Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Mathias Blumreiter

Algorithmus zum nichtdeterministischen Matching in rekonfigurierbaren Petrinetzen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Mathias Blumreiter

Algorithmus zum nichtdeterministischen Matching in rekonfigurierbaren Petrinetzen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Julia Padberg
Zweitgutachter: Prof. Dr. rer. nat. Christoph Klauck

Eingereicht am: 17. April 2013

Mathias Blumreiter

Thema der Arbeit

Algorithmus zum nichtdeterministischen Matching in rekonfigurierbaren Petrinetzen

Stichworte

rekonfigurierbare Petrinetze, dekorierte Petrinetze, S/T-Netz, Regeln, Matchingalgorithmus, Nichtdeterminismus, Simulation, Graphtransformation, Regelanwendung, PNVF2, ReConNet

Kurzzusammenfassung

Bei der Modellierung und der Simulation des Verhaltens dynamischer Systeme müssen neben ihrem Ablauf auch die Veränderungen am System im Modell abgebildet werden. Daher bestehen rekonfigurierbare Petrinetze aus einem dekorierten Petrinetz und einer Menge von Regeln, die das Petrinetz bei Bedarf flexibel anpassen. Vor einer Regelanwendung wird zuerst das zu transformierende Teilnetz bestimmt. Der dabei zu findende Match muss sowohl die Anforderungen aus der Sicht der Regelanwendung als auch der Simulation erfüllen. Um die Suche effizient zu gestalten, wird in dieser Arbeit ein Algorithmus zum nichtdeterministischen Matching in rekonfigurierbaren Petrinetzen vorgestellt. Die bei einer Graphtransformation einzuhaltenden Klebebedingungen und negativen Anwendungsbedingungen werden direkt in den Matchingprozess einbezogen. Durch seinen nichtdeterministischen Ablauf sorgt er außerdem dafür, dass bei einer Simulation in jedem Transformationsschritt ein unterschiedlicher Match gefunden werden kann. Die Wahrscheinlichkeit der einzelnen Matches hängt dabei von der Struktur der Petrinetze ab. Abschließend werden die Implementierung des Algorithmus in dem Modellierungswerkzeug ReConNet und die daraus resultierenden Konsequenzen beschrieben.

Mathias Blumreiter

Title of the paper

Algorithm for nondeterministic matching in reconfigurable Petri nets

Keywords

reconfigurable Petri nets, decorated Petri nets, P/T nets, rules, matching algorithm, nondeterminism, simulation, graph transformation, rule application, PNVF2, ReConNet

Abstract

When modelling and simulating the behaviour of dynamic systems it is important to represent their workflow and the system changes in the model. To that effect, reconfigurable Petri nets consist of a decorated Petri net and a set of rules which can flexibly modify the Petri net if required. Before a rule is applied, the transforming subnet must be determined first. The resulting match must meet all requirements of the rule application as well as the simulation. To make this search efficient, an algorithm is presented for the nondeterministic matching in reconfigurable Petri nets. Gluing conditions and negative application conditions which must be met during graph transformation are directly included in the matching process. Through its nondeterminism it also ensures that during simulation a different match can be found in every transformation step. The probability of an individual match is dependent on the structure of the Petri nets. Finally, the implementation of the algorithm in the modelling tool ReConNet and the resulting consequences are described.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Die Modellierung ist eine der wesentlichen Aufgaben in der Softwareentwicklung. Bei der Konzeption von Software, aber auch bei anderen Systemen, werden die einzelnen Anforderungen oder ihre Lösungen durch unterschiedliche Modelle dargestellt. Um die verschiedenen Eigenschaften des zu modellierenden Gegenstandes geeignet repräsentieren zu können, werden verschiedene Modellierungstechniken genutzt. Die Petrinetze sind eine dieser Techniken. Sie dienen unter anderem zur Abbildung des Verhaltens von dynamischen Systemen und von Prozessen. Seit ihrer Einführung durch Carl Petri [Pet62] hat sich eine große Vielfalt verschiedener Petrinetzvarianten herausgebildet. Eine dieser Varianten sind die dieser Arbeit zugrunde gelegten rekonfigurierbaren Petrinetze (siehe Abschnitt 2.1). Diese besitzen zusätzlich zu dem Schaltverhalten einfacher Netze noch Knotendekorationen. Außerdem können sie ihre Struktur durch die Anwendung von Regeln verändern. Dies versetzt sie in die Lage, sich dynamisch auf die Anforderungen des modellierten Systems einstellen zu können. Um die Modellierung einzelner Systeme und die darauffolgende Simulation und Analyse einfacher zu gestalten, wird an der Hochschule für Angewandte Wissenschaften Hamburg das Modellierungswerkzeug ReConNet entwickelt. Mit dessen Hilfe werden die verschiedenen Abläufe im Living Place Hamburg simuliert und analysiert.

1.1.1. Living Place Hamburg

Das Living Place Hamburg ist eine an der HAW Hamburg betriebene Modellwohnung zur Entwicklung von IT-basierten Konzepten für das moderne Wohnen [vgl. Luc10]. Dazu wurde auf dem Campus der HAW ein 140m² großes Appartement eingerichtet, das von verschiedenen Bewohnern unter Realbedingungen getestet wird. Die Aufenthalte der einzelnen Bewohner reichen dabei von einigen Stunden bis zu mehreren Tagen. Da die Wohnung auf die Bewohner reagiert und mit ihnen interagiert, müssen die möglichen Reaktionen zuvor definiert werden. Die Modellierung der einzelnen Abläufe

erfolgt hierbei mit dekorierten Petrinetzen (siehe Unterabschnitt 2.1.1). Im Rahmen einer Bachelorarbeit [Rei12] sind beispielsweise die morgendlichen Handlungsabläufe eines fiktiven Bewohners betrachtet worden. Da das Appartement auch auf die verschiedenen Vorlieben der einzelnen Bewohner eingehen kann, werden die Petrinetze dynamisch durch Regeln (siehe Unterabschnitt 2.1.3) verändert.

1.1.2. ReConNet

Der Aufbau der einzelnen Petrinetze und Regeln erfolgt in ReConNet direkt über eine grafische Benutzeroberfläche [vgl. Pad+12]. Dies erlaubt eine einfache Veränderung der Struktur der Netze. Außerdem kann das Entwicklungsteam des Living Place Hamburg auf diese Weise schnell unterschiedliche Szenarien modellieren. Bevor eine Simulation gestartet werden kann, müssen ein Ausgangsnetz und eine Menge von Regeln in das Werkzeug geladen werden. Da die Regeln nicht an die einzelnen Netze gebunden sind, können sie von Simulation zu Simulation variieren. Danach muss die Anzahl der zu vollziehenden Schritte und deren Art vorgegeben werden. Nachdem alle notwendigen Informationen vorliegen, kann die Simulation gestartet werden. ReConNet unterscheidet zwischen einer reinen Schalt-, einer Transformations- sowie einer kombinierten Simulation. Falls eine kombinierte Simulation aus Schaltvorgängen und Regelanwendungen gewählt wird, werden diese zufällig auf das Netz angewendet. Dadurch sollen die möglichen Abläufe der modellierten Szenarien betrachtet werden. Für das Entwicklungsteam des Living Place Hamburg ist beispielsweise interessant, ob eine bestimmte Kombination aus Schaltvorgängen und/oder Regelanwendung zu einem Deadlock innerhalb des Szenarios führt. Auch die Information, in wieweit die Netze ausarten, ist von Interesse. Um eine unerwünschte doppelte Regelanwendung zu vermeiden, wird ihre Anwendung in zukünftigen Versionen mit negativen Anwendungsbedingungen gesteuert (siehe Unterabschnitt 2.1.4 und Abschnitt 5.4).

1.2. Regelanwendung und das Matchingproblem

Damit eine Regel von ReConNet angewendet werden kann, muss zuerst ein Match zwischen dem linken Teil der Regel L und dem aktuellen Petrinetz N bekannt sein. Die übrigen Zuordnungen sind entweder durch den Regelaufbau statisch vorgeben oder ergeben sich aus dem aktuellen Match (siehe Abbildung 1.1).

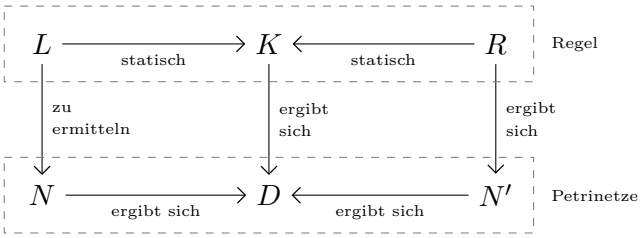


Abbildung 1.1.: Netztransformation in ReConNet

Das Matchingproblem besteht darin, wie dieser Match gezielt gefunden werden kann. Aufgrund des automatisierten Ablaufs der Simulation ist eine Einbeziehung des Benutzers in den Suchprozess ausgeschlossen.

1.3. Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung eines effizienten Algorithmus zum Matching rekonfigurierbarer Petrinetze. Dabei muss der Algorithmus verschiedene Anforderungen erfüllen. Diese betreffen zum einen die Regelanwendung und zum anderen die Simulation.

Aus der Sicht der Regelanwendung muss der Matchingalgorithmus gewährleisten, dass der gefundene Match die Klebebedingungen und die negativen Anwendungsbedingungen erfüllt. Damit die Regelanwendung in ReConNet effizient erfolgen kann, soll der Algorithmus diese Bedingungen direkt in den Matchingprozess mit einbeziehen. Des Weiteren soll der Algorithmus zum effizienten Matching einer Regeldatenbank geeignet sein. Bei der Anwendung von Regeln ist im vornherein meist nicht bekannt, ob für eine bestimmte Regel ein Match existiert oder nicht. Da schon das Matching einer einzelnen Regel aufwändig ist, soll der Algorithmus die Untersuchung der inkompatiblen Regeln vermeiden können.

Aufgrund seines Einsatzes in einer Simulationsumgebung sind die Korrektheit und die Vollständigkeit des Algorithmus zwingend. Außerdem sollen beim Matching verschiedene Matches gefunden werden können. Das bedeutet, dass er bei gleicher Ausgangslage unterschiedliche Matches finden können soll. Für die Verwertbarkeit der Simulationsergebnisse ist wichtig, dass jeder Match mit der gleichen Wahrscheinlichkeit getroffen werden kann. Um das zu erreichen, soll der Algorithmus sowohl die Fähigkeit besitzen, alle Matches zu generieren, als auch nichtdeterministisch einen einzelnen Match auszugeben.

Abschließend soll der entwickelte Algorithmus in ReConNet implementiert werden.

1.4. Abgrenzung des Themas

Bei der Suche nach Matches können mehrere Arten des Matchings unterschieden werden. Die Art der Abbildung teilt sie beispielsweise in injektive und nicht-injektive Matches ein [vgl. Ehr+07, S. 5 - 6]. Ein injektiver Match bildet eine Regel untergraphisomorph in das Petrinetz ab, während bei einem nicht-injektiven Match Knoten zusammenfallen können. Aufgrund der Komplexität des nicht-injektiven Matchings befasst sich diese Arbeit mit der effizienten Suche von injektiven Matches.

1.5. Aufbau der Arbeit

Kapitel 2 bildet eine Einführung in die theoretischen Grundlagen der rekonfigurierbaren Petrinetze. Die Anforderungen an den Matchingalgorithmus und die Auswahl des Ausgangsalgorithmus werden in Kapitel 3 beschrieben. Anschließend werden in Kapitel 4 die Adaption des Ausgangsalgorithmus auf die rekonfigurierbaren Petrinetze und die verschiedenen Eigenschaften des entstehenden Algorithmus erläutert. Die Darstellung der notwendigen Veränderungen bei der Implementierung in ReConNet erfolgt in Kapitel 5. Den Abschluss bildet Kapitel 6, welches den Algorithmus hinsichtlich der erreichten Ziele einordnet und Entwicklungsmöglichkeiten aufzeigt.

2. Grundlagen und Begriffe

2.1. Rekonfigurierbare Petrinetze

Rekonfigurierbare Petrinetze [EP03; LO04; Ehr+07; Pra+08; Pad12] sind eine mächtige Technik zur Modellierung von dynamischen Systemen und Prozessen. Bei der Erstellung eines Modells für einen spezifischen Prozess müssen zwei Arten von Veränderungen abgebildet werden. Zum einen ändert sich der Zustand durch den Prozessablauf und zum anderen kann sich der Prozess durch verschiedene Einwirkungen grundlegend verändern. Als Beispiel soll eine Fabrik mit mehreren Fertigungsstraßen dienen. Bei der Modellierung des Fertigungsprozesses müssen die einzelnen Maschinen, deren Belegung und die vonstatten gehende Fertigung abgebildet werden. Eine grundlegende Veränderung des Prozesses ist beispielsweise die Eröffnung einer neuen Fertigungsstraße. Um diese beiden Aspekte geeignet repräsentieren zu können, bestehen rekonfigurierbare Petrinetze aus einem dekorierten Petrinetz und einer Menge von Regeln. Der Ablauf und die daraus resultierenden Zustände werden durch das dekorierte Petrinetz dargestellt und die möglichen Anpassungen am Prozess durch die Regeln.

2.1.1. Dekorierte Petrinetze

Mit der Hilfe von Petrinetzen können nebenläufige und nichtdeterministische Abläufe modelliert werden [vgl. SS10, S. 232 ff]. In der Literatur werden für Petrinetze auch die Bezeichnungen Stellen-/Transitions-Netze und S/T-Netze verwendet. Allerdings genügen die einfachen Petrinetze nicht zur Modellierung der in Kapitel 1 erwähnten Szenarien. Um ihre Umsetzung dennoch zu ermöglichen und die Regelanwendung in ReConNet zu steuern, werden die einfachen Petrinetze um Dekorationen erweitert [vgl. Pad12, S. 4].

Aufbau

Ein Petrinetz ist ein gerichteter bipartiter Graph, der aus Stellen und Transitionen besteht. Die Kanten des Netzes werden in zwei Gruppen unterteilt, die jeweils aus der Sicht der Transition angegeben werden. Die bei einer Transition eingehenden Kanten

heißen Vorbereichskanten. Die ausgehenden Kanten werden als Nachbereichskanten bezeichnet. Um das Schalten der Netze zu ermöglichen, besitzt jede Knotenart eine eigene Aufgabe. Die Stellen dienen der Lagerung von Token, welche beim Schalten von den Transitionen konsumiert oder erzeugt werden. Die Anzahl der konsumierten oder erzeugten Token wird durch das Gewicht einer Vor- oder Nachbereichskante bestimmt. Die Summen der auf den Stellen liegenden Token bilden die Markierung des Netzes. Sie drückt einen Zustand des modellierten Systems aus [vgl. SS10, S. 232 ff]. Diese Definition eines einfachen Petrinetzes wird für die dekorierten Petrinetze um Kapazitäten und Namen für die Stellen sowie Namen, Labels und Labelerneuerungsfunktionen für die Transitionen erweitert.

$$N = (P, T, pre, post, m, cap, pname, tname, tlb, rnw)$$

Die Definition eines dekorierten Petrinetzes besteht aus einem 10-Tupel, dessen einzelne Elemente die folgende Bedeutung besitzen:

- P und T stehen für die Stellen und Transitionen. Durch die Bipartitheit des Netzes kann kein Knoten in beiden Mengen vertreten sein ($P \cap T = \emptyset$).
- Die beiden Funktionen *pre* und *post* repräsentieren die Vor- und Nachbereichskanten der Transitionen. Das Gewicht einer Kante bildet den Funktionswert.

  $$pre : P \times T \to \mathbb{N}_0 \qquad\qquad post : P \times T \to \mathbb{N}_0$$

  Falls zwischen zwei Knoten keine Kante besteht, ist der dazugehörige Wert 0.
- Die Markierung des Netzes wird durch die Abbildung $m : P \to \mathbb{N}_0$ angegeben. Dazu bildet m jede Stelle auf die Anzahl der auf ihr liegenden Token ab. Zur Begrenzung der maximalen Tokenanzahl je Stelle wird die Kapazitätsfunktion $cap : P \to \mathbb{N}$ verwendet.
- Die Namen der Stellen und Transitionen werden durch $pname : P \to A_P$ und $tname : T \to A_T$ bestimmt. Die Mengen A_P und A_T stehen für die Namensräume.
- Abschließend werden die Labels $tlb : T \to W$ und die Labelerneuerungsfunktionen $rnw : T \to END(W)$ angegeben. Dabei steht W für die Menge der möglichen Labels und $END(W)$ für die Menge der Endomorphismen auf W. Also der Menge der Abbildungen von W auf sich selbst. Die Labelerneuerungsfunktion einer spezifischen Transition lautet somit $rnw(t) : W \to W$ [vgl. Pad12, S. 4 - 5].

Beispiel und grafische Darstellung

Um die Definitionen zu verdeutlichen, werden in der Abbildung 2.1 zwei Beispielnetze dargestellt. Die beiden Petrinetze besitzen die gleiche Grundstruktur. Auf der linken Seite wird ein Netz in seiner einfachen Form aufgeführt und auf der rechten Seite das um Dekorationen ergänzte Netz. Die Stellen werden durch die blauen Kreise und die Transitionen durch die roten Rechtecke repräsentiert. Die schwarzen Kugeln auf den Stellen stehen für die Token. Die Summe der Token entspricht der Markierung.

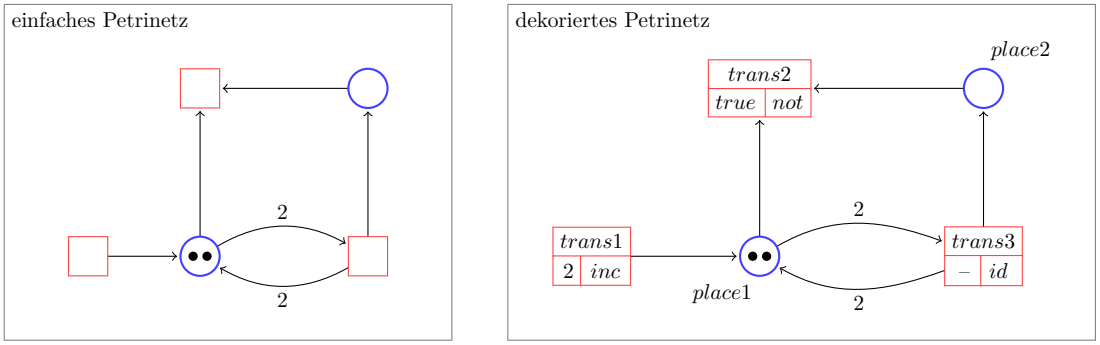


Abbildung 2.1.: Beispiel für ein Petrinetz in einfacher und dekorierter Form

Auf die Angabe der Kapazitäten wird hier verzichtet. Daher wird angenommen, dass jede Stelle beliebig viele Token aufnehmen kann. Die Namen der Stellen stehen direkt über oder unter ihnen. Die Dekorationen der Transitionen befinden sich in den roten Rechtecken. Im oberen Teil steht der Name, links darunter das Label und rechts die Labelerneuerungsfunktion der Transition. In diesem Beispiel werden die Funktionen $inc(x) = x + 1$ (Zähler), $id(x) = x$ (Identität) und $not : true \mapsto false, false \mapsto true$ (Schalter) verwendet. Somit lauten die einzelnen Elemente des dekorierten Petrinetzes $N = (P, T, pre, post, m, cap, pname, tname, tlb, rnw)$ aus Abbildung 2.1:

$P = \{p_1, p_2\}$	$T = \{t_1, t_2, t_3\}$	$pname:$	$pre:$	$post:$
$m: p_1 \mapsto 2$	$cap: p_1 \mapsto \infty$	$p_1 \mapsto place1$	$(p_1, t_1) \mapsto 0$	$(p_1, t_1) \mapsto 1$
$p_2 \mapsto 0$	$p_2 \mapsto \infty$	$p_2 \mapsto place2$	$(p_2, t_1) \mapsto 0$	$(p_2, t_1) \mapsto 0$
$tname:$	$tlb:$	$rnw:$	$(p_1, t_2) \mapsto 1$	$(p_1, t_2) \mapsto 0$
$t_1 \mapsto trans1$	$t_1 \mapsto 2$	$t_1 \mapsto inc(tlb(t_1))$	$(p_2, t_2) \mapsto 1$	$(p_2, t_2) \mapsto 0$
$t_2 \mapsto trans2$	$t_2 \mapsto true$	$t_2 \mapsto not(tlb(t_2))$	$(p_1, t_3) \mapsto 2$	$(p_1, t_3) \mapsto 2$
$t_3 \mapsto trans3$	$t_3 \mapsto -$	$t_3 \mapsto id(tlb(t_3))$	$(p_2, t_3) \mapsto 0$	$(p_2, t_3) \mapsto 1$

Schaltverhalten

Petrinetze zeichnen sich durch die Möglichkeit des Schaltens und der daraus folgenden Markierungsveränderung aus. Auf diese Weise können die Abläufe von Prozessen auf eine intuitive Art abgebildet werden. Da für das Matching nur ein einfacher Schaltbegriff benötigt wird, wird auf die Darstellung der unterschiedlichen Schaltvarianten und ihrer Konsequenzen verzichtet. Für weiterführende Information zu dem Schaltverhalten rekonfigurierbarer Petrinetze sei unter anderem auf [Pad12] und [Ehr+07] verwiesen.

Das Schaltverhalten der Petrinetze wird in den Transitionen umgesetzt. Dazu konsumieren sie bestehende Token von den Vorbereichsstellen und erzeugen neue Token für die Nachbereichsstellen.

$$P_V : t \mapsto \{p \in P \mid pre(p,t) > 0\} \qquad P_N : t \mapsto \{p \in P \mid post(p,t) > 0\} \qquad \text{mit } t \in T$$

Eine Stelle befindet sich im Vorbereich P_V einer Transition, wenn sie eine Vorbereichskante mit der Transition verbindet. Die Stellen, die mit einer Nachbereichskante getroffen werden, bilden den Nachbereich. Bei einem Schaltvorgang werden so viele Token konsumiert oder erzeugt, wie das Gewicht der verbindenden Kante beträgt. Da die Gewichte höher ausfallen können als die Markierungen der Stellen, wird zwischen aktivierten und (unter dieser Markierung) nicht-aktivierten Transitionen unterschieden. Dabei heißt eine Transition aktiviert, wenn die Markierungen aller Stellen ihres Vorbereichs größer oder gleich den Gewichten der verbindenden Kanten sind. Die neue Markierung wird berechnet, indem zuerst von den Vorbereichsstellen die jeweilige Anzahl an Token abgezogen und danach die Tokenanzahl der Nachbereichsstellen erhöht wird. Bei dekorierten Petrinetzen muss ein Schaltvorgang außerdem die Kapazitätsgrenzen der Stellen beachten. Daher wird der Begriff der Schaltfähigkeit dahingehend erweitert, dass alle Nachbereichsstellen die entstehenden Token aufnehmen können müssen. Des Weiteren wird bei einem Schaltvorgang das Label der Transitionen erneuert. Dazu wird die Labelerneuerungsfunktion mit dem aktuellen Label aufgerufen und das Ergebnis als neues Label übernommen. Die Abbildung 2.2 zeigt die Ergebnisse dreier aufeinanderfolgender Schaltvorgänge zu dem in Abbildung 2.1 vorgestellten dekorierten Petrinetz.

1. **Schaltschritt (Transition t_1):** Da die Transition t_1 im Vorbereich keine Stellen besitzt, ist sie unter jeder Markierung aktiviert. Durch den Schaltvorgang wird die Anzahl der Token der Stelle p_1 um ein Token auf 3 erhöht. Außerdem hat sich das Label der Transition verändert. Vor dem Schaltschritt lautete es 2. Da t_1 als Labelerneuerungsfunktion den Zähler verwendet, lautet das neue Label 3.

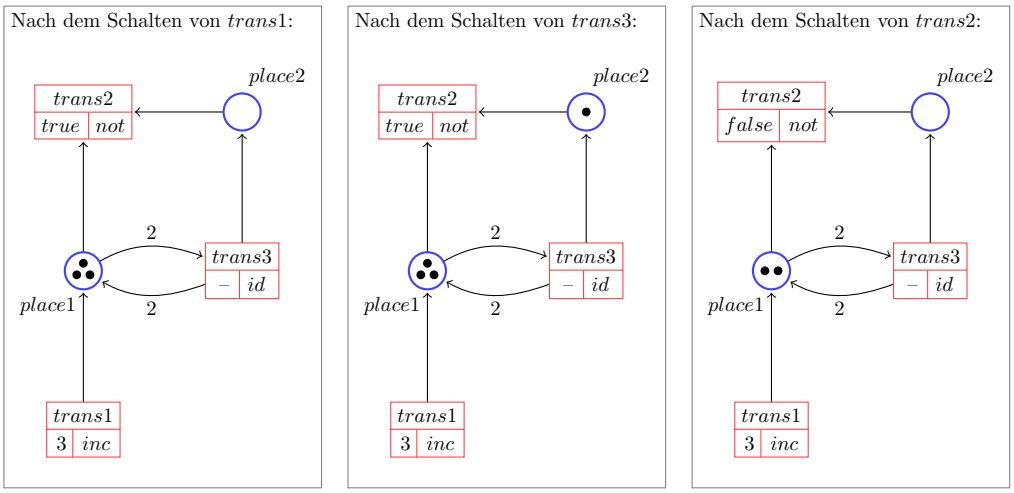


Abbildung 2.2.: Schaltverhalten dekorierter Petrinetze

2. Schaltschritt (Transition t_3): Im Gegensatz zu t_1 hat t_3 sowohl Stellen im Vor- als auch Nachbereich. Dabei lautet $P_V(t_3) = \{p_1\}$ und $P_N(t_3) = \{p_1, p_2\}$. Bevor die Transition geschaltet werden kann, muss sie aktiviert sein. Das betrifft die ausreichende Anzahl an Token auf den Vorbereichsstellen und die Einhaltung der Kapazitäten der Nachbereichsstellen. Im Vorbereich besitzt t_3 nur eine Stelle, wobei die Vorbereichskante ab p_1 das Gewicht 2 hat. Die Markierung von p_1 lautet nach dem ersten Schaltschritt 3 (siehe linkes Netz aus Abbildung 2.2). Damit erfüllen alle Vorbereichsstellen die Bedingungen zur Schaltfähigkeit. Da bei diesem Netz die Stellen beliebig viele Token aufnehmen können, kann kein Kapazitätslimit verletzt werden. Somit ist t_3 nach dem ersten Schaltschritt schaltfähig.

Bei dem Schalten wird die Markierung von p_2 um 1 erhöht (siehe mittleres Netz aus Abbildung 2.2). Die Markierung von p_1 bleibt unverändert erhalten, da diese Stelle sowohl im Vor- als auch Nachbereich von t_3 liegt und die verbindenden Kanten das gleiche Gewicht besitzen. Wegen der Identität als Labelerneuerungsfunktion bleibt das Label von t_3 ebenso unverändert erhalten.

3. Schaltschritt (Transition t_2): Durch die beiden vorherigen Schaltvorgänge wird die Schaltfähigkeit von t_2 hergestellt (siehe mittleres Netz aus Abbildung 2.2). Als Ergebnis des Schaltschrittes von t_2 hat sich dessen Label von $true$ zu $false$ verändert und die Markierungen von p_1 und p_2 jeweils um 1 verringert (siehe rechtes Netz aus Abbildung 2.2).

2.1.2. Netzmorphismen und Match

Ein Netzmorphismus f ist ein Paar von Abbildungen zwischen den Stellen und Transitionen zweier Petrinetze N_1 und N_2. Dabei existieren verschiedene Arten von Morphismen. Unterschieden wird beispielsweise zwischen den injektiven und den nicht-injektiven Morphismen [vgl. Ehr+07, S. 5 - 6]. In dieser Arbeit wird die Menge der injektiven Netzmorphismen betrachtet.

$$\begin{aligned} f &= (f_P : P_1 \to P_2, f_T : T_1 \to T_2) \\ N_{i\in\{1,2\}} &= (P_i, T_i, pre_i, post_i, m_i, cap_i, pname_i, tname_i, tlb_i, rnw_i) \quad \text{über } A_P,\ A_T,\ W \end{aligned}$$

Ein Paar von Knotenabbildungen heißt Morphismus, wenn es für alle Stellen und Transitionen von N_1 die folgenden Bedingungen erfüllt [vgl. Pad12, S. 5 - 6]:

- Der Vor- und Nachbereich der Transitionen muss hinsichtlich der Kanten und ihrer Gewichte exakt erhalten bleiben. Das bedeutet, dass für jede Kante zwischen den Knoten von N_1 eine entsprechende Kante in N_2 existieren muss und die getroffenen Transitionen keine zusätzliche Kanten besitzen dürfen.

  $$\begin{aligned} pre_1(p_1, t_1) = pre_2(f_P(p_1), f_T(t_1)) \quad &\wedge \quad |P_V(t_1)| = |P_V(f_T(t_1))| \\ &\wedge \quad |P_N(t_1)| = |P_N(f_T(t_1))| \end{aligned}$$

  Ein Morphismus erhält somit die Struktur des Quellnetzes im Zielnetz. Eine getroffene Stelle $f_P(p_1)$ darf allerdings zusätzliche Nachbarn aufweisen. Die Strukturerhaltung entspricht daher einer erweiterten Form der Untergraphisomorphie.

- Zusätzlich zur Struktur muss die Aktivierung der Transitionen bewahrt werden. Daher dürfen die getroffenen Stellen nicht weniger Token besitzen.

  $$m_1(p_1) \leq m_2(f_P(p_1)) \quad \wedge \quad cap_1(p_1) = cap_2(f_P(p_1)) \tag{2.1}$$

  Allerdings kann eine nicht-aktivierte Transition durchaus auf eine aktivierte abgebildet werden.

- Abschließend müssen die restlichen Dekorationen übereinstimmen:

  $$\begin{aligned} pname_1(p_1) &= pname_2(f_P(p_1)) & tname_1(t_1) &= tname_2(f_T(t_1)) \\ tlb_1(t_1) &= tlb_2(f_T(t_1)) & rnw_1(t_1) &= rnw_2(f_T(t_1)) \end{aligned}$$

Um diese Anforderungen zu verdeutlichen, sollen die Beispiele in Abbildung 2.3 und Abbildung 2.4 dienen. Die erste Abbildung konzentriert sich auf die Darstellung der Strukturerhaltung des Morphismus. Dazu wird auf die Wiedergabe der Dekorationen verzichtet. Die Beispiele in der zweiten Abbildung haben wiederum in jedem Fall einen strukturellen Morphismus als Grundlage. Sie sollen die verschiedenen Abbildungsarten der Knotendekorationen verdeutlichen.

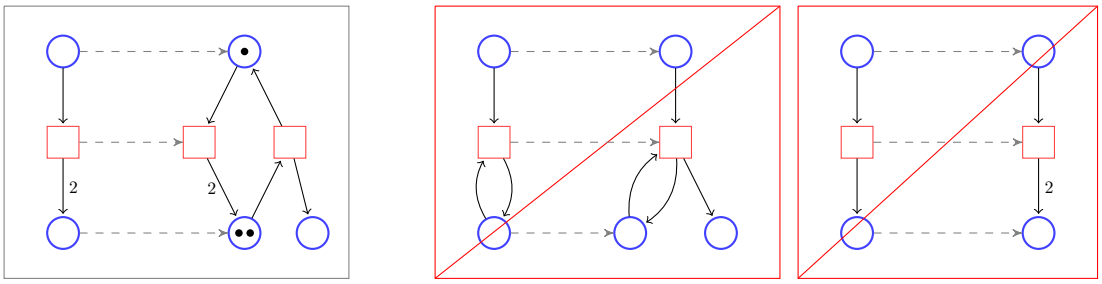


Abbildung 2.3.: Beispiele für einen strukturell gültigen und zwei ungültige Morphismen

Da Stellen im Zielnetz zusätzliche Nachbarn aufweisen dürfen, ist das erste Beispiel von Abbildung 2.3 ein Morphismus. Im mittleren Beispiel besitzt die Zieltransition einen überzähligen Nachbarn und im rechten Beispiel eine falsch gewichtete Kante. Daher sind die beiden Paare von Abbildungen keine gültigen Netzmorphismen.

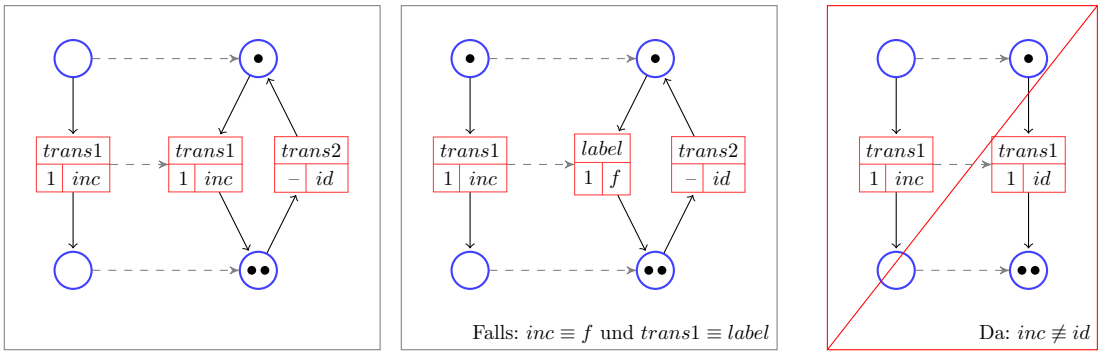


Abbildung 2.4.: Beispiele für Morphismen in dekorierten Petrinetzen

Die in Abbildung 2.4 aufgeführten Beispiele konzentrieren sich auf die Kompatibilität der Transitionsdekorationen. Das rechte Paar von Abbildungen ist offensichtlich kein Morphismus, da *inc* und *id* inkompatibel sind. Das mittlere Beispiel ist ein Morphismus unter der Bedingung, dass das undefinierte Label *label* und die Labelerneuerungsfunktion f gleich den Dekorationen der Quelltransition sind, was im linken Beispiel der Fall ist.

2.1.3. Regeln und die Klebebedingung

Neben einem dekorierten Petrinetz umfassen rekonfigurierbare Petrinetze eine Menge von Regeln. Mit ihnen werden die Netze zur Laufzeit verändert [vgl. Pad12, S. 7]. Ein Beispiel bilden die in Unterabschnitt 1.1.1 erwähnten Szenarien zur Modellierung des morgendlichen Handlungsablaufes eines fiktiven Bewohners im Living Place Hamburg. Das Basisnetz umfasst beispielsweise nur zwei Stellen, wobei die möglichen Aktionen des Bewohners durch die Regeln in das Netz eingefügt werden [vgl. Rei12, S. 45 ff.]. Auf diese Weise kann ein rekonfigurierbares Petrinetz flexibel auf die möglichen Abläufe reagieren, ohne von Anfang an ein riesiges Netz verwalten zu müssen. Neben der Fähigkeit, ein Netz zu vergrößern, können die Regeln auch die nicht mehr benötigten Netzteile entfernen.

Aufbau

Eine Regel besteht aus drei dekorierten Petrinetzen, die über strikte Morphismen verbunden werden. Ein Morphismus heißt strikt, wenn er injektiv ist und für die Übereinstimmung der Markierungen der Quell- und Zielstellen sorgt [vgl. Pad12, S. 6 - 7]. Da die vorgestellten Morphismen immer injektiv sind, muss nur die Tokenübereinstimmung der Netze sichergestellt werden. Dazu wird die Markierungsbedingung in Gleichung 2.1 entsprechend angepasst. Eine Regel hat die Form:

$$rule = (L \rightarrow K \leftarrow R)$$

Die linke Seite der Regel bildet das Netz L. Damit die Regel angewendet werden kann, muss L in einem gegebenen Petrinetz N gefunden werden. Die zu erzeugenden und entfernenden Stellen, Transitionen und Kanten werden durch die rechte Seite der Regel R bestimmt. Als Verbindung zwischen L und R dient das Klebenetz K. Dabei wird der von ReConNet verwendete [vgl. Pad+12, S. 3 ff.] und in [EHP09] beschriebene co-span Ansatz genutzt. Das bedeutet, dass K eine „Vereinigung“ von L und R darstellt. K enthält somit nur die Stellen, Transitionen und Kanten, die von wenigstens einem der beiden strikten Morphismen $L \rightarrow K$ und $R \rightarrow K$ getroffen werden.

Regelanwendung

Durch die Anwendung einer Regel wird ein Ausgangsnetz N in ein neues Netz N' transformiert. Die Zusammenhänge zwischen den einzelnen Regelteilen und den dabei entstehenden Petrinetzen werden in der Abbildung 2.5 dargestellt.

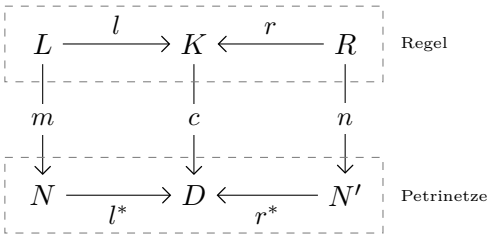


Abbildung 2.5.: Zusammenhänge bei einer Regelanwendung

Bei einer Regelanwendung werden sechs Netze betrachtet, die durch Netzmorphismen verbunden werden. Im oberen Bereich werden die verschiedenen Regelteile aufgeführt und darunter die dazugehörigen Petrinetze. Die vier Matches l, l^*, r und r^* sind strikte Morphismen, wohingegen m, c und n normale Morphismen sind. Mit der Hilfe der einzelnen Netze und Matches können die hinzuzufügenden, die zu erhaltenden und die zu löschenden Elemente bestimmt werden. Die Menge $K - L$ enthält die Stellen, Transitionen und Kanten, die von der Regel angelegt werden, wohingegen $K - R$ die zu löschenden Elemente beinhaltet. Die von der Regel zu erhaltenden Elemente werden durch $L \cap R \subseteq K$ repräsentiert. Die Transformation eines Netzes erfolgt in drei Schritten:

1. Schritt Als Erstes muss ein normaler Match m von L nach N gefunden werden. Dieser bildet die Basis für die Regelanwendung.

2. Schritt Danach werden die Elemente aus $K - L$ zu N hinzugefügt. Das Ergebnis ist das Zwischennetz D. Dabei bestimmen die beiden Matches m und l, welche Knoten durch etwaige neue Kanten verbunden werden.

3. Schritt Abschließend werden aus D alle Knoten und Kanten von $K - R$ entfernt. Das Ergebnis der Netztransformation ist das Petrinetz N'. Dieses enthält neben den Elementen aus N, die nicht von m getroffen werden, die zu erhaltenden Elemente aus $L \cap R \subseteq K$ und die neuen Knoten und Kanten aus $K - L$.

Allerdings besteht bei der Regelanwendung ein Problem. Falls ein Knoten gelöscht wird, der mit einem von m nicht getroffenen Knoten verbunden ist, wird aus der Netztransformation kein gültiges Petrinetz resultieren. Um diesen Fall auszuschließen, muss m die Klebebedingung erfüllen.

Klebebedingung

Die Klebebedingung teilt sich in die Identifikations- und die Kontaktbedingung auf. Die Identifikationsbedingung besagt, dass kein Knoten aus N so getroffen werden darf,

dass er von der Regel gleichzeitig erhalten und gelöscht werden muss. Da die hier betrachteten Morphismen immer injektiv sind, kann die Identifikationsbedingung nicht verletzt werden. Daher muss nur die Einhaltung der Kontaktbedingung geprüft werden. Die Kontaktbedingung besagt, dass kein Knoten gelöscht werden darf, der mit einem nicht in m enthaltenden Knoten verbunden ist. Das bedeutet, dass alle Nachbarn der zu löschenden Knoten Teil von m sein müssen.

Beispiel

In Abbildung 2.6 wird die Anwendung einer Regel demonstriert. Dazu wird zuerst der Match m bestimmt. Zu beachten ist, dass die linke obere Stelle aus L auf die rechte obere Stelle aus N abgebildet wird und umgekehrt. Das soll verdeutlichen, dass ein Match nicht an die optische Repräsentation und die Reihenfolge der Knoten gebunden ist. Danach wird in D die Transition b eingefügt. Abschließend werden die in R fehlenden Elemente gelöscht. Das betrifft die Transition a und die linke obere Stelle aus D. Wie der Abbildung zu entnehmen ist, sind m, c und n normale Morphismen. Für die anderen vier müssen die Markierungen jedoch übereinstimmen. Außerdem enthalten K und D nur die Elemente, die in wenigstens einem der beiden benachbarten Netze enthalten sind.

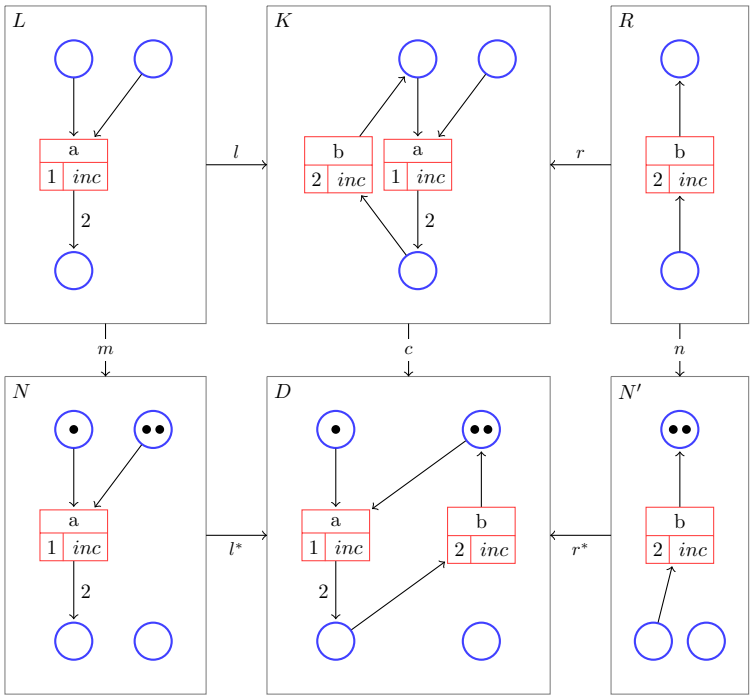


Abbildung 2.6.: Netztransformation bei dekorierten Petrinetzen

2.1.4. Negative Anwendungsbedingungen

Mit Hilfe der Regeln können die dekorierten Petrinetze erweitert und verkleinert werden. Allerdings wird eine Kontrollstruktur benötigt, die die Anwendung einer Regel in bestimmten Fällen unterbindet. Die negativen Anwendungsbedingungen (NACs) sind eine solche Kontrollstruktur (siehe Abbildung 2.7).

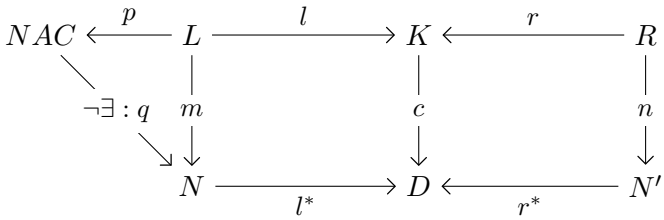


Abbildung 2.7.: Negative Anwendungsbedingung

Eine Regel kann nur angewendet werden, wenn alle ihr zugewiesenen negativen Anwendungsbedingungen erfüllt werden. Dabei wird eine NAC erfüllt, wenn kein Match q existiert, so dass $q \circ p = m$ ist [vgl. Pad12, S. 7]. Eine NAC fordert also, dass ihr Netz (ausgehend von m) nicht in N enthalten ist. In Abbildung 2.8 wird eine NAC dargestellt, die das doppelte Anlegen der Transition b verhindert.

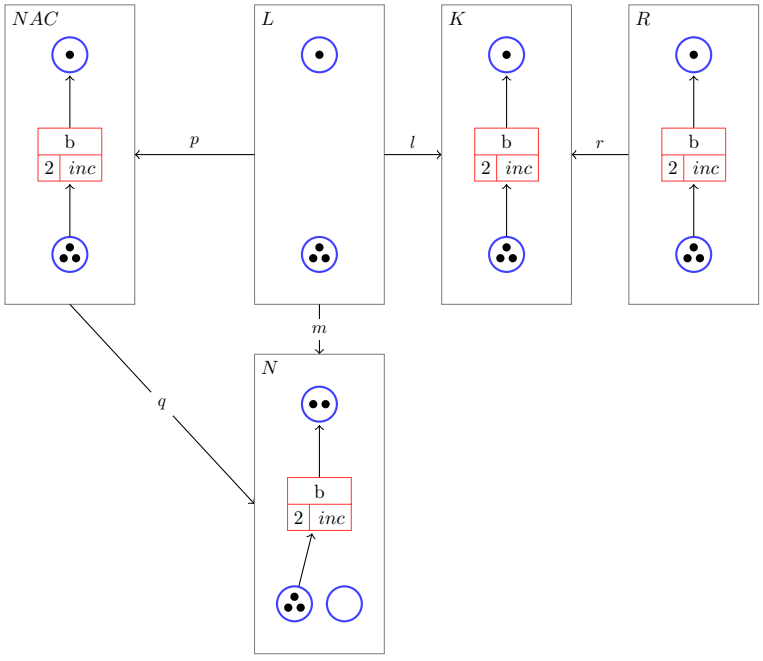


Abbildung 2.8.: Steuerung der Regelanwendung mit NACs

2.2. Eigenschaften der Matchingalgorithmen

2.2.1. Korrektheit und Vollständigkeit

Aus der Sicht des Matchings wird ein Algorithmus als korrekt angesehen, wenn jeder gefundene Netzmorphismus die Anforderungen an einen Match erfüllt (siehe Unterabschnitt 2.1.2). Das bedeutet, dass ein injektiver Morphismus den Vor- und Nachbereich der Transitionen erhält sowie die Knoten hinsichtlich ihrer Markierungen und Dekorationen korrekt abbildet. Darüber hinaus muss er die Anforderungen seines Einsatzkontextes erfüllen. Bei dem Matching zur Regelanwendung betrifft das beispielsweise die Einhaltung der Klebebedingung (siehe Abschnitt 2.1.3). Da ein nach dieser Definition korrekter Algorithmus nicht unbedingt alle existierenden Matches findet, ist seine Vollständigkeit von Interesse. Dabei heißt ein Algorithmus vollständig, wenn er alle korrekten Netzmorphismen findet.

2.2.2. Nichtdeterminismus

Bei der Betrachtung der Matchingalgorithmen können vier weitere Eigenschaften unterschieden werden. Die Ersten beiden betreffen die zu berechnenden Ergebnisse und die Letzten beiden die möglichen Abläufe.

Determiniertes und nichtdeterminiertes Ergebnis:

Ein Algorithmus liefert ein determiniertes Ergebnis, wenn er bei der gleichen Eingabe immer das gleiche Ergebnis liefert. Falls bei gleicher Eingabe verschiedene Ausgaben entstehen können, wird der Algorithmus als nichtdeterminiert bezeichnet.

Deterministischer und nichtdeterministischer Ablauf:

Von dem Ergebnis losgelöst kann der Ablauf eines Algorithmus als deterministisch oder nichtdeterministisch charakterisiert werden. Ein Algorithmus besitzt einen deterministischen Ablauf, wenn die auszuführende Schrittfolge eindeutig festgelegt ist. Falls eine Wahlmöglichkeit innerhalb seines Ablaufes besteht, wird ein Algorithmus als nichtdeterministisch bezeichnet.

Ein deterministischer Algorithmus umfasst auch ein determiniertes Ergebnis, wohingegen ein determiniertes Ergebnis durchaus mit einem nichtdeterministischen Algorithmus berechnet werden kann [vgl. SS10, S. 17 - 18]. In dieser Arbeit werden die Begriffe *Nichtdeterminismus* und *nichtdeterministisch* immer in der Bedeutung des nichtdeterministschen Ablaufes mit einem nichtdeterminierten Ergebnis verwendet. Für die

gleiche Eingabe werden also auf unterschiedlichen Wegen gegebenenfalls unterschiedliche Ergebnisse berechnet.

2.2.3. Fairness

Ein Matchingalgorithmus wird als fair bezeichnet, wenn er alle der n möglichen Matches zwischen zwei Petrinetzen mit der Wahrscheinlichkeit von $\frac{1}{n}$ trifft. Das Ziel ist eine Gleichverteilung bei der Machthäufigkeit. Davon abzugrenzen ist der Begriff der Pseudofairness. Diese besagt, dass jeder Match mit einer Wahrscheinlichkeit von $\frac{1}{n} \pm x$ getroffen wird, wobei x eine akzeptierte Abweichung ausdrückt.

3. Auswahl des Ausgangsalgorithmus

3.1. Anforderungen

Aufgrund der verschiedenen Betrachtungsweisen lassen sich die Anforderungen in vier Kategorien einteilen (siehe Abbildung 3.1). Zuerst muss ein möglicher Ausgangsalgorithmus – im Folgenden „Kandidat“ genannt – die Besonderheiten der vorgestellten Petrinetze und der Regeln erfassen. Da die Netze durch ihre Dekoration einige zusätzliche Merkmale aufweisen, wird kein Kandidat sofort alle Eigenschaften abdecken. Die Anpassbarkeit ist also ein wichtiges Kriterium. Die nächste Kategorie betrifft den späteren Einsatz. Der Zielalgorithmus wird in einer Simulationsumgebung verwendet. Dadurch werden beispielsweise die Korrektheit und die Fairness entscheidend. Abschließend muss der adaptierte Algorithmus in ReConNet umgesetzt werden. Daraus ergeben sich zusätzlich implementierungsspezifische Anforderungen.

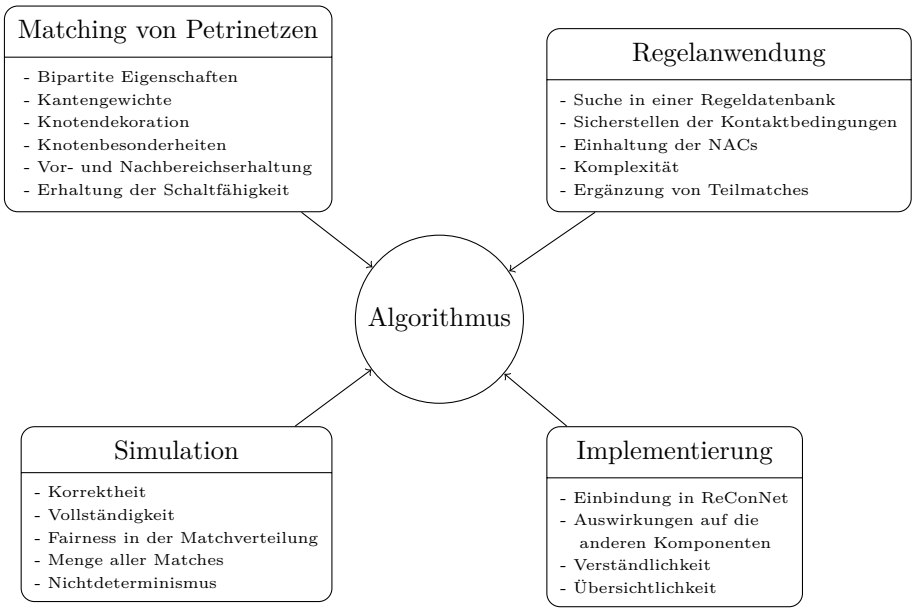


Abbildung 3.1.: Anforderungen an den Ausgangsalgorithmus

3.1.1. Matching von Petrinetzen

Die Untergraphisomorphie zwischen zwei einfachen Graphen erfordert, dass der Ausgangsgraph zu einem Untergraph des Zielgraphen isomorph ist. Bei der Betrachtung der Korrektheit der gefundenen Isomorphie sind nur die Knoten und Kanten dieses Untergraphen von Bedeutung. Im Vergleich dazu müssen für Petrinetze einige zusätzliche Bedingungen erfüllt werden, da die verschiedenen Knotenarten unterschiedliche Semantiken besitzen. Diese lassen sich am einfachsten aus der Perspektive der Transitionen betrachten. Eine grundlegende Bedingung ist, dass ihr Vor- und Nachbereich bei der Abbildung erhalten bleibt. Die Erhaltung bezieht sich sowohl auf die Anzahl der Nachbarn als auch auf die Gewichtung der Kanten. Daraus folgt, dass für Transitionen keine zusätzlichen Nachbarn im Zielnetz existieren dürfen. Sie müssen genau so gefunden werden, wie sie im Ausgangsnetz vorkommen. Für die Stellen besteht diese Einschränkung im Allgemeinen nicht. Neben den strukturellen Zwängen des Matchings müssen auch die Dekorationen (*tlb*, *tname* usw.) und Eigenheiten (z.B. Bipartitheit) beider Knotenklassen korrekt erfasst werden. Darüber hinaus muss die Schaltfähigkeit erhalten bleiben. Ein schaltfähiges Ausgangsnetz darf also nicht auf ein nichtschaltfähiges Unternetz abgebildet werden.

3.1.2. Regelanwendung

Die Suche eines Matches kann in verschiedenen Kontexten erfolgen. Aus der Sicht der Regelanwendung können wenigstens zwei Fälle unterschieden werden. Zum einen wird ein Match vom linken Teil der Regel L in das Ausgangsnetz gesucht und zum anderem dieser gegen eine Menge von negativen Anwendungsbedingungen (NACs) verifiziert. Je nach Kontext muss der Algorithmus verschiedene Kriterien erfüllen. Bei der Suche eines Matches für L sind beispielsweise die Indentifikations- und Kontaktbedingung einzuhalten. Die Indentifikationsbedingung wird durch die Injektivität des Matches automatisch erfüllt. Was die Kontaktbedingung angeht, kann diese nur durch die Abbildung der Stellen verletzt werden. Allerdings ist im vornherein bekannt, welche Stellen durch eine Regel gelöscht werden. Damit ist auch bekannt, welche Stellen für die Einhaltung der Kontaktbedingung relevant sind. Um die Erzeugung von Matches zu vermeiden, die im Nachhinein (vorhersehbar) abgelehnt werden, muss der Algorithmus bei den Stellen zwischen verschiedenen Abbildungsarten unterscheiden. Der andere angesprochene Fall, die Prüfung der NACs, ist nicht weniger wichtig. Nachdem ein potentieller Match gefunden wurde, wird geprüft, ob ein NAC bzgl. des Matches in das Ausgangsnetz abbildbar ist.

Das ungerichtete Suchen ist dabei wenig sinnvoll, da durch den Match von L schon ein Teilmatch der NAC bekannt ist. Der Algorithmus muss von einem Teilmatch ausgehend einen vollständigen Match berechnen können.

Die Anforderungen der beiden beschriebenen Kontexte ergeben sich aus dem Ziele der Effizienzsteigerung. Dabei ist zu beachten, dass bei der Simulation eine anwendbare Regel aus einer Regeldatenbank gesucht wird. Im Vornherein ist meist nicht bekannt, ob ein Match existiert oder nicht. Da schon die Ausführung einer Regel aufwändig ist, muss neben der Matchingeffizienz auch der Aufwand beachtet werden, der betrieben werden muss, wenn kein Match möglich ist [vgl. IB03].

3.1.3. Simulation

Die Simulation von Szenarien betrifft sowohl das Schalten der Netze als auch deren Veränderung durch Regeln. Wenn eine Regel auf das Netz angewendet wird, kann der Fall eintreten, dass L im Ausgangsnetz eine Vielzahl von möglichen Matches besitzt. Für die Simulation ist entscheidend, dass die Ausführung derselben Regel nicht immer den selben Netzteil verändert. Falls also n Matches möglich sind, soll der Algorithmus jeden Match mit einer Wahrscheinlichkeit von $\frac{1}{n}$ treffen. Es gibt verschiedene Möglichkeiten, diese Fairness zu erreichen. Die erste ist die Menge aller Matches zu generieren und zufällig ein Element zu wählen. Allerdings ist das Generieren dieser Menge bei großen Netzen recht aufwändig. Der zweite Ansatz versucht dieses Effizienzproblem zu lösen, indem die Anforderung an die Fairness etwas gelockert wird. Dazu soll die Wahl des Matches durch einen nichtdeterministischen Ablauf des Algorithmus simuliert werden. Das Ziel ist, dass jeder Match mit etwa der gleichen Wahrscheinlichkeit ausgegeben werden kann. Allerdings stellt dies unter Umständen nur eine Pseudofairness dar. Also muss die Generierung der Menge aller Matches ebenfalls möglich sein.

3.1.4. Implementierung

Die letzten Kriterien ergeben sich aus der praktischen Umsetzbarkeit des Algorithmus in ReConNet. Da auf einem bestehenden Werkzeug aufgebaut wird, muss sich der Algorithmus in dessen Architektur einbetten (siehe Abschnitt 5.1). Er darf also nicht so grundlegend in seiner Philosophie abweichen, dass er nur mit unvertretbarem Aufwand umsetzbar ist. Dies betrifft insbesondere grundlegende Eingriffe in die Architektur oder die Kommunikation innerhalb des Werkzeugs. Allerdings sind die Implementierungsschwierigkeiten durch die Komplexität des Matchingproblems eher nachrangig.

3.2. Matchingalgorithmen

In den vergangenen Jahrzehnten wurden eine Vielzahl verschiedener Matchingalgorithmen entwickelt. Allerdings befindet sich unter ihnen kein Algorithmus zum Matching von Petrinetzen. Daher bezieht sich die Menge der möglichen Kandidaten auf allgemeine Matchingalgorithmen. Eine ausführliche Diskussion verschiedener Vertreter wurde von Conte et al. mit [Con+04] veröffentlicht, worin sie sowohl exakte als auch inexakte Algorithmen untersuchen. Da das Thema dieser Arbeit nicht die Ausarbeitung der Vor- und Nachteile der einzelnen Algorithmen ist, werden zuerst beispielhaft einige Vertreter vorgestellt und danach ein geeigneter Ausgangsalgorithmus gewählt. Conte teilt die exakten Algorithmen in drei Kategorien ein. Die erste bildet die Gruppe der Baumsuchalgorithmen. Die meisten vorgestellten Vertreter fallen in diese Kategorie. Algorithmen, die keine einfache Baumsuche umfassen, fallen in die zweite Kategorie. Die letzte Gruppe bilden die effizienten Algorithmen für spezielle Graphen (z.B. Bäume, planare Graphen und Graphen mit begrenztem Grad). Allerdings erkennen die Algorithmen der dritten Kategorie nur die Graphisomorphie [vgl. Con+04, S. 267 ff.].

Einen Vertreter der zweiten Kategorie bildet der Algorithmus von Messmer und Bunke [MB00]. Dieser wurde in den folgenden Jahren von verschiedenen Autoren aufgegriffen und für ihre Einsatzzwecke verbessert. Die Grundidee ist dabei immer die gleiche. Auf der einen Seite steht eine Datenbank von Modellgraphen und auf der anderen Seite ein Eingabegraph. Nun soll eine Untergraphisomorphie zwischen einem Modellgraphen und dem Eingabegraphen gefunden werden. Der naive Ansatz zur Lösung dieses Problems ist, dass die einzelnen Modellgraphen sequentiell durchprobiert werden. Allerdings besitzt diese Strategie den Nachteil, dass die Laufzeit von der Anzahl der beteiligten Graphen abhängt [vgl. MB00, S. 309]. Messmer und Bunke wollen mit ihrem Algorithmus diese Abhängigkeit so weit wie möglich vermeiden. Sie verfolgen den Ansatz, mehrere Modellgraphen gleichzeitig auf den Eingabegraphen abzubilden. Damit dieses gemeinsame Matching stattfinden kann, muss zuerst ein Vorverarbeitungsschritt durchgeführt werden. Im Zuge dessen werden die Modellgraphen in kleinere Untergraphen zerlegt. Das Ziel der Zerlegung ist es, Graphen zu erzeugen, die in möglichst vielen Modellgraphen vorkommen. Die entstehenden Untergraphen werden wiederum in ihre Bestandteile zerlegt. Dieser Vorgang hält so lange an, bis die Graphen nur noch aus einem Knoten bestehen. Anschließend umfasst die Graphdatenbank sowohl die Modellgraphen als auch deren gemeinsame Unterstrukturen. Beim Matching werden diese Unterstrukturen geordnet in den Zielgraphen abgebildet. Als Erstes wird versucht, die Graphen aus der Datenbank zu

matchen, die nur aus einem Knoten bestehen. Aus diesen einfachen Knotenzuordnungen werden anschließend die größeren gemeinsamen Untergraphen hergeleitet. Das geschieht so lange, bis ein Modellgraph erreicht wird. Durch das Matching der gemeinsamen Unterstrukturen werden gleichzeitig verschiedene Modellgraphen abgebildet. Aber dieser Ansatz ist nicht unproblematisch. Der Erfolg der Strategie hängt maßgeblich von der Güte der Vorverarbeitung ab. Die Berechnung der besten Zerlegung ist jedoch mit exponentiellem Zeitaufwand verbunden [vgl. MB00, S. 310]. Außerdem ist die Größe der entstehenden Datenbank nicht zu unterschätzen. Die von anderen Autoren vorgenommenen Verbesserungen betreffen hauptsächlich die Einführung von verschiedenen Zerlegungsstrategien und die Reduktion des Speicherverbrauchs. Bei der Bewertung des Algorithmus für das Matching von rekonfigurierbaren Petrinetzen fällt die Ähnlichkeit zwischen der beschriebenen Graphdatenbank und der Regeldatenbank auf. Obwohl der Ansatz eine gute Effizienz verspricht, können verschiedene Schwierigkeiten nicht übersehen werden. Eine der Kernanforderungen der Simulation ist die Fairness. Zwar können unterschiedliche Matches bei jedem Durchlauf auftreten, allerdings beeinflusst die zuvor vorgenommene Zerlegung deren Verteilung. Außerdem besitzen die dekorierten Petrinetze eine Vielzahl von Eigenschaften. Dadurch sind gemeinsame Unterstrukturen zwischen verschiedenen Netzen unwahrscheinlicher als bei allgemeinen Graphen.

Einer der bekanntesten Baumsuchalgorithmen ist der Matchingalgorithmus von Ullmann [Ull76]. Eine angepasste Form wird bisher in ReConNet (siehe Unterabschnitt 1.1.2) eingesetzt. Allerdings besitzt diese Version einige Nachteile. Bei der Suche verwendet der Ullmann eine Vielzahl von Matrizen und besitzt dadurch einen nicht zu vernachlässigenden Speicherbedarf. Da in den letzten Jahren viele Algorithmen entwickelt wurden, die sowohl im Zeit- als auch Speicherbedarf besser sind, scheidet er für eine weitere Nutzung aus.

Beim Matching von Graphen werden normalerweise Knoten abgebildet und die Struktur durch die Prüfung der Kanten sichergestellt. Der Graph Explorer [DZ09] verfolgt einen anderen Ansatz, indem er nach und nach die Kanten aufeinander abgebildet. Dazu wandelt der Algorithmus den zu findenden Graphen zuerst in einen Baum um. Die Knoten und Kanten des Baumes stehen für die Knoten und Kanten des Graphen. Allerdings kann der Ausgangsgraph Kanten besitzen, die die Struktur des Baumes verletzen würden. Deswegen wird der Graph an diesen Stellen aufgespalten und die entsprechenden Kanten gesondert betrachtet. Mit Hilfe des Baumes wird anschließend die Reihenfolge der abzubildenden Kanten bestimmt. Auf diese Weise wandelt der Graph Explorer das Matching- in ein Wegesuchproblem um. Zur Beschneidung des Suchraumes

wird außerdem eine intelligente Backtrackingstrategie eingeführt [vgl. DZ09, S. 2]. Ein Vorteil dieses Vorgehens ist, dass die Labels je Knoten kaum öfter als einmal geprüft werden müssen [vgl. DZ09, S. 5]. Für das Matching von Petrinetzen erscheint dieser Ansatz jedoch problematisch. Bei der Anwendung der Regeln müssen beispielsweise die NACs geprüft werden. Damit die Prüfung einer NAC auf einem Teilmatch aufbauen kann, müsste diese entsprechend dem Teilmatch sinnvoll zerlegt werden. Außerdem gestaltet sich die Generierung der Menge aller Matches schwierig. Da die Fairness aber eines der wichtigsten Kriterien ist und nicht jede Simulation auf eine Pseudofairness vertrauen kann, ist der Ansatz des Graph Explorers ungeeignet.

3.3. VF2

Wie der Ullmann und der Graph Explorer zeigen, existieren im Bereich der Baumsuchalgorithmen viele verschiedene Ansätze. Einen der bedeutendsten Matchingalgorithmen der jüngeren Vergangenheit bildet der VF-Algorithmus von Cordella et al. [Cor+99], welcher später zum VF2 [Cor+04] verbessert wurde. Für die Entwicklung eines Matchingalgorithmus für dekorierte Petrinetze (siehe Kapitel 4) wird der VF2 als Ausgangsalgorithmus verwendet. Zur besseren Nachvollziehbarkeit der vorgenommen Anpassungen wird er im Folgenden ausführlich erläutert[1].

3.3.1. Funktionsweise

Der VF2 sucht eine Untergraphisomorphie zwischen dem Quellgraphen G_1 und dem Zielgraphen G_2. Dabei sind $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ gerichtete Graphen, die weder Schleifen noch Mehrfachkanten enthalten. Die gefundene Untergraphisomorphie wird durch den Match M beschrieben und hat je nach Kontext die Form einer Abbildung $M : V_1 \longrightarrow V_2$ oder Relation $M \subseteq V_1 \times V_2$. Wenn der Match als Relation aufgefasst wird, besitzt er die der Abbildung entsprechenden mengentheoretischen Eigenschaften. Außerdem repräsentieren M_1 und M_2 die jeweiligen Mengen der abgebildeten Knoten von G_1 und G_2. Somit gilt :

$$M(v) = w \;\equiv\; (v, w) \in M \qquad\qquad M_1 = \{v \mid (v, w) \in M\}$$
$$M_2 = \{w \mid (v, w) \in M\}$$

[1] Bei der folgenden Beschreibung wird eine zu den Originalquellen leicht abweichende Definition verwendet. Bevor also eine Quellenrecherche betrieben wird, sei auf die Hinweise in den Anlagen dieser Arbeit verwiesen.

Zur Berechnung von M führt der VF2 eine rekursive Tiefensuche durch. Dabei stellen die möglichen Knotenabbildungen von G_1 nach G_2 den zu durchsuchenden Suchraum dar. Auf jeder Rekursionsebene versucht er, einen Quellknoten aus V_1 auf verschiedene Zielknoten aus V_2 abzubilden. Nach der erfolgreichen Abbildung eines Paares vollzieht der Algorithmus einen Rekursionsschritt und sucht auf der nächsten Ebene ein weiteres Paar. Auf diese Weise werden immer größere Teilmatches berechnet. Allerdings führt nicht jede Knotenkombination zu einer strukturerhaltenden Abbildung. Damit die Korrektheit des am Ende gefundenen Matches gewährleistet ist, verfolgt der VF2 die Philosophie, dass auf dem Weg zum Match nur korrekte Teilmatches erlaubt sind. Das bedeutet, dass nur die Teilmatches erzeugt werden, bei denen die induzierten Untergraphen isomorph sind. Nachdem alle Knoten aus V_1 bearbeitet worden sind, muss der aktuelle Teilmatch ein gültiger Match von G_1 nach G_2 sein. Dieser Vorgang wird im VF2 mit der rekursiv definierten Methode $Match(s)$ umgesetzt.

```
1  PROCEDURE Match(s)
2     INPUT:   an intermediate state s; the initial state s_0 has M(s_0) = ∅
3     OUTPUT:  the mappings between the two graphs
4
5     IF M(s) covers all the nodes of G_1 THEN
6        OUTPUT M(s)
7     ELSE
8        Compute the set P(s) of the pairs candidate for inclusion in M(s)
9        FOREACH p in P(s)
10          IF the feasibility rules succeed for the inclusion of p in M(s) THEN
11             Compute the state s' obtained by adding p to M(s)
12             CALL Match(s')
13          END IF
14       END FOREACH
15       Restore data structures
16    END IF
17 END PROCEDURE Match
```

Quelle: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs [Cor+04, S. 1368][2]

Abbildung 3.2.: VF2-Algorithmus

Der aktuelle Zustand des Matchingvorganges wird als s_n und der entsprechende Teilmatch als $M(s_n)$ bezeichnet (Zeile 2). Bei der Berechnung von $Match$ sind verschiedene Schritte zu vollziehen. Als Erstes muss die Menge der möglichen Kandidaten generiert werden

[2] In [Cor+04] wird ein Match von G_2 nach G_1 gesucht. Da hier die Betrachtung in umgekehrter Reihenfolge erfolgt, wurde der Pseudocode so angepasst, dass als Quellgraph G_1 verwendet wird.

(Zeile 8). Anschließend werden alle Paare auf ihre Gültigkeit geprüft (Zeilen 9 und 10). Falls der entstehende Teilmatch korrekt ist, wird der Zustand aktualisiert und ein Rekursionsschritt vollzogen (Zeilen 11 und 12). Danach werden die restlichen Knoten auf den folgenden Ebenen untersucht. Sofern alle Knoten von G_1 abgearbeitet worden sind, bricht der Algorithmus mit einem gültigen Match ab (Zeile 6). Ansonsten kehrt er zur vorherigen Rekursionsebene zurück, stellt den letzten Zustand wieder her und untersucht das nächste Kandidatenpaar (Zeile 15).

Berechnung der Kandidatenpaare

Zwar stellen die möglichen Knotenabbildungen von G_1 nach G_2 den Suchraum dar, allerdings sind nicht alle Äste gleich vielversprechend. Die Wahl einer guten Abbildungsreihenfolge trägt maßgeblich zur Effizienz des Algorithmus bei. Da die Berechnung der perfekten Abbildungsreihenfolge der Lösung des Matchingproblems gleichkommt, kann diese nur abgeschätzt werden. Jeder Algorithmus verfolgt eine etwas andere Strategie zur Berechnung der nächsten Kandidaten. Im Falle des VF2 werden die dem Teilmatch angrenzenden Knoten zuerst betrachtet. Falls ein Quellnetz aus verschiedenen Komponenten besteht, versucht der VF2 sie nacheinander als Ganzes in das Zielnetz abzubilden. Dieses Vorgehen besitzt verschiedene Vorteile. Beispielsweise kann der VF2 frühzeitig erkennen, ob das Weiterverfolgen des aktuellen Zustands zu einem gültigen Match führen kann. Da der Algorithmus die benachbarten Knoten bei der Kandidatengenerierung bevorzugt, vermeidet er außerdem das Fehlschlagen des Matchingvorganges in einer hohen Rekursionstiefe und daraus resultierendes Backtracking. Damit die Strategie angewendet werden kann, müssen zuerst Ordnungen über die Knoten beider Graphen definiert werden. Die Reihenfolge der Knoten ist unwichtig, solange sie während des Durchlaufes stabil bleibt. Die Ordnungen werden benötigt, damit der Algorithmus dieselben Zustände nicht über verschiedene Ausführungspfade erzeugt. Anschließend können die Kandidaten berechnet werden. Dazu werden die Terminal- und Restknotenmengen des aktuellen Zustandes betrachtet.

$$T^{out}(s_n) = T_1^{out}(s_n) \cup T_2^{out}(s_n) \qquad T_1(s_n) = T_1^{out}(s_n) \cup T_1^{in}(s_n)$$
$$T^{in}(s_n) = T_1^{in}(s_n) \cup T_2^{in}(s_n) \qquad T_2(s_n) = T_2^{out}(s_n) \cup T_2^{in}(s_n)$$

$$\widetilde{V_1}(s_n) = V_1 - M_1(s_n) - T_1(s_n) \qquad \widetilde{V_2}(s_n) = V_2 - M_2(s_n) - T_2(s_n)$$

Die Terminalmengen enthalten die Knoten, die direkt mit dem Teilmatch verbunden sind. Die Nachbarn des Teilmatches werden dazu in zwei Gruppen eingeteilt. Die erste Gruppe bildet die Menge der Knoten, die von einer Kante erreicht werden, die im Teilmatch startet. Diese Gruppe wird als out-Menge T^{out} bezeichnet. Im Gegensatz dazu enthält die in-Menge T^{in} die Knoten, bei denen eine Karte startet, die im Teilmatch endet. Falls ein Knoten sowohl eine ausgehende als auch eine eingehende Kante besitzt, ist er in beiden Terminalmengen enthalten. Die Restknotenmenge $\widetilde{V}$ beinhaltet die übrigen Knoten, die weder im Match noch in den Terminalmengen enthalten sind. Die Indizes der einzelnen Mengen entsprechen der Nummer des Graphen, aus dem die Knoten stammen. Diese Zusammenhänge sollen mit der Abbildung 3.3 verdeutlicht werden.

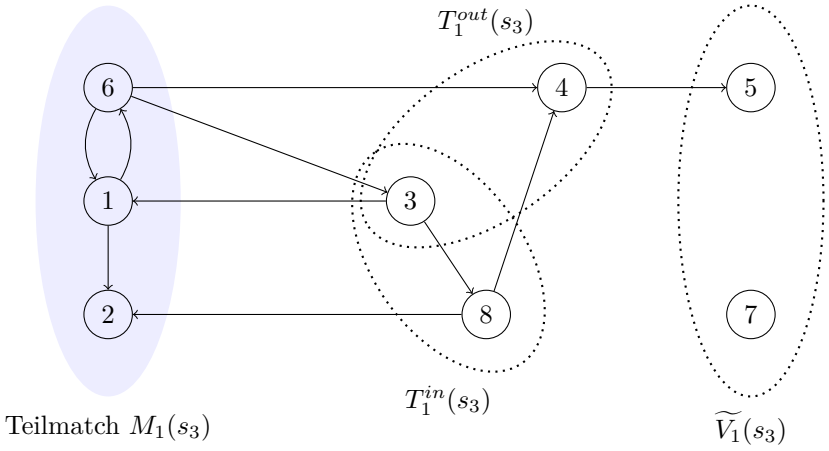


Abbildung 3.3.: Knotenmengen

Die Information über die Mengenzugehörigkeit eines Knoten verwendet der VF2 zur Beschneidung des Suchraumes. Beispielsweise ist es wenig sinnvoll, einen Knoten aus T_1^{out} auf einen Knoten aus $\widetilde{V_2}$ abzubilden. Bei der Berechnung der endgültigen Kandidatenmenge $P(s_n)$ spiegelt sich diese Strategie in den unterschiedlichen speziellen Kandidatenmengen wider.

$$
\begin{aligned}
P_{T^{out}}(s_n) &= \{\min T_1^{out}(s_n)\} \times T_2^{out}(s_n) \\
P_{T^{in}}(s_n) &= \{\min T_1^{in}(s_n)\} \times T_2^{in}(s_n) \\
P_{\widetilde{V}}(s_n) &= \{\min(V_1 - M_1(s_n))\} \times (V_2 - M_2(s_n))
\end{aligned}
$$

Allen gemeinsam ist, dass die jeweilige Menge eine Kombination aus dem kleinsten Quellknoten und allen freien Zielknoten derselben Art darstellt. Die Zugehörigkeit eines

Knoten zu einer bestimmten Knotenmenge ist jedoch nicht der einzige Grund, aus dem zwischen $P_{T^{out}}$, $P_{T^{in}}$ und $P_{\widetilde{V}}$ unterschieden wird. Da der VF2 korrekte Teilmatches berechnet und die umliegenden Knoten beim Matching bevorzugt, muss die Nachbarschaft von $M_1(s_n)$ nach G_2 abbildbar sein. Daher untersucht der Algorithmus auf jeder Rekursionsebene genau eine dieser drei Kandidatenmengen. Falls beispielsweise alle Paare aus $P_{T^{out}}$ fehlschlagen, kann es in diesem Rekursionsast selbst in einer höheren Tiefe keinen gültigen Match gegeben. Die Wahl der jeweiligen Menge ist durch die Strategie des VF2 fest vorgegeben.

$$P(s_n) = \begin{cases} P_{T^{out}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| > 0 \\ P_{T^{in}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| = 0 \wedge |P_{T^{in}}(s_n)| > 0 \\ P_{\widetilde{V}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| = 0 \wedge |P_{T^{in}}(s_n)| = 0 \end{cases}$$

Der Algorithmus versucht zuerst die Knotenpaare aus den out-Mengen zu untersuchen. Falls eine der beiden out-Mengen (T_1^{out} oder T_2^{out}) leer ist und somit keine Kandidaten zur Verfügung stehen, verwendet er die Kandidaten der in-Mengen. Die Paare aus den Restknotenmengen werden erst untersucht, wenn sowohl $P_{T^{out}}$ als auch $P_{T^{in}}$ leer sind. Da die Knoten aus $\widetilde{V}$ nicht mit dem bisherigen Teilmatch verbunden sind, springt der Algorithmus durch $P_{\widetilde{V}}$ in einen anderen Bereich des Graphen.

Prüfung eines Kandidatenpaares auf $feasible$

Nachdem die zu untersuchenden Paare vorliegen, müssen sie auf ihre Gültigkeit geprüft werden. Für diesen Test führt der VF2 das Prädikat $feasible$ ein. Ein Paar ist zu einem Zustand s_n $feasible$, wenn die resultierenden induzierten Untergraphen isomorph sind.

$$\begin{aligned} feasible(s_n, v, w) \equiv\ & rule_{pred}(s_n, v, w) \wedge rule_{succ}(s_n, v, w) \\ & \wedge rule_{in}(s_n, v, w) \wedge rule_{out}(s_n, v, w) \wedge rule_{new}(s_n, v, w) \end{aligned}$$

Da bei der Prüfung eines Paares verschiedene Aspekte eine Rolle spielen, teilt sich $feasible$ in fünf Regeln auf. Die ersten beiden Regeln prüfen die Vorgänger und Nachfolger des Kandidatenpaares, während die letzten drei der Suchraumbeschneidung dienen.

$$\begin{aligned} rule_{pred}(s_n, v, w) \equiv & \Big[\forall v' \in M_1(s_n) \wedge (v', v) \in E_1 : \exists (w', w) \in E_2 : (v', w') \in M(s_n)\Big] \\ & \wedge \Big[\forall w' \in M_2(s_n) \wedge (w', w) \in E_2 : \exists (v', v) \in E_1 : (v', w') \in M(s_n)\Big] \end{aligned} \tag{3.1}$$

$$rule_{succ}(s_n, v, w) \equiv \Big[\forall v' \in M_1(s_n) \wedge (v, v') \in E_1 : \exists (w, w') \in E_2 : (v', w') \in M(s_n)\Big] \\ \wedge \Big[\forall w' \in M_2(s_n) \wedge (w, w') \in E_2 : \exists (v, v') \in E_1 : (v', w') \in M(s_n)\Big] \tag{3.2}$$

Beim rekursiven Abstieg verlässt sich der VF2 darauf, dass der Teilmatch einer jeden Rekursionsebene für sich gesehen korrekt ist. Diese Eigenschaft kann allerdings verloren gehen, wenn ein Paar zum aktuellen Zustand hinzugefügt wird. Für den neuen Zustand ist bekannt, dass die Korrektheit nur durch Kanten zwischen den Kandidatenknoten und dem bisherigen Teilmatch verletzt werden kann. Damit die Strukturerhaltung des entstehenden Teilmatches sichergestellt wird, umfasst $feasible$ die beiden Regeln $rule_{pred}$ und $rule_{succ}$. Mit der Hilfe von $rule_{pred}$ prüft der VF2 die korrekte Abbildung der Vorgänger der Kandidatenknoten. Dazu wird getestet, ob für alle Kanten, die von einem abgebildeten Knoten v' nach v führen, eine entsprechende Kante zwischen den jeweiligen Knoten w' und w besteht. Damit im Zielnetz keine Kanten existieren, die im Quellnetz fehlen, muss die Prüfung noch einmal in Gegenrichtung erfolgen. Nachdem die Korrektheit der Vorgänger beider Knoten sichergestellt ist, prüft der Algorithmus mit $rule_{succ}$ die entsprechende Bedingung aus der Sicht der Nachfolger von v und w. Jedes Paar, dass die ersten beiden Regeln besteht, führt bei der Hinzunahme zum aktuellen Zustand zu einem korrekten Teilmatch. Allerdings lohnt sich nicht die Verfolgung aller korrekten Zustände, da mit den letzten drei Regeln frühzeitig die Abbildbarkeit der Nachbarschaft der Kandidatenknoten geprüft werden kann.

$$rule_{in}(s_n, v, w) \equiv |\{(v', v) \in E_1 \mid v' \in T_1^{in}\}| \leq |\{(w', w) \in E_2 \mid w' \in T_2^{in}\}| \\ \wedge |\{(v, v') \in E_1 \mid v' \in T_1^{in}\}| \leq |\{(w, w') \in E_2 \mid w' \in T_2^{in}\}| \tag{3.3}$$

$$rule_{out}(s_n, v, w) \equiv |\{(v', v) \in E_1 \mid v' \in T_1^{out}\}| \leq |\{(w', w) \in E_2 \mid w' \in T_2^{out}\}| \\ \wedge |\{(v, v') \in E_1 \mid v' \in T_1^{out}\}| \leq |\{(w, w') \in E_2 \mid w' \in T_2^{out}\}| \tag{3.4}$$

$$rule_{new}(s_n, v, w) \equiv |\{(v', v) \in E_1 \mid v' \in \widetilde{V_1}\}| \leq |\{(w', w) \in E_2 \mid w' \in \widetilde{V_2}\}| \\ \wedge |\{(v, v') \in E_1 \mid v' \in \widetilde{V_1}\}| \leq |\{(w, w') \in E_2 \mid w' \in \widetilde{V_2}\}| \tag{3.5}$$

Der Test der Nachbarschaft erfolgt in der Restknotenmenge und den beiden Terminalmengen. Damit ein gültiges Paar weiter verfolgt wird, muss der Zielknoten in jeder

Menge wenigstens so viele nicht abgebildete Nachbarn besitzen, wie der Quellknoten im Quellnetz. Durch die Entfernung der untersuchten Nachbarn zum aktuellen Teilmatch stellen $rule_{out}$ und $rule_{in}$ einen 1-Lookahead und $rule_{new}$ einen 2-Lookahead dar. Falls keine Untergraphisomorphie, sondern eine Graphisomorphie gesucht wird, müssen die verschiedenen $\leq$-Beziehungen durch die Gleichheit ersetzt werden.

Aktualisierung des Zustandes

Eine der bedeutendsten Verbesserungen des VF2 stellt die Art der verwendeten Datenstruktur dar. Damit der Algorithmus auch für den Einsatz bei Graphen mit mehreren tausend Knoten geeignet ist, verwendet er als Zustandsbeschreibung eine State-Space-Representation (SSR). Die SSR besteht für beide Graphen aus jeweils drei Arrays, die die Länge der Knotenanzahl der Graphen haben, und wird über alle Rekursionsebenen hinweg verwendet.

core Die jeweiligen core-Arrays von G_1 und G_2 enthalten den bisherigen Teilmatch, wobei jeder Knoten ein eigenes Feld besitzt. Die Position dieses Feldes wird durch die Position des Knoten in seiner Ordnung bestimmt. Zu Beginn sind alle Felder mit $NULL\_NODE$ vorbelegt. Dieser Wert zeigt an, dass der jeweilige Knoten noch nicht abgebildet wurde. Während des Matchings werden die verschiedenen Felder der core-Arrays mit Werten belegt. Der Eintrag entspricht dem Index des Knotens im anderen Graphen. Falls beispielsweise der zweite Knoten von G_1 auf den vierten Knoten von G_2 abgebildet wird, enthält $core\_1[2]$ die 4 und $core\_2[4]$ die 2.

out und in Die out- und in-Arrays dienen der Verwaltung der Terminalmengen. Anfangs sind alle Felder mit 0 belegt. Bei dem Hinzufügen eines Knotens zum aktuellen Zustand werden die Felder der entsprechenden Nachbarknoten gesetzt. Der Wert deckt sich mit der Rekursionstiefe, in der ein Knoten in die Terminalmenge eintritt. Falls ein Paar zum Zustand hinzugefügt wird, dessen Felder noch den Wert 0 haben, werden diese ebenfalls auf die aktuelle Tiefe gesetzt. Aus der Kombination der core- und Terminarrays können die Terminalmengen berechnet werden. Ein Knoten ist genau dann in T^{out} bzw. T^{in}, wenn das entsprechende Feld im out-Array bzw. in-Array gesetzt ist und der Knoten noch nicht abgebildet wurde. Also wenn der Eintrag im core-Array $NULL\_NODE$ lautet.

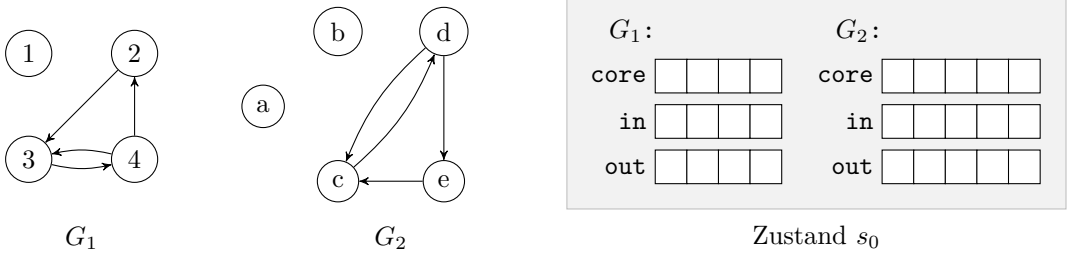
Nachdem ein Eintrag in der SSR gesetzt wird, darf dieser in den folgenden **tieferen** Rekursionsschritten nicht mehr verändert werden. Die Stabilität der SSR in Kombination

mit den sorgfältig gesetzten Werten (die Rekursionstiefe) bilden die Grundlage dafür, dass über alle Rekursionsebenen hinweg eine gemeinsame SSR verwendet werden kann. Dadurch wird der Speicherverbrauch deutlich reduziert. Falls das rekursive Matching fehlschlägt, muss der vorherige Zustand wiederhergestellt werden. Dazu entfernt der VF2 alle Einträge aus der SSR, die in der **aktuellen** Rekursionstiefe vorgenommen wurden. Alle anderen Einträge bleiben unangetastet.

3.3.2. Beispiel

Start des Matchings von G_1 nach G_2

$$M(s_0) = \emptyset \qquad P(s_0) = \{\min(V_1 - M_1(s_0))\} \times (V_2 - M_2(s_0))$$
$$= \{(1,a),(1,b),(1,c),(1,d),(1,e)\}$$
$$T_1^{in}(s_0) = \emptyset \qquad T_2^{in}(s_0) = \emptyset$$
$$T_1^{out}(s_0) = \emptyset \qquad T_2^{out}(s_0) = \emptyset$$
$$\widetilde{V_1}(s_0) = \{1,2,3,4\} \qquad \widetilde{V_2}(s_0) = \{a,b,c,d,e\}$$



G_1 G_2 Zustand s_0

$$feasible(s_0,1,a) \equiv rule_{pred}(s_0,1,a) \wedge rule_{succ}(s_0,1,a)$$
$$\wedge rule_{in}(s_0,1,a) \wedge rule_{out}(s_0,1,a) \wedge rule_{new}(s_0,1,a)$$
$$\equiv true$$

Abbildung 3.4.: VF2-Beispiel: Start des Matchings von G_1 nach G_2

In diesem Beispiel soll ein Vorkommen des Graphen G_1 in G_2 gefunden werden. Beim Start des Algorithmus werden die Knoten beider Graphen beliebigen Ordnungen unterworfen. Diese werden in der Abbildung durch die Ziffern 1 bis 4 und Buchstaben a bis e veranschaulicht. Die SSR (State-Space-Representation, siehe graue Box) wird im

Folgenden als Zustand bezeichnet. Die einzelnen Felder der core-, in- und out-Arrays stehen für die Knoten der Graphen, entsprechend ihrer Reihenfolge in der jeweiligen Ordnung. Da noch kein Knoten gematcht wurde, sind alle Felder der core-Arrays mit dem $NULL\_NODE$ und die Terminarrays mit „0“ vorbelegt. Um das Beispiel übersichtlicher zu gestalten, werden diese als leeres Kästchen dargestellt. Zur Aktualisierung des Zustandes ist die aktuelle Rekursionstiefe notwendig. Die Tiefe entspricht der Nummer des wievielten Paares, welches gematcht werden soll. Auf dieser Rekursionsebene wird das erste $feasible$ Paar gesucht, also beträgt die Tiefe eins. Die über den Graphen aufgeführten Mengen ergeben sich aus dem Zustand.

Zu Beginn dieses Matchingschrittes wird die Menge der Kandidatenpaare $P(s_0)$ generiert. Die Strategie des VF2 sieht dabei vor, dass zuerst die T^{out}- und danach ggf. die T^{in}-Mengen untersucht werden. Da auf dieser Ebene sämtliche Terminalmengen leer sind, werden alle ungematchten Knoten $V_{i\in\{1,2\}} - M_i(s_0)$ betrachtet. Die Kandidatenmenge ergibt sich aus dem Kreuzprodukt der einelementigen Menge des kleinsten freien Quellknotens $\{\min(V_1 - M_1(s_0))\} = \{1\}$ und der Menge aller ungematchten Zielknoten $V_2 - M_2(s_0) = \{a, b, c, d, e\}$. Sobald die Kandidaten berechnet sind, wird geprüft, ob die jeweiligen Paare (in Bezug auf den bisherigen Match) $feasible$ sind. Um das Prädikat zu erfüllen, müssen die folgenden Prüfungen bestanden werden:

$rule_{pred}(s_n, v, w)$ **und** $rule_{succ}(s_n, v, w)$**:** (siehe Gleichung 3.1 und Gleichung 3.2)
Sämtliche gematchten Vorgänger ($pred$) bzw. Nachfolger ($succ$) von v müssen auf einen ebenfalls gematchten Vorgänger bzw. Nachfolger von w abgebildet worden sein und umgekehrt.

$rule_{in}(s_n, v, w)$ **und** $rule_{out}(s_n, v, w)$**:** (siehe Gleichung 3.3 und Gleichung 3.4)
Die Anzahl der Vorgänger und Nachfolger von v in T_1^{in} und T_1^{out} ist kleiner oder gleich der Anzahl der Vorgänger und Nachfolger von w in T_2^{in} und T_2^{out}.

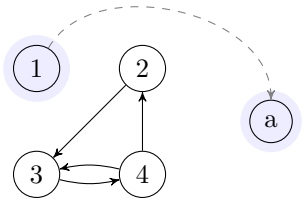
$rule_{new}(s_n, v, w)$**:** (siehe Gleichung 3.5)
Die Anzahl der Vorgänger und Nachfolger von v in $\widetilde{V_1}(s_n)$ ist kleiner oder gleich der Anzahl der Vorgänger und Nachfolger von w in $\widetilde{V_2}(s_n)$. $\widetilde{V_i}(s_n)$ sind die Knoten, welche weder im Match noch in den Terminalmengen enthalten sind.

Die ersten beiden Regeln gewährleisten einen gültigen Match, während die letzteren drei der Beschneidung des Suchraumes dienen. Die einzelnen Kandidatenpaare werden der Ordnung nach auf die Erfüllung von $feasible$ geprüft. Die Vorgänger- und Nachfolgerbedingungen können auf dieser Ebene gar nicht verletzt werden, da der bisherige

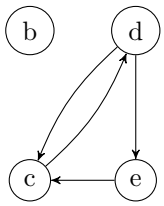
Match leer ist. Aufgrund der fehlenden Nachbarn erfüllt das erste Paar $(1, a)$ direkt die gestellten Anforderungen und wird dem aktuellen Zustand hinzugefügt. Danach geht der Algorithmus in die nächste Rekursionsebene über.

Nach dem Match von $(1, a)$

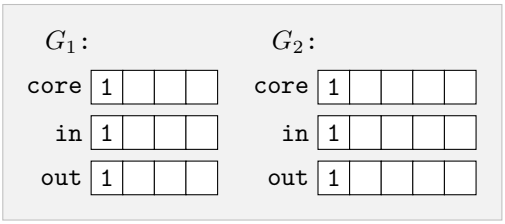
$$
\begin{aligned}
M(s_1) &= \{(1,a)\} & P(s_1) &= \{\min(V_1 - M_1(s_1))\} \times (V_2 - M_2(s_1)) \\
& & &= \{(2,b),(2,c),(2,d),(2,e)\} \\
T_1^{in}(s_1) &= \emptyset & T_2^{in}(s_1) &= \emptyset \\
T_1^{out}(s_1) &= \emptyset & T_2^{out}(s_1) &= \emptyset \\
\widetilde{V_1}(s_1) &= \{2,3,4\} & \widetilde{V_2}(s_1) &= \{b,c,d,e\}
\end{aligned}
$$



G_1



G_2



Zustand s_1

$$
\begin{aligned}
feasible(s_1,2,b) &\equiv rule_{pred}(s_1,2,b) \wedge rule_{succ}(s_1,2,b) \\
&\quad \wedge rule_{in}(s_1,2,b) \wedge rule_{out}(s_1,2,b) \wedge rule_{new}(s_1,2,b) \\
&\equiv true \wedge true \wedge true \wedge true \wedge true \wedge false \\
&\equiv false \\
feasible(s_1,2,c) &\equiv true
\end{aligned}
$$

Abbildung 3.5.: VF2-Beispiel: Zustand s_1 - nach dem Match von 1 auf a

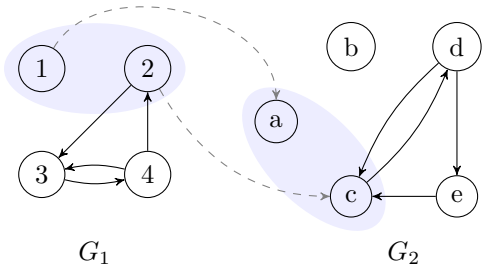
Im Zustand s_1 sind einige Felder mit Werten belegt. Die Einträge in den core-Arrays geben an, auf welchen Knoten des anderen Graphen ein Knoten abgebildet wird. Weil nach den Ordnungsrelationen 1 und a jeweils die ersten Knoten repräsentieren, enthalten die entsprechenden Felder die „1“. Durch das Hinzufügen dieses Paares tritt außerdem der Sonderfall ein, bei dem die gematchten Knoten nicht aus den Terminalmengen stammen. Das würde dazu führen, dass die Felder der in- und out-Arrays mit „0“ belegt wären.

In solch einem Fall werden die Knoten beim Hinzufügen zum Zustand in die eigenen Terminalmengen eingetragen. Die „1“ in den in- und out-Arrays steht allerdings nicht für die Position des Knoten in der Ordnungsrelation, sondern für die Rekursionstiefe in der die Knoten zur jeweiligen Menge hinzukommen. Die Tiefe war im vorherigen Rekursionsschritt die Eins und ist momentan die Zwei.

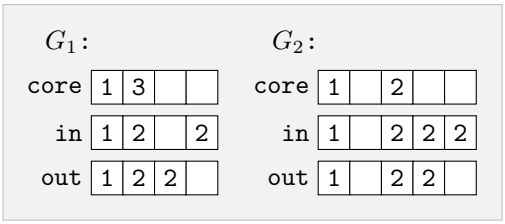
Da die Knoten 1 und a keine Nachbarn haben und dementsprechend die aktuellen Terminalmengen leer sind, läuft die Kandidatengenerierung genauso wie im vorherigen Schritt ab. Eine Besonderheit stellt die fehlschlagende $feasible$-Prüfung von $(2, b)$ dar. Erwartungsgemäß werden die ersten vier Regeln aufgrund fehlender Nachbarn in $M(s_1)$, $T^{in}(s_1)$ und $T^{out}(s_1)$ erfüllt. Allerdings besitzt der Knoten 2 in $\widetilde{V_1}(s_1)$ einen Vorgänger und einen Nachfolger, während b in $\widetilde{V_2}(s_1)$ gar keine Nachbarn vorweisen kann. Da das Paar $(2, c)$ auch die fünfte Regel besteht, geht der VF2 in die nächste Ebene über.

Nach dem Match von $(2, c)$

$$M(s_2) = \{(1, a), (2, c)\} \qquad P(s_2) = \{\min T_1^{out}(s_2)\} \times T_2^{out}(s_2) = \{(3, d)\}$$
$$T_1^{in}(s_2) = \{4\} \qquad T_2^{in}(s_2) = \{d, e\}$$
$$T_1^{out}(s_2) = \{3\} \qquad T_2^{out}(s_2) = \{d\}$$
$$\widetilde{V_1}(s_2) = \emptyset \qquad \widetilde{V_2}(s_2) = \{b\}$$



G_1 $\qquad$ G_2



Zustand s_2

$$\begin{aligned} feasible(s_2, 3, d) &\equiv rule_{pred}(s_2, 3, d) \wedge rule_{succ}(s_2, 3, d) \\ &\quad \wedge rule_{in}(s_2, 3, d) \wedge rule_{out}(s_2, 3, d) \wedge rule_{new}(s_2, 3, d) \\ &\equiv true \wedge false \wedge \ldots \\ &\equiv false \end{aligned}$$

Abbildung 3.6.: VF2-Beispiel: Zustand s_2 - nach dem Match von 2 auf c

Durch das Hinzufügen des Paares $(2, c)$ verändert sich der Zustand stark. Der Wert „3“ im zweiten Feld des G_1-core-Arrays und die „2“ bei G_2 verdeutlichen nochmal das System für die Belegung. Der zweite Knoten aus G_1 wird auf den dritten Knoten von G_2 und der dritte Knoten von G_2 auf den zweiten Knoten von G_1 abgebildet. Die Werte sind also die Positionen der entsprechenden Knoten des anderen Graphen.

Bei den in- und out-Arrays werden die betroffenen Felder für 2 und c wieder gemäß dem Sonderfall behandelt. Dieses Mal enthalten sie allerdings die „2“, da die Einträge in der Rekursionstiefe zwei vorgenommen wurden. Des Weiteren sind noch die Werte für die Nachbarknoten Knoten eingefügt worden. Der aktuelle Match in G_1 deckt die Knoten 1 und 2 ab, wobei 2 eine eingehende Kante von 4 und eine ausgehende Kante nach 3 besitzt. Dieses spiegelt sich in den in- und out-Arrays wieder. Der in-Wert vom Knoten 4 ist die „2“, weil dieser durch das Hinzufügen des Paares $(2, c)$ in T_1^{in} eintrