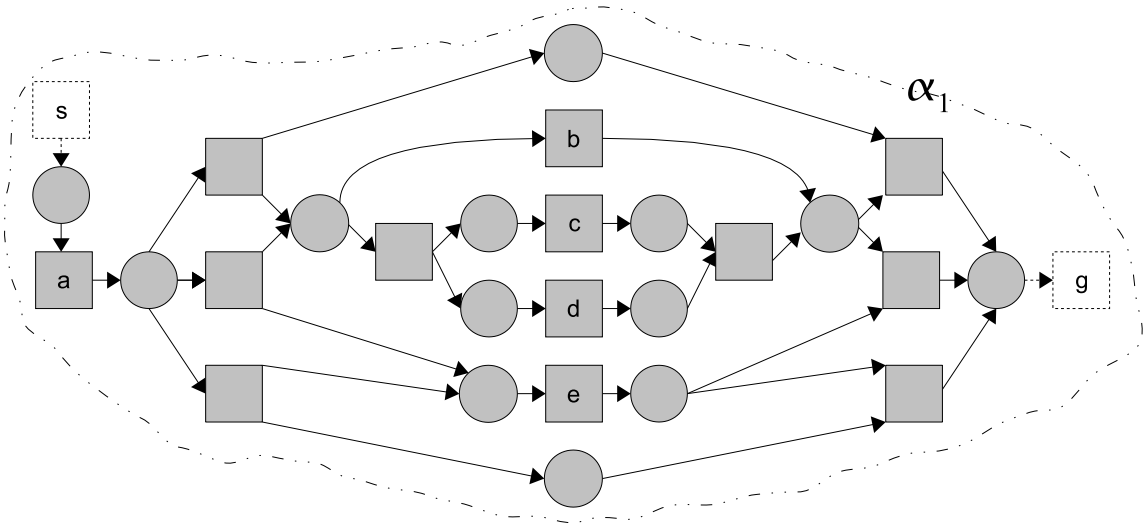
$M[\textit{Aktivität abbrechen}] = [a \boxleft [b \boxplus \bot] \boxleft c]$

Die beiden Pattern *Arbiträre Zykel* und *Instanz abbrechen* fehlen in dieser Liste, da diese nicht aktionslogisch darstellbar sind.

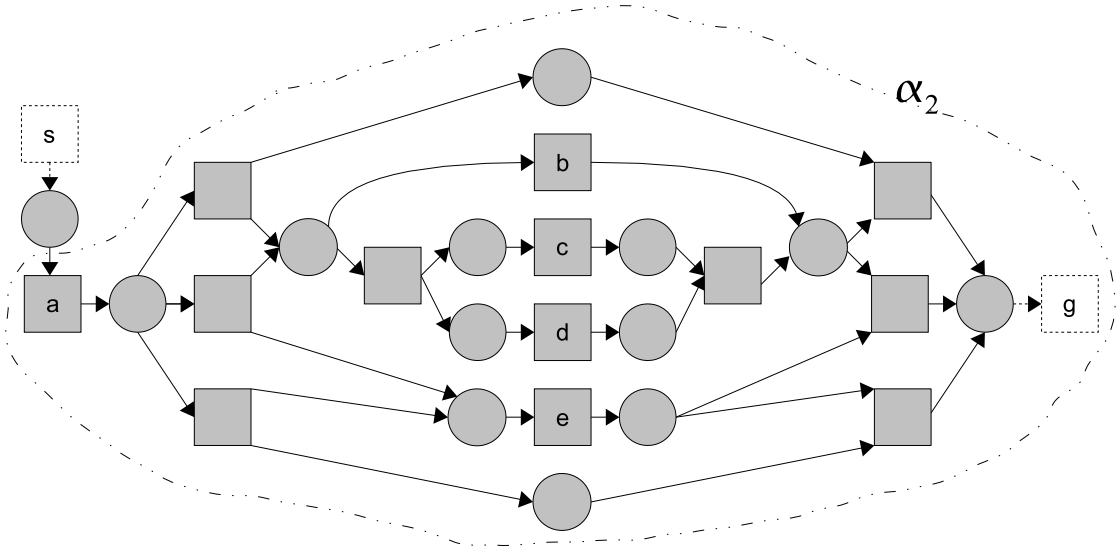
$\mathcal{LA}$-Implementierung mit Hilfe von Pattern

In diesem Kapitel wird beschrieben, wie mit Hilfe dieser Pattern und ihrer $\mathcal{LA}$-Implementierung eine aktionslogische Implementierung $\mathcal{M}_\mathcal{N}$ eines gegebenen vernünftigen, zykelfreien Netzes $\mathcal{N} = (P, T, F)$ konstruiert wird. Dazu werden in $\mathcal{N}$ genau diese Pattern identifiziert, für die schließlich bereits eine $\mathcal{LA}$-Implementierung bekannt ist und es wird versucht, das gesamte Netz mit Hilfe dieser Pattern zu erfassen. Dies kann sowohl Top-down als auch Bottom-up geschehen. Beispielhaft wird dies zunächst an dem oben erstellten Netz aus Abb. 4.17 im Top-down-Verfahren durchgeführt.

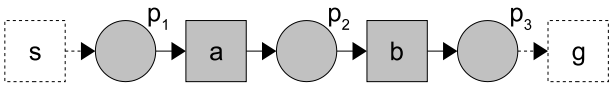
Beispiel 18 Sei α_i im Folgenden das $\mathcal{LA}$-Modul für den Teil des Netzes, für den noch kein Pattern gefunden und somit auch noch keine $\mathcal{LA}$-Implementierung erstellt wurde. Die α_i stellen somit im Netz Subnetze dar, die als solche jeweils im Netz eingerahmt und markiert sind. So ergibt sich zu Anfang das Netz



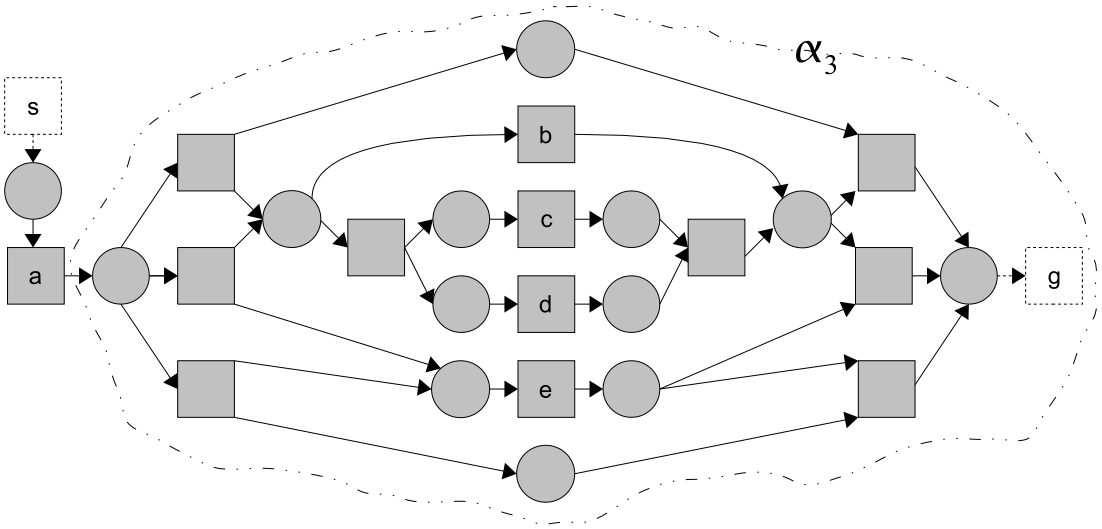
und $\mathcal{M}_\mathcal{N} = \alpha_1$. Wird α_2 wie in der folgenden Abbildung dargestellt festgelegt,



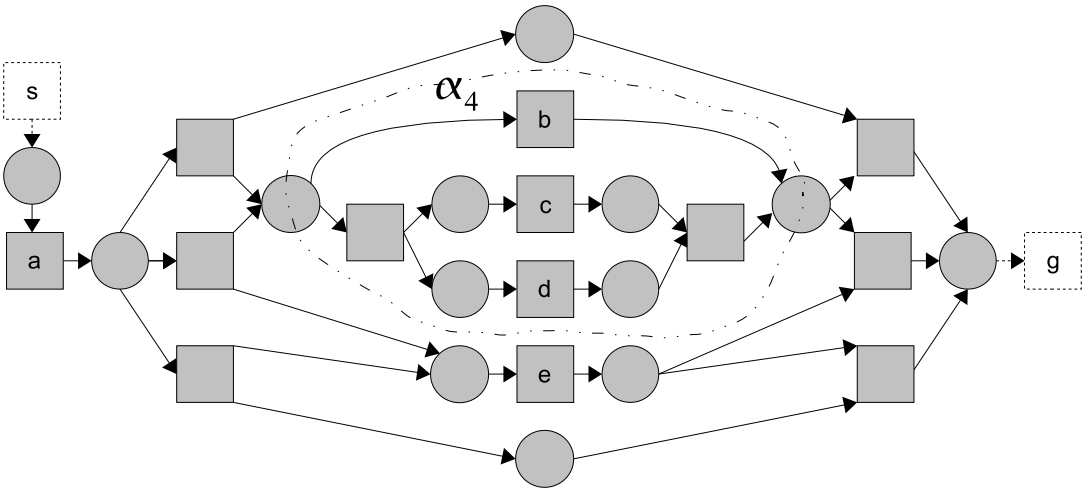
erinnert dies an die Netzdarstellung



des Moduls $a \boxtimes b$.[11] Übertragen auf das aktuelle Beispiel bedeutet dies, dass $\mathcal{M}_{\mathcal{N}} = [s \boxtimes \alpha_2]$. Das gleiche Pattern kann direkt noch zweimal für die Transitionen a und g gefunden werden. In der folgenden Abbildung ist α_3 entsprechend festgelegt.

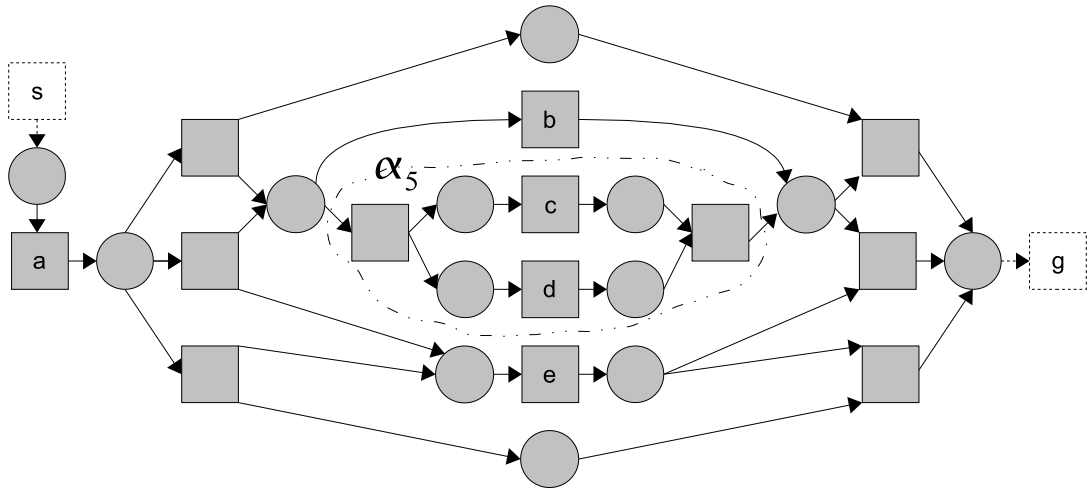


Nun ist $\mathcal{M}_{\mathcal{N}} = [s \boxtimes a \boxtimes \alpha_3 \boxtimes g]$. Das Subnetz α_3 implementiert $[\alpha_4 \boxtimes e]$,



sodass $\mathcal{M}_{\mathcal{N}} = [s \boxtimes a \boxtimes [\alpha_4 \boxtimes e] \boxtimes g]$. Danach wird α_4 durch $[b \boxplus \alpha_5]$ ersetzt

[11] Es sei daran erinnert, dass die Transitionen s und g bei der Netzimplementierung hinzugefügt wurden.



und $\mathcal{M}_{\mathcal{N}} = [s \boxslash a \boxslash [b \boxplus \alpha_5 \boxtriangledown e] \boxslash g]$. α_5 entspricht $[c \boxtriangle d]$. Mit diesem letzten Schritt wurde auch zu dem letzten Subnetz ein Modul gefunden, sodass das endgültige Modul

$$\mathcal{M}_{\mathcal{N}} = [s \boxslash a \boxslash [b \boxplus [c \boxtriangle d] \boxtriangledown e] \boxslash g]$$

ist.

□

Beide Methoden der Implementierung eines gegebenen, vernüftigen, zykelfreien Workflow-Netzes als $\mathcal{LA}$-Modul haben ihre Vorzüge, aber auch ihre Nachteile. Vorteile der ersten Methode sind, dass

- die elementare Implementierung leicht zu erstellen ist und
- die Methode bei allen solchen Netzen anwendbar ist.

Nachteilig ist, dass

- die Umformung nach den Gesetzen auf S. 90f. bei großen Netzen sehr kompliziert ist.

Die zweite Methode hat den Vorzug, dass

- sie auch auf großen Netzen (relativ) leicht anzuwenden ist.

Es zeigt sich allerdings, dass

- die Methode nicht auf alle Workflow-Netze anwendbar ist,

da nicht festgelegt ist, dass ein Workflow-Netz nur nach den hier angegebenen Pattern konstruiert sein muss. Weicht das Netz nur leicht von den gegebenen Pattern ab, schlägt diese Methode fehl.

Als Lösung bietet sich eine Kombination aus beiden Methoden an:

1. Suche sowohl Bottom-up als auch Top-down nach bekannten Pattern und implementiere so einen möglichst großen Teil des Netz in Aktionslogik.
2. Implementiere die nach 1. nicht implementierbaren Subnetze als elementare Implementierungen und wende die Gesetze auf S. 90f. an.

So nutzt man die Vorteile beider Methoden aus. Möglichst große Teile des Netzes werden mit Hilfe der Pattern-Methode implementiert, in der Hoffnung, dass die verbleibenden Teile, die durch die elementare Implementierung und Gesetze implementiert werden müssen, klein und übersichtlich sind.

Bevor diese Kombination der beiden Methoden im folgenden Kapitel auf zwei Beispiele angewendet wird, soll noch einmal kurz zur Transaktionslogik zurückgekehrt werden.

4.4.4 Zusammenhang zwischen Aktionslogik und Transaktionslogik

Am Ende des Abschnitts 4.3.3 wurde bereits angedeutet, dass die gerade entwickelte Vorgehensweise auch für die Implementierung einer $\mathcal{TR}$-Formel zu einem gegebenen Netz verwendet werden kann. Zu diesem Zweck wird zunächst gezeigt, wie alle Konnektoren der Aktionslogik durch Konnektoren der $\mathcal{TR}$ dargestellt werden können. Für die meisten der $\mathcal{LA}$-Konnektoren lässt sich dies durch einfaches Austauschen des $\mathcal{LA}$-Konnektors durch den $\mathcal{TR}$-Konnektor, der die gleiche Semantik hat, erreichen.

Definition 71 Seien $a, b \in \mathcal{M}$ zwei Module, sei $(\mathcal{B}, P, D)$ ein $\mathcal{TR}$-Programm und $a_{\mathcal{TR}}, b_{\mathcal{TR}} \in P$ seien respektive a und b entsprechende Regeln. Es gilt, dass

- $[\overline{a}] \hat{=} (\neg a_{\mathcal{TR}})$,
- $[a \boxleftarrow b] \hat{=} (a_{\mathcal{TR}} \otimes b_{\mathcal{TR}})$,
- $[a \boxrightarrow b] \hat{=} (b_{\mathcal{TR}} \otimes a_{\mathcal{TR}})$,
- $[a \boxminus b] \hat{=} (a_{\mathcal{TR}} | b_{\mathcal{TR}})$,
- $[a \boxwedge b] \hat{=} (a_{\mathcal{TR}} \wedge b_{\mathcal{TR}})$ und
- $[a \boxvee b] \hat{=} (a_{\mathcal{TR}} \vee b_{\mathcal{TR}})$.

□

Das Modul $a \boxplus b$ soll in der $\mathcal{TR}$ wie folgt dargestellt werden: Zunächst wird ein eindeutig identifizierbares Datum $XRes(a_{\mathcal{TR}} b_{\mathcal{TR}})$ in die Datenbasis gelegt:

$$\langle D, D \cup XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}) \rangle \mathit{ins{:}XRes}(a_{\mathcal{TR}} b_{\mathcal{TR}}) \in \mathcal{B}$$

Danach wird nach Entfernen dieses Datums $a_{\mathcal{TR}}$ ausgeführt oder es wird später (wiederum nach Entfernen des Datums aus der Datenbasis) $b_{\mathcal{TR}}$ ausgeführt:

$$(\mathit{del{:}XRes}(a_{\mathcal{TR}} b_{\mathcal{TR}}) \otimes a_{\mathcal{TR}}) \oplus (\mathit{del{:}XRes}(a_{\mathcal{TR}} b_{\mathcal{TR}}) \otimes b_{\mathcal{TR}}).$$

Dabei ist

$$\langle D \cup XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}), D \rangle \mathit{del{:}XRes}(a_{\mathcal{TR}} b_{\mathcal{TR}}) \in \mathcal{B}.$$

Das nur einfache Vorhandensein des Datums $XRes(a_{\mathcal{TR}} b_{\mathcal{TR}})$ in der Datenbasis garantiert hierbei, dass nur eine der beiden Transaktionen $a_{\mathcal{TR}}$ und $b_{\mathcal{TR}}$ ausgeführt werden kann. Formal zusammengefasst ergibt das:

Definition 72 Seien $a, b \in \mathcal{M}_\mathcal{A}$ Module über $\mathcal{A}$ und $a_{\mathcal{TR}}, b_{\mathcal{TR}}$ seien die entsprechenden $\mathcal{TR}$-Formeln. Dann ist die Transaktion

$$(a_{\mathcal{TR}} \boxplus b_{\mathcal{TR}})$$

definiert durch die Regel

$$\begin{aligned}(a_{\mathcal{TR}} \boxplus b_{\mathcal{TR}}) \leftarrow (&ins{:}XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}) \otimes \\ &(del{:}XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}) \otimes b_{\mathcal{TR}}) \oplus (del{:}XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}) \otimes b_{\mathcal{TR}}))\end{aligned}$$

und die elementaren Transitionen

$$\langle D, D \cup XRes(a_{\mathcal{TR}} b_{\mathcal{TR}})\rangle\, ins{:}XRes(a_{\mathcal{TR}} b_{\mathcal{TR}})$$

und

$$\langle D \cup XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}), D\rangle\, del{:}XRes(a_{\mathcal{TR}} b_{\mathcal{TR}}).$$

□

Da der $\boxplus$-Operator also mit den Mitteln der $\mathcal{TR}$ ausgedrückt werden kann, erweitert er diese nicht und soll im Folgenden in der $\mathcal{TR}$ als Abkürzung wie gerade beschrieben verwendet werden.

Auch für den $\oplus$-Operator der $\mathcal{TR}$ wird nun gezeigt, wie dieser aktionslogisch ausgedrückt werden kann. Die Formel $a_{\mathcal{TR}} \oplus b_{\mathcal{TR}}$ ist genau dann wahr, wenn einer der folgenden Fälle eintritt:

- $a_{\mathcal{TR}} \otimes b_{\mathcal{TR}}$,
- $\neg a_{\mathcal{TR}} \otimes b_{\mathcal{TR}}$ und
- $a_{\mathcal{TR}} \otimes \neg b_{\mathcal{TR}}$.

Diese einzelnen Fälle lassen sich nach Definition 71 aktionslogisch darstellen als

- $[a \boxleftarrow b]$,
- $[\overline{a} \boxleftarrow b]$ und
- $[a \boxleftarrow \overline{b}]$.

Dies kann wiederum zusammengefasst werden als $[[a \boxleftarrow b] \boxplus [\overline{a} \boxleftarrow b] \boxplus [a \boxleftarrow \overline{b}]]$, sodass

$$[a \oplus b] = [[a \boxleftarrow b] \boxplus [\overline{a} \boxleftarrow b] \boxplus [a \boxleftarrow \overline{b}]] \hat{=} (a_{\mathcal{TR}} \oplus b_{\mathcal{TR}}).$$

Wie der Konnektor $\boxplus$ in der $\mathcal{TR}$ kann auch der Konnektor $\oplus$ in der $\mathcal{LA}$ als abkürzende Schreibweise, wie gerade beschrieben, verwendet werden.

Schließlich wird noch für die Iterations-Operatoren eine entsprechende $\mathcal{TR}$-Darstellung benötigt. Sei $\mathcal{A}$ eine Menge von Aktionen und $\mathcal{M}_\mathcal{A}$ die Menge der Module über $\mathcal{A}$. Die n-fache Iteration M^n eines Moduls $M \in \mathcal{M}_\mathcal{A}$ ist nur eine abgekürzte Schreibweise von $[M_1 \boxleftarrow \ldots \boxleftarrow M_n]$ wobei M_i die i-te Ausführung des Moduls M ist und hat somit in der $\mathcal{TR}$ die Entsprechung $\underbrace{M_{\mathcal{TR}} \otimes \cdots \otimes M_{\mathcal{TR}}}_{n-mal}$.

Ebenso ist $[M^+]$ eine Abkürzung für $[M \boxleftarrow M^*]$ sodass nur eine $\mathcal{TR}$-Formel gefunden werden muss, die dem Modul $[M^*]$ entspricht. Dies kann mit Hilfe der Rekursion realisiert werden.

Satz 1 *Der *-Operator der $\mathcal{LA}$ wird in der $\mathcal{TR}$ realisiert als*

$$M^*_{\mathcal{TR}} \leftarrow (M_{\mathcal{TR}} \otimes M^*_{\mathcal{TR}}) \vee \epsilon.$$

□

Dabei kommt dem '*' in 'M^*' in der $\mathcal{TR}$ keine besondere Bedeutung zu, sondern 'M^*' ist nur der Name einer $\mathcal{TR}$-Formel.

Auf diese Art und Weise lässt sich zum Beispiel das oben bestimmte Modul

$$\mathcal{M}_{\mathcal{N}} = s \boxleftarrow a \boxleftarrow [b \boxplus [c \boxwedge d] \boxvee e] \boxleftarrow g$$

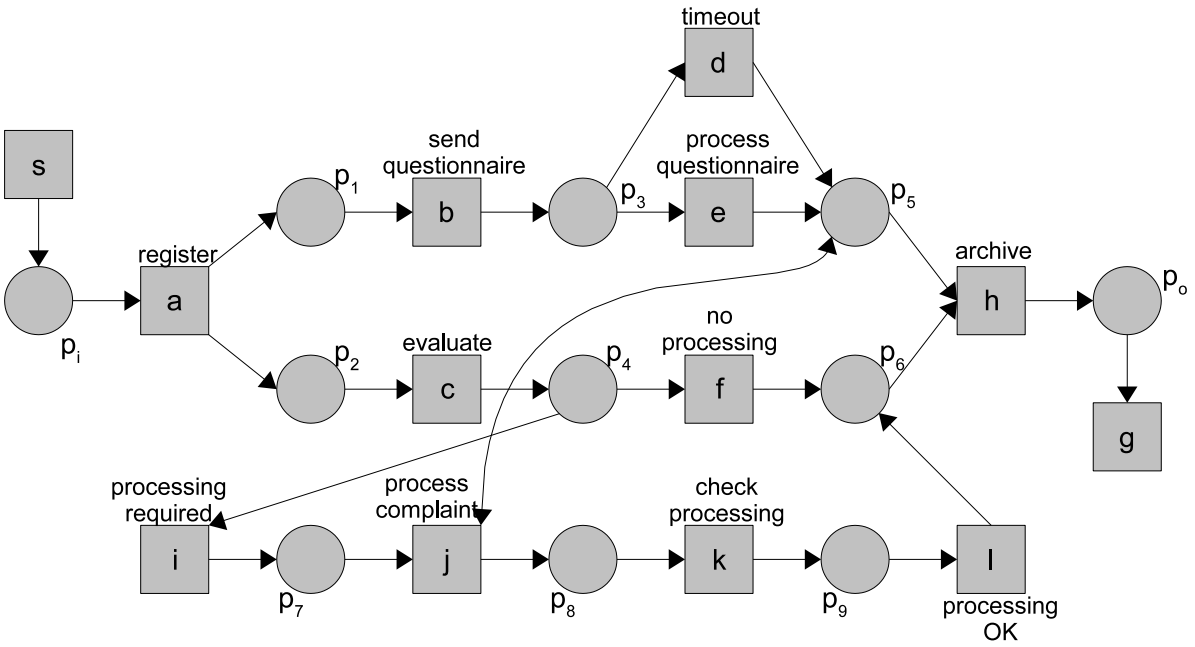
als $\mathcal{TR}$-Formel so darstellen:

$$\alpha_{\mathcal{M}_{\mathcal{N}}} \leftarrow s_{\mathcal{TR}} \otimes a_{\mathcal{TR}} \otimes (b_{\mathcal{TR}} \boxplus (c_{\mathcal{TR}} \wedge d_{\mathcal{TR}}) \vee e_{\mathcal{TR}}) \otimes g_{\mathcal{TR}}.$$

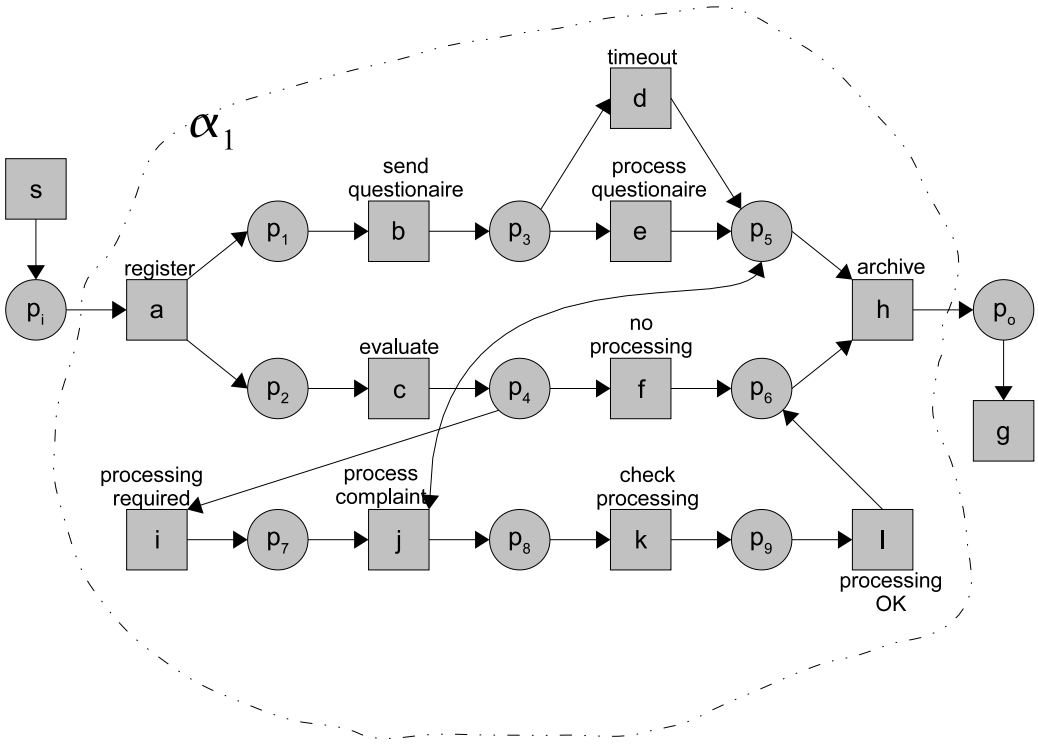
4.5 Abschließende Beispiele

In diesem Abschnitt wird das bisher Beschriebene noch einmal auf zwei größere Beispiele angewendet. Als erstes Beispiel wird das p/t-Netz des Reklamationsprozesses von S. 26 aktionslogisch implementiert. Anschließend wird dieses Modul in eine Transaktionsbasis übersetzt und ergibt zusammen mit der $\mathcal{TR}$-Implementierung aus Abschnitt 4.3.3 dieses Netzes eine neue, der Netzstruktur mehr entsprechende Implementierung in $\mathcal{TR}$.

Beispiel 19 Das p/t-Netz $\mathcal{N}$ des Reklamationsprozesses ist hier noch einmal dargestellt.



Sei α das Modul, das das Netz $\mathcal{N}$ aktionslogisch implementiert: $\mathcal{M}_{\mathcal{N}} = \alpha$. Wird nach evtl. vorhandenen Pattern gesucht, so findet man bei der Top-down-Suche das Modul α_1,

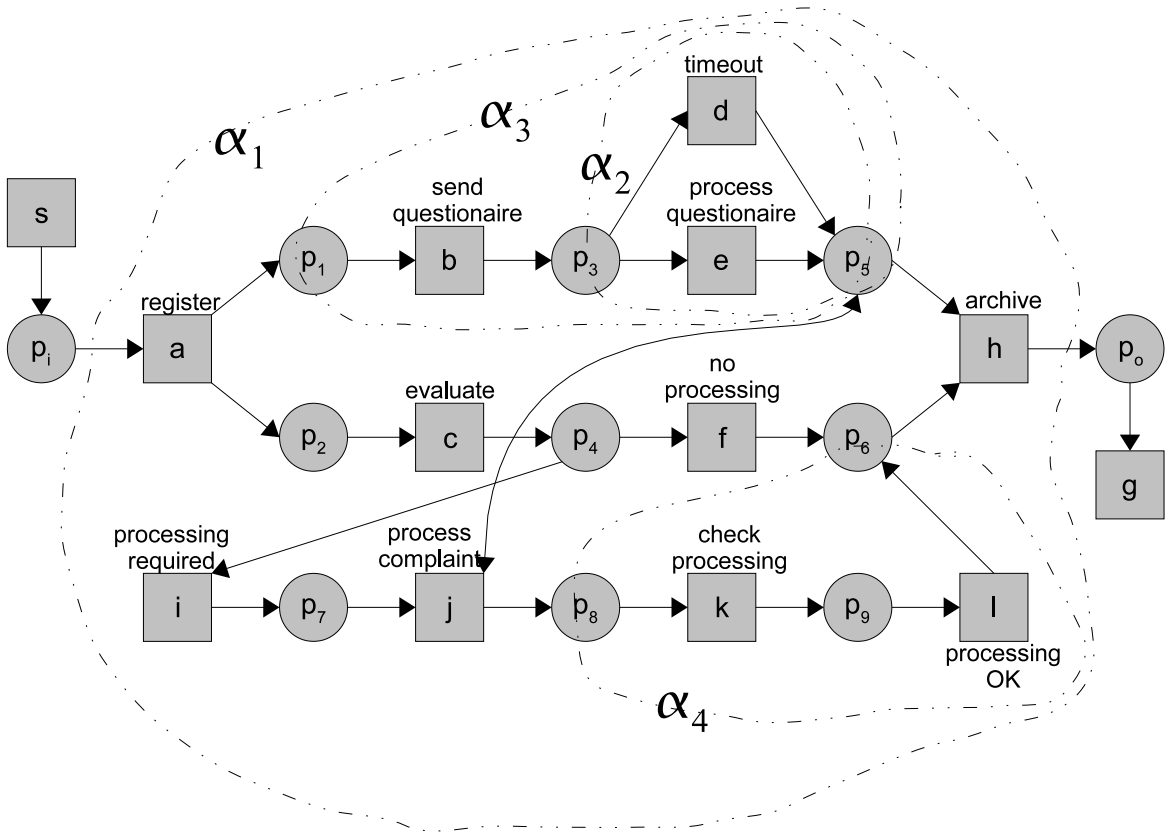


das nach s und vor g ausgeführt wird:

$$\mathcal{M}_{\mathcal{N}} = [s \boxtimes \alpha_1 \boxtimes g]$$

An diesem Punkt können bereits keine weiteren Pattern eindeutig identifiziert werden. Die Kante zwischen p_5 und j verhindert dies auch wenn auf den ersten Blick das Meilenstein-Pattern anwendbar erscheint. Auf Aktion c folgt aber ein XODER-Split und der Meilenstein p_5 wird nur von einem Teil des Splits, nämlich der Sequenz der Aktionen i, j, k und l, benötigt.

Neben der Top-down-Suche besteht aber noch die Möglichkeit, Bottom-up nach möglichen Pattern zu suchen. Auf diese Weise lassen sich die Module α_2 bis α_4 finden,



die aktionslogisch so implementiert werden:

$$\alpha_2 = [d \boxplus e]$$
$$\alpha_3 = [b \boxtimes \alpha_2]$$
$$\alpha_4 = [k \boxtimes l]$$

Da kein weiteres Pattern gefunden werden kann, geht man nun dazu über, elementare Prozesse für α_1 unter Verwendung der gerade identifizierten Module α_2 bis α_4 zu implementieren und diese anschließend umzuformen.

$$
\begin{aligned}
\alpha_1 =& [[a \boxleft \alpha_3] \boxwedge [a \boxleft c] \boxwedge [c \boxleft [f \boxplus i]] \boxwedge \\
& [\alpha_3 \boxleft j^* \boxleft h] \boxwedge [[f \boxplus \alpha_4] \boxleft h] \boxwedge \\
& [i \boxleft j] \boxwedge [j \boxleft \alpha_4]] \\
=& [[a \boxleft \alpha_3] \boxwedge [a \boxleft c] \boxwedge [c \boxleft [f \boxplus i]] \boxwedge && (Transitivität) \\
& [\alpha_3 \boxleft j^* \boxleft h] \boxwedge [[f \boxplus \alpha_4] \boxleft h] \boxwedge [i \boxleft j \boxleft \alpha_4]] \\
=& [[a \boxleft [\alpha_3 \boxwedge c]] \boxwedge [c \boxleft [f \boxplus i]] \boxwedge && (Distributivität) \\
& [\alpha_3 \boxleft j^* \boxleft h] \boxwedge [[f \boxplus \alpha_4] \boxleft h] \boxwedge [i \boxleft j \boxleft \alpha_4]] \\
=& [[a \boxleft [\alpha_3 \boxwedge c]] \boxwedge [c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]]] \boxwedge \\
& [\alpha_3 \boxleft j^* \boxleft h] \boxwedge [[f \boxplus [i \boxleft j \boxleft \alpha_4]] \boxleft h] \boxwedge \\
=& [[a \boxleft [\alpha_3 \boxwedge c]] \boxwedge [c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]] \boxleft h] \boxwedge && (Transitivität) \\
& [\alpha_3 \boxleft j^* \boxleft h] \boxwedge \\
=& [[a \boxleft [\alpha_3 \boxwedge c]] \boxwedge && (Distributivität) \\
& [[[c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]]] \boxwedge [\alpha_3 \boxleft j^*]] \boxleft h] \boxwedge \\
=& [[a \boxleft [[\alpha_3 \boxleft j^*] [c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]]]]] \boxwedge \\
& [[[c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]]] \boxwedge [\alpha_3 \boxleft j^*]] \boxleft h] \boxwedge \\
=& [a \boxleft [[\alpha_3 \boxleft j^*] \boxwedge [c \boxleft [f \boxplus [i \boxleft j \boxleft \alpha_4]]]] \boxleft h] && (Transitivität)
\end{aligned}
$$

Wenn man jetzt wieder die Werte von α_2 bis α_4 einsetzt, ist

$$\alpha_1 = [a \boxleft [[[b \boxleft [d \boxplus e]] \boxleft j^*] \boxwedge [c \boxleft [f \boxplus [i \boxleft j \boxleft k \boxleft l]]]] \boxleft h].$$

So kann jetzt auch in $\mathcal{M}_\mathcal{N} = [s \boxleft \alpha_1 \boxleft g]$, α_1 ersetzt werden und man erhält für das gesamte Netz $\mathcal{N}$ die $\mathcal{LA}$-Implementierung

$$\mathcal{M}_\mathcal{N} = [s \boxleft [a \boxleft [[[b \boxleft [d \boxplus e]] \boxleft j^*] \boxwedge [c \boxleft [f \boxplus [i \boxleft j \boxleft k \boxleft l]]]] \boxleft h] \boxleft g].$$

Nach Erstellen der $\mathcal{LA}$-Implementierung $\mathcal{M}_\mathcal{N}$ des Netzes $\mathcal{N}$ kann jetzt unter Zuhilfenahme derselben eine $\mathcal{TR}$-Implementierung $Prog_{\mathcal{TR}_\mathcal{N}} = (\mathcal{B}_\mathcal{N}, P_\mathcal{N}, D_\mathcal{N})$ erstellt werden. Die Menge $\mathcal{B}$ der elementaren Transitionen wurde bereits in Beispiel 13 auf S. 75 bestimmt als

$$
\begin{aligned}
\mathcal{B}_\mathcal{N} = \ \{ \ & (\langle D, D \cup \{p_i\}\rangle s), (\langle D \cup \{p_i\}, D \cup \{p_1, p_2\}\rangle a), (\langle D \cup \{p_1\}, D \cup \{p_3\}\rangle b), \\
& (\langle D \cup \{p_2\}, D \cup \{p_4\}\rangle c), (\langle D \cup \{p_3\}, D \cup \{p_5\}\rangle d), (\langle D \cup \{p_3\}, D \cup \{p_5\}\rangle e), \\
& (\langle D \cup \{p_4\}, D \cup \{p_6\}\rangle f), (\langle D \cup \{p_5, p_6\}, D \cup \{p_o\}\rangle h), (\langle D \cup \{p_4\}, D \cup \{p_7\}\rangle i), \\
& (\langle D \cup \{p_5, p_7\}, D \cup \{p_5, p_8\}\rangle j), (\langle D \cup \{p_8\}, D \cup \{p_9\}\rangle k), \\
& (\langle D \cup \{p_9\}, D \cup \{p_6\}\rangle l), (\langle D \cup \{p_o\}, D \cup \{\}\rangle g)\}.
\end{aligned}
$$

Ebenso ist auch die Datenbasis D von dort zu übernehmen als

$$D_\mathcal{N} = \{\}.$$

Die Transaktionsbasis hingegen soll nicht von dort übernommen werden, sondern durch Umwandeln des zuvor implementierten Moduls $\mathcal{M}_\mathcal{N}$ in die entsprechende $\mathcal{TR}$-Formel bestimmt werden. Durch das einfache Umwandeln wie, in Abschnitt 4.4.4 beschrieben, erhält man:

$$
\begin{aligned}
P_\mathcal{N} = \{(\mathit{ProcessOfComplaints} \leftarrow & (s \otimes (a \otimes (((b \otimes (d \boxplus e)) \otimes j^*) \\
& \wedge (c \otimes (f \boxplus (i \otimes j \otimes k \otimes l)))) \otimes h) \otimes g))\}.
\end{aligned}
$$

Natürlich können auch wieder Subformeln gebildet werden, um die Transaktionsbasis übersichtlicher zu gestalten. Dann ist

$$\begin{aligned} P_{\mathcal{N}} = \quad \{ \quad & (ProcessOfComplaints \leftarrow s \otimes a \otimes regel_1 \otimes h \otimes g) \\ & (regel_1 \leftarrow (regel_2 \wedge regel_3)) \\ & (regel_2 \leftarrow ((b \otimes (d \boxplus e)) \otimes j^*)) \\ & (regel_3 \leftarrow (c \otimes (f \boxplus (i \otimes j \otimes k \otimes l)))) \\ & (j^* \leftarrow ((j \otimes j^*) \vee \epsilon))\}. \end{aligned}$$

□

Im zweiten Beispiel werden die Implementierungstechniken auf einen aus [RoKiFe, S. 4] entnommenen Kontrollflussgraphen (KFG) angewendet. Auch wenn KFGs normalerweise dazu verwendet werden, den Kontrollfluss von Computer-Programmen zu modellieren, können sie auch dazu verwendet werden, Workflows zu beschreiben. Zum Erstellen einer $\mathcal{LA}$-Implementierung wird dieser zuerst in ein Petri-Netz umgewandelt. Es wird sich zeigen, dass dieses Netz vollständig als Zusammensetzung von Pattern erfasst werden kann.

Beispiel 20 Der umzuwandelnde Kontrollflussgraph ist in der folgenden Abb. 4.18 dargestellt. Der Graph besteht aus Knoten (z. B. *placeOrder*), die Aktionen darstellen, und aus

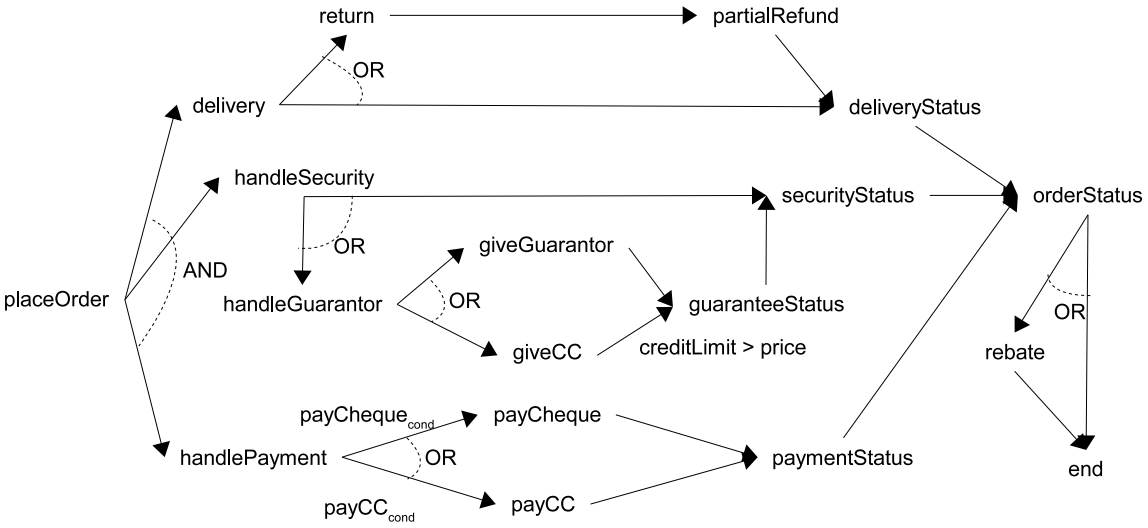
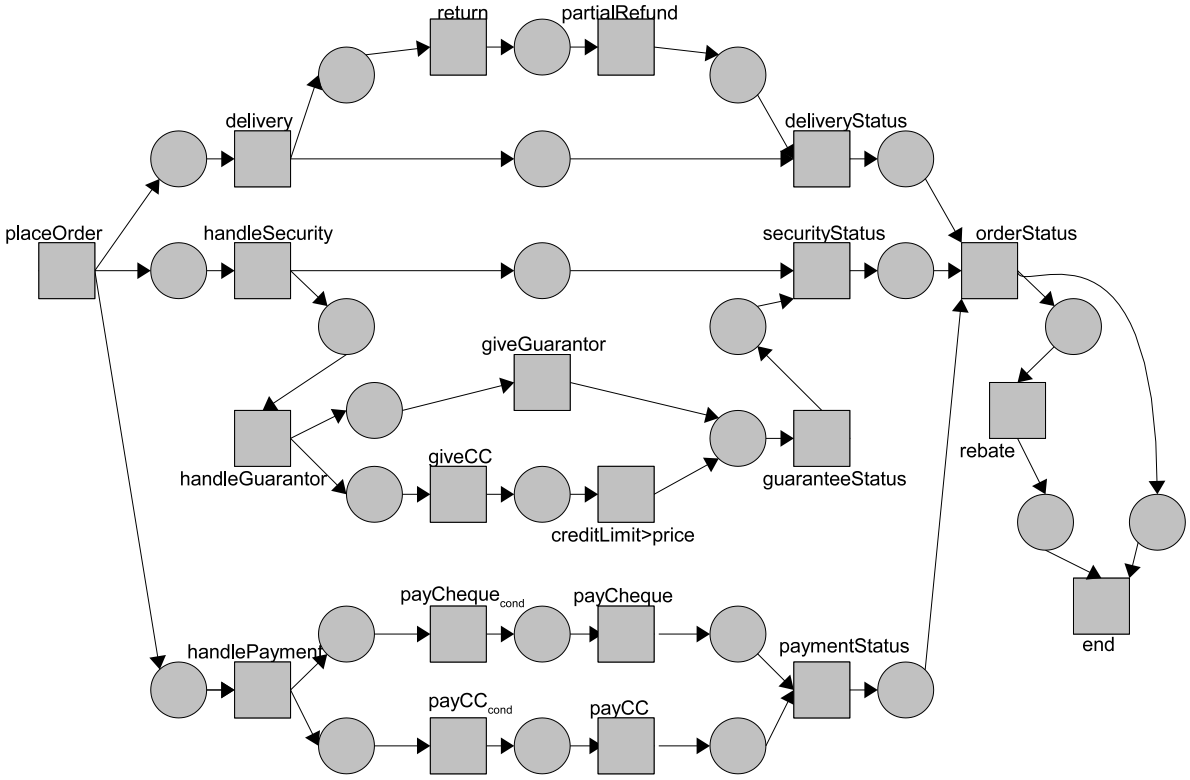
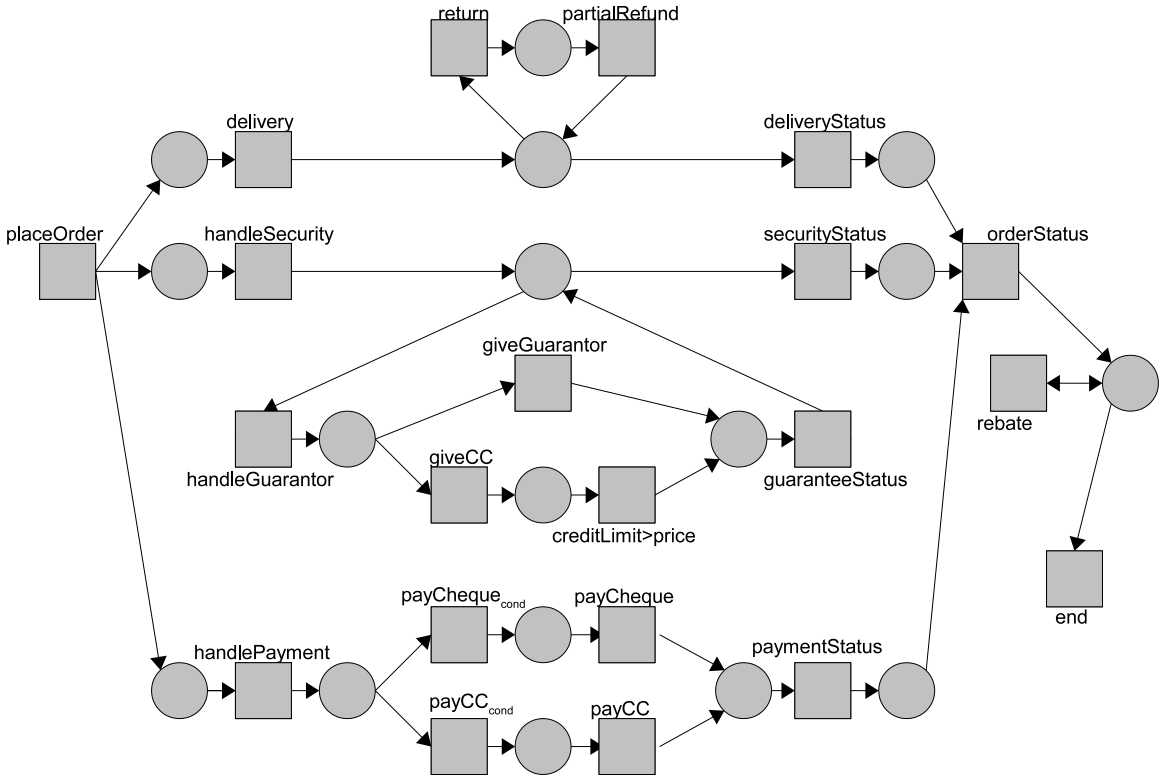


Abb. 4.18: Ein Kontrollflussgraph

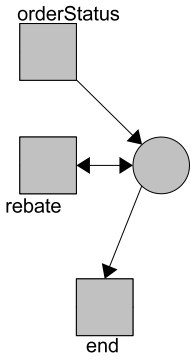
gerichteten Kanten (Pfeile) zwischen diesen Knoten. Bei der Übertragung in Petri-Netze werden zunächst diese Knoten ebenso wie evtl. vorhandene Bedingungen (z. B. $payCC_{cond}$) zu Transitionen. An den Kanten zwischen den Transitionen müssen noch Stellen eingefügt werden:



Sind die ausgehenden Kanten einer Aktion im KFG allerdings, wie bei Aktion *delivery*, durch ein *OR* miteinander verbunden, bedeutet dies, dass nur eine der nachfolgenden Aktionen ausgeführt wird. Somit stellt dies einen XODER-Split dar. Dort wo sich der Kontrollfluss wieder vereinigt, wie bei *deliveryStatus*, befindet sich der komplementäre XODER-Join. Dies muss durch das XODER-Pattern modelliert werden, wodurch z. B. *delivery* nur noch eine Nachfolgestelle und *deliveryStatus* nur noch eine Stelle im Vorbereich erhält:



Um evtl. entstandene Zykel wie



zu entfernen, werden an solchen Stellen zusätzliche Stellen und Transitionen, die den leeren Prozess $\perp$ darstellen sollen, eingefügt, um diese Zykel wieder aus dem Netz zu entfernen. So erhält man schließlich die in Abb. 4.19 gegebene p/t-Netz-Darstellung $\mathcal{N}_{KFG}$ des Kontrollflussgraphen. Hier wurden zusätzlich noch Buchstaben zu jeder Transition hinzugefügt, die im weiteren Verlauf diese Transition bezeichnen sollen. Dabei entspricht z. B. die Transition a der Aktion *placeOrder*.

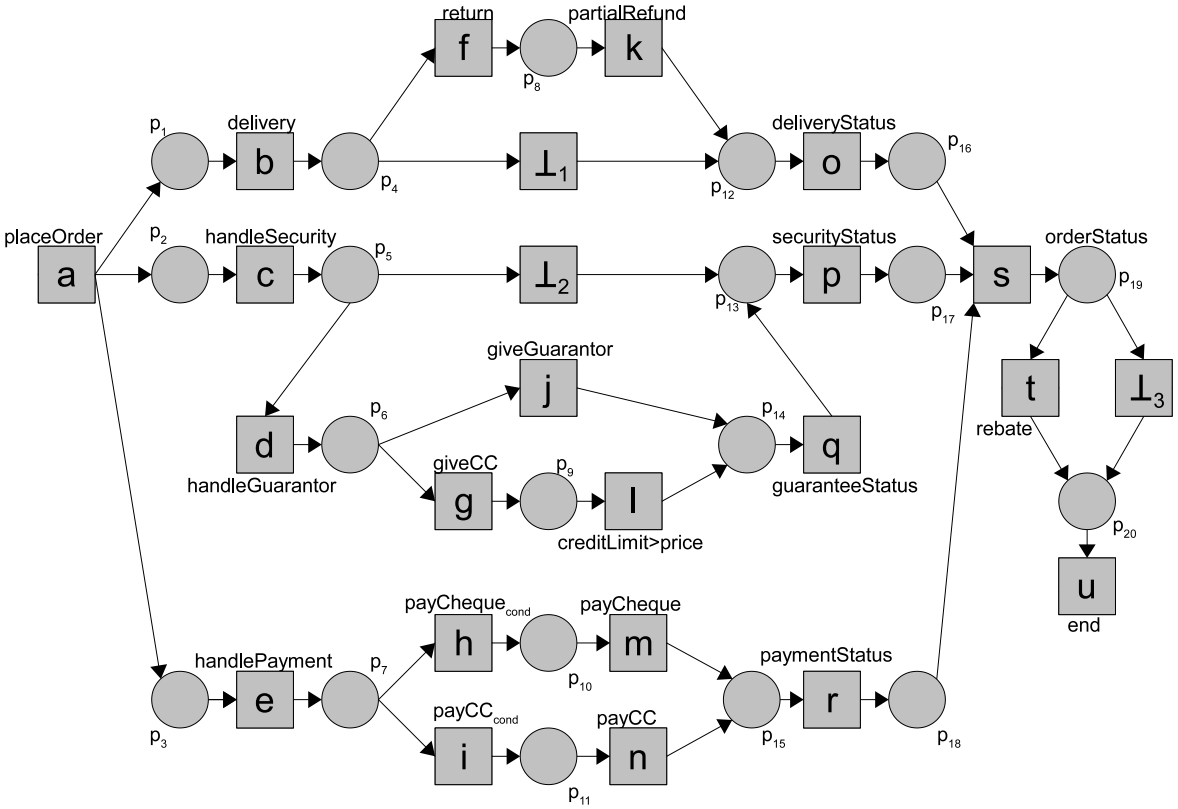
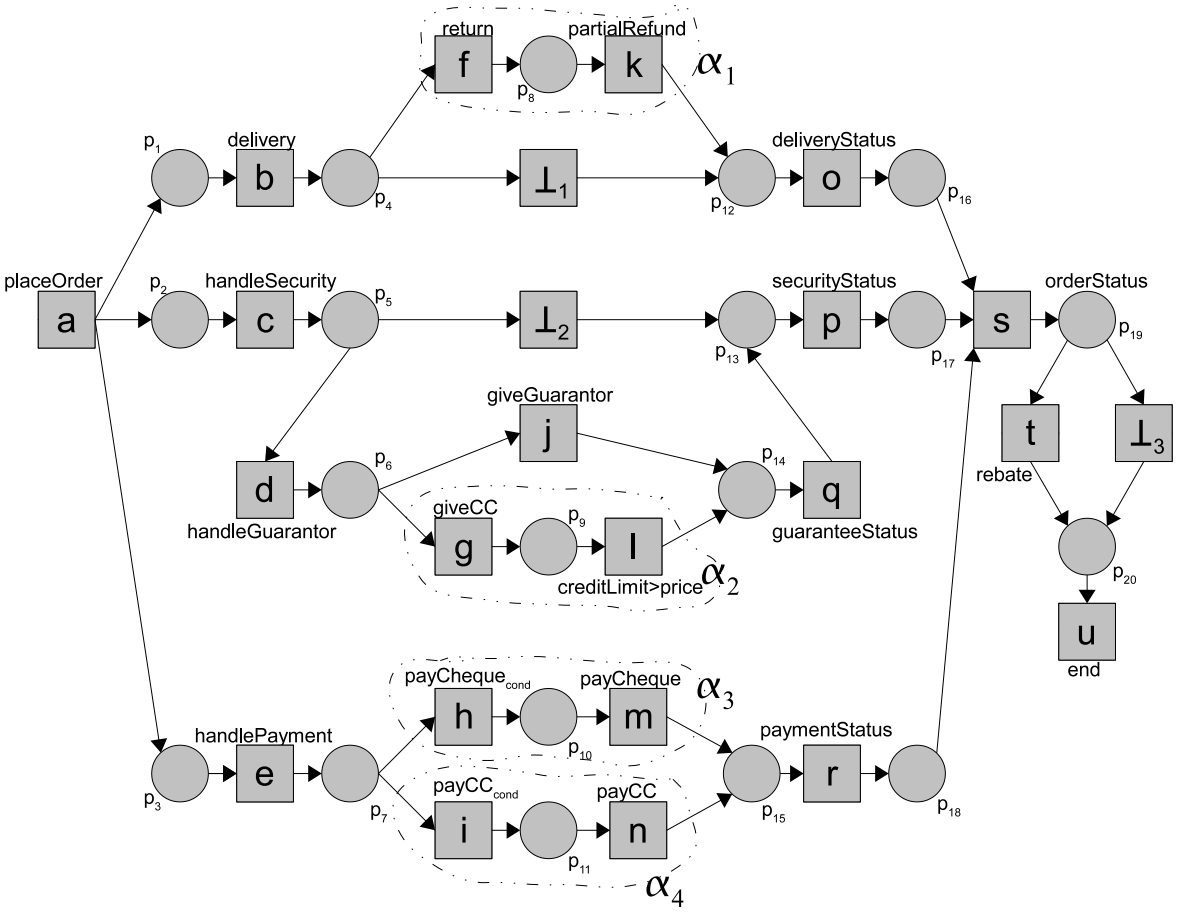


Abb. 4.19: p/t-Netz-Darstellung des KFG

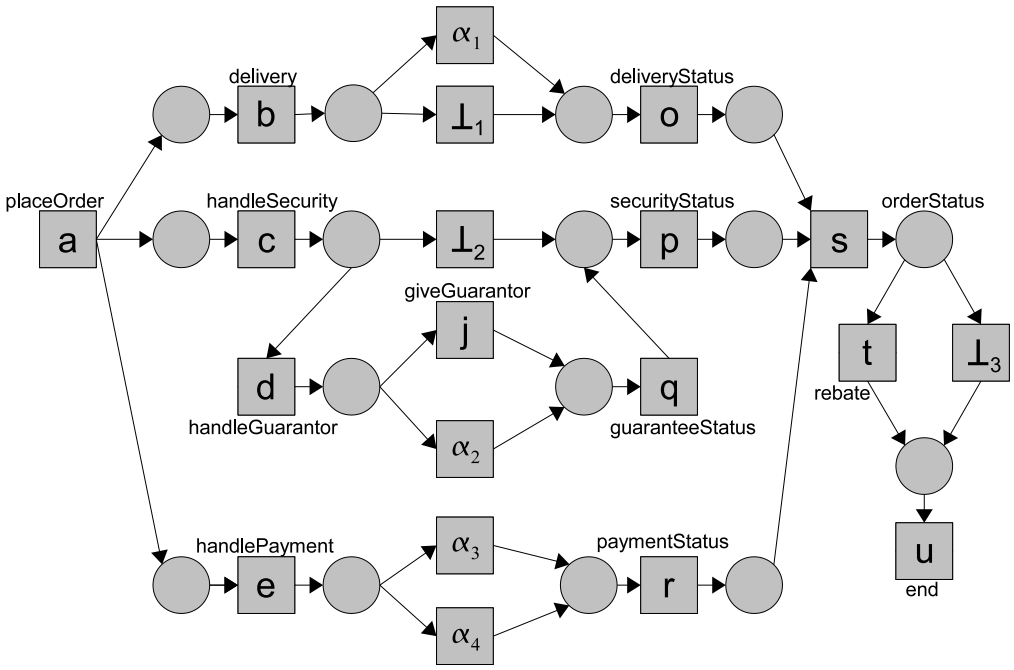
Bei der Bottom-up-Suche nach Pattern man zunächst einige Sequenzen zweier Aktionen wie z. B. f und k.



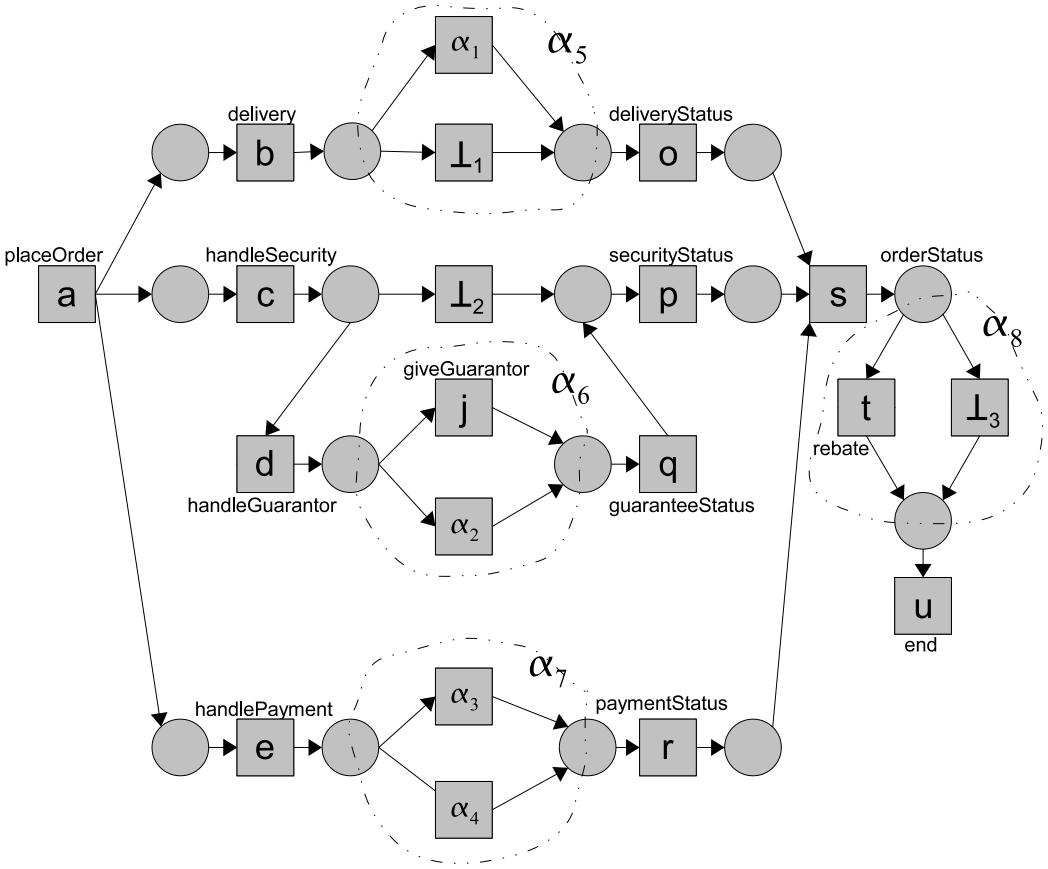
Somit erhält man für diese Teile des Netzes die aktionslogischen Implementierungen

$$\alpha_1 = [f \boxleftarrow k], \quad \alpha_2 = [g \boxleftarrow l], \quad \alpha_3 = [h \boxleftarrow m] \quad \text{und} \quad \alpha_4 = [i \boxleftarrow n].$$

Um die folgenden Schritte im Netz übersichtlicher gestalten zu können, wird der Teil von $\mathcal{N}$, der bereits durch ein Modul α_i implementiert wurde, jeweils durch eine Transition α_i ersetzt. Dies ist in der nächsten Abbildung für α_1 bis α_4 getan worden.



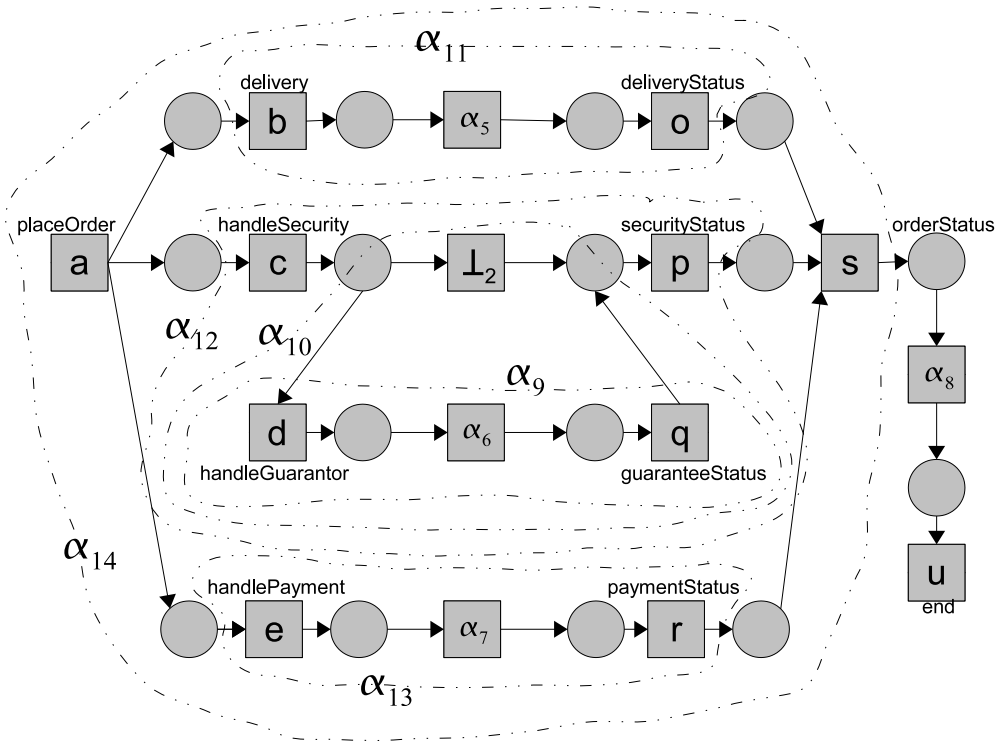
Danach identifiziert man die folgenden XODER-Pattern α_5 bis α_8



und bekommt die Module

$$\alpha_5 = [\alpha_1 \boxplus \bot_1], \quad \alpha_6 = [j \boxplus \alpha_2], \quad \alpha_7 = [\alpha_3 \boxplus \alpha_4] \quad \text{und} \quad \alpha_8 = [t \boxplus \bot_3].$$

Zuletzt werden noch die Pattern α_9 bis α_{14} erkannt,



die durch die folgenden Module

$$\alpha_9 = [d \boxleft \alpha_6 \boxleft q], \quad \alpha_{10} = [\bot_2 \boxplus \alpha_9], \quad \alpha_{11} = [b \boxleft \alpha_5 \boxleft o],$$

$$\alpha_{12} = [c \boxleft \alpha_{10} \boxleft p], \quad \alpha_{13} = [e \boxleft \alpha_7 \boxleft r] \text{ und } \alpha_{14} = [\alpha_{11} \boxwedge \alpha_{12} \boxwedge \alpha_{13}]$$

implementiert werden. Somit ist die aktionslogische Implementierung des gesamten Netzes

$$\mathcal{M}_\mathcal{N} = [\alpha_{14} \boxleft \alpha_8 \boxleft u].$$

In diese Formel lassen sich jetzt die einzelnen α_i wieder einsetzen:

$$\begin{aligned}
\mathcal{M}_\mathcal{N} &= [\alpha_{14} \boxleft \alpha_8 \boxleft u] \\
&= [[\alpha_{11} \boxwedge \alpha_{12} \boxwedge \alpha_{13}] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u] \\
&= [[b \boxleft \alpha_5 \boxleft o] \boxwedge [c \boxleft \alpha_{10} \boxleft p] \boxwedge [e \boxleft \alpha_7 \boxleft r]] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u] \\
&= [[b \boxleft [\alpha_1 \boxplus \bot_1] \boxleft o] \boxwedge [c \boxleft [\bot_2 \boxplus \alpha_9] \boxleft p] \boxwedge [e \boxleft [\alpha_3 \boxplus \alpha_4] \boxleft r]] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u] \\
&= [[b \boxleft [[f \boxleft k] \boxplus \bot_1] \boxleft o] \boxwedge [c \boxleft [\bot_2 \boxplus [d \boxleft \alpha_6 \boxleft q]] \boxleft p] \\
&\quad \boxwedge [e \boxleft [[h \boxleft m] \boxplus [i \boxleft n]] \boxleft r]] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u] \\
&= [[b \boxleft [[f \boxleft k] \boxplus \bot_1] \boxleft o] \boxwedge [c \boxleft [\bot_2 \boxplus [d \boxleft [j \boxplus \alpha_2] \boxleft q]] \boxleft p] \\
&\quad \boxwedge [e \boxleft [[h \boxleft m] \boxplus [i \boxleft n]] \boxleft r]] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u] \\
&= [[b \boxleft [[f \boxleft k] \boxplus \bot_1] \boxleft o] \boxwedge [c \boxleft [\bot_2 \boxplus [d \boxleft [j \boxplus [g \boxleft l]] \boxleft q]] \boxleft p] \\
&\quad \boxwedge [e \boxleft [[h \boxleft m] \boxplus [i \boxleft n]] \boxleft r]] \\
&\quad \boxleft [t \boxplus \bot_3] \boxleft u]
\end{aligned}$$

Nach Erstellen dieser $\mathcal{LA}$-Formel soll der Kontrollflussgraph als $\mathcal{TR}$-Programm $Prog_{\mathcal{TR}_\mathcal{N}} = (\mathcal{B}_\mathcal{N}, P_\mathcal{N}, D_\mathcal{N})$ implementiert werden. Zuerst wird das $\mathcal{LA}$-Modul $\mathcal{M}_\mathcal{N}$ in die entsprechende Transaktionsbasis $P'_\mathcal{N}$ transformiert:

$$\begin{aligned}
P'_\mathcal{N} = \{ \quad (KFG \leftarrow \quad & (((b \otimes ((f \otimes k) \boxplus \bot_1) \otimes o) \wedge (c \otimes (\bot_2 \boxplus (d \otimes (j \boxplus (g \otimes l)) \otimes q)) \otimes p) \\
& \wedge (e \otimes ((h \otimes m) \boxplus (i \otimes n)) \otimes r)) \\
& \otimes (t \boxplus \bot_3) \otimes u))\}
\end{aligned}$$

Die $\bot_i$ werden dabei in der $\mathcal{TR}$ als normale Transitionen behandelt. Auch sollen wieder Subformeln verwendet werden. So entsteht aus $P'_\mathcal{N}$ die endgültige Transaktionsbasis

$$\begin{aligned}
P_\mathcal{N} = \quad \{ \quad & ((KFG \leftarrow ((regel_1 \wedge regel_2 \wedge regel_3) \otimes regel_4)), \\
& (regel_1 \leftarrow (b \otimes ((f \otimes k) \boxplus \bot_1) \otimes o)), \\
& (regel_2 \leftarrow (c \otimes (\bot_2 \boxplus (d \otimes (j \boxplus (g \otimes l)) \otimes q)) \otimes p)), \\
& (regel_3 \leftarrow (e \otimes ((h \otimes m) \boxplus (i \otimes n)) \otimes r)), \\
& (regel_4 \leftarrow ((t \boxplus \bot_3) \otimes u))\}.
\end{aligned}$$

In die Transitionsbasis $\mathcal{B}_\mathcal{N}$ wird für jede Transaktion des Netzes eine elementare Transition eingefügt, sodass

$$\begin{aligned}\mathcal{B}_\mathcal{N} = \{\ & (\langle D, D \cup \{p_1, p_2, p_3\}\rangle a), (\langle D \cup \{p_1\}, D \cup \{p_4\}\rangle b),\\ & (\langle D \cup \{p_2\}, D \cup \{p_5\}\rangle c), (\langle D \cup \{p_5\}, D \cup \{p_6\}\rangle d),\\ & (\langle D \cup \{p_3\}, D \cup \{7\}\rangle e), (\langle D \cup \{p_4\}, D \cup \{p_8\}\rangle f),\\ & (\langle D \cup \{p_6\}, D \cup \{p_9\}\rangle g), (\langle D \cup \{p_7\}, D \cup \{p_{10}\}\rangle h),\\ & (\langle D \cup \{p_7\}, D \cup \{p_{11}\}\rangle i), (\langle D \cup \{p_6\}, D \cup \{p_{14}\}\rangle j),\\ & (\langle D \cup \{p_8\}, D \cup \{p_{12}\}\rangle k), (\langle D \cup \{p_9\}, D \cup \{p_{14}\}\rangle l),\\ & (\langle D \cup \{p_{10}\}, D \cup \{p_{15}\}\rangle m), (\langle D \cup \{p_{11}\}, D \cup \{p_{15}\}\rangle n),\\ & (\langle D \cup \{p_{12}\}, D \cup \{p_{16}\}\rangle o), (\langle D \cup \{p_{15}\}, D \cup \{17\}\rangle p),\\ & (\langle D \cup \{p_{14}\}, D \cup \{p_{13}\}\rangle q), (\langle D \cup \{p_{15}\}, D \cup \{p_{16}\}\rangle r),\\ & (\langle D \cup \{p_{16}, p_{17}, p_{18}\}, D \cup \{p_{19}\}\rangle s), (\langle D \cup \{p_{19}\}, D \cup \{p_{20}\}\rangle t),\\ & (\langle D \cup \{p_{20}\}, D\rangle u), (\langle D \cup \{p_4\}, D \cup \{p_{12}\}\rangle \perp_1),\\ & (\langle D \cup \{p_5\}, D \cup \{p_{15}\}\rangle \perp_2), (\langle D \cup \{p_{19}\}, D \cup \{p_{20}\}\rangle \perp_3)\}.\end{aligned}$$

Zuletzt wird noch die Datenbasis $D_\mathcal{N}$ aufgrund der leeren Markierung implementiert als $D_\mathcal{N} = \{\}$.

□

Nach diesen beiden Beispielen werden im folgenden und letzten Abschnitt die Ergebnisse dieser Arbeit noch einmal abschließend zusammengefasst.

5 Fazit und Ausblick

In dieser Arbeit wurde gezeigt, wie die Netzdarstellung eines Workflows in eine aktionslogische, prädikatenlogische, transaktionslogische oder aktionslogische Formel umgeformt werden kann. Ebenso wurde der umgekehrte Weg beschrieben. So ist es jetzt möglich, Workflows auch mit Hilfe der verschiedenen Inferenztechniken der einzelnen Logiken zu untersuchen. Bei der Transformation wurden allerdings einige Einschränkungen getroffen. So wurde auf die Transformation von Quantoren sowie auf Zykel in den zu transformierenden Workflow-Netzen verzichtet. Außerdem wurden die möglichen Kantenlabels von pr/t-Netzen zur Umwandlung in eine prädikatenlogische Darstellung auf einzelne Tupel beschränkt. Die Überwindung dieser Einschränkungen wie auch die Einbindung von globalen Constraints, wie in [DaKiRaRa98] bereits für KFGs und die $\mathcal{TR}$ vorgestellt, kann allerdings durchaus Gegenstand von folgenden Arbeiten sein.

Außerdem kann auf der Grundlage dieser theoretischen Arbeit ein Computerprogramm implementiert werden, das die hier vorgestellten Techniken implementiert und somit automatisiert. Auch dies kann Gegenstand einer anschließenden Arbeit sein.

Literaturverzeichnis

[AaHoKiBa03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski und A.P. Barros: *Workflow patterns.* Distributed and Parallel Databases, 14(3): 5-51, 2003

[Aals98] W.M.P. van der Aalst: *The Application of Petri Nets toWorkflow Management.* The Journal of Circuits, Systems and Computers, 8(1):21–66, 1998

[AaHo05] W.M.P. van der Aalst and A.H.M. ter Hofstede: *YAWL: Yet another workflow language.* Information Systems, 30(4): 245-275, 2005

[BoKi92] A. J. Bonner und M. Kifer: *Transaction logic programming.* Technical Report CSRI-270, University of Toronto, April 1992. Überarbeitet: Dezember 1992. (Dieser Bericht kann über anonymes FTP von cs.sunysb.edu heruntergeladen werden; Datei: pub/TechReports/kifer/transaction-logic.dvi.Z)

[BoKi94] A. J. Bonner und M. Kifer: *An Overview of Transaction Logic.* in *Theoretical Computer Science*, Bd. 133, S. 205-265, Oktober 1994

[BoKi96] A. J. Bonner und M. Kifer: *Concurrency and Communication in Transaction Logic.* in *Joint Intl. Conference and Symposium on Logic Programming.* S. 142-156, Bonn, September 1996, MIT Press

[DaKiRaRa98] H. Davulcu, M. Kifer, C. R. Ramakrishnan, I. V. Ramakrishnan: *Logic Based Modeling and Analysis of Workflows.*, in Proc. ACM PODS, S. 25-33, 1998

[Fide93] M. Fidelak: *Integritätsbedingungen in Petri-Netzen.* Koblenz, Landau, Univ. Diss., 1993

[Fuhr03] U. Fuhrbach: *Logic.* Skrip Koblenz, 2003

[Genr73] H. J. Genrich: *Formale Eigenschaften des Entscheidens und Handelns.* Interner Bericht 09/73-11-29, GMD, St. Augustin, 1973

[Laut97] K. Lautenbach: *Action logical correctness proving.* MATHMOD, Wien, 1997

[Laut01] K. Lautenbach: *Reproducibility of the Empty Marking.* Fachberichte Informatik, Universität Koblenz-Landau, 2001

[Laut02-1] K. Lautenbach: *Logical Reasoning and Petri Nets.* Fachberichte Informatik, Universität Koblenz-Landau, 2002

[Laut02-2] K. Lautenbach: *Petri Nets.* Köln 2002

[Laut06] K. Lautenbach: *Discrete Systems.* Koblenz 2007

[Petr62] C. A. Petri: *Kommunikation mit Automaten.* Technical Report, Schriften des Instituts für instrumentelle Mathematik, Bonn 1962

[RoKiFe] D. Roman, M. Kifer und D. Fensel: *WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic.*

[RuHoAaMu] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, N. Mulyar: *Workflow Control-Flow Patterns - A Revised View*

[Simo01] C. Simon: *A Logic of Actions and Its Application to the Development of Programmable Controllers.* Koblenz, Landau, Univ. Diss., 2001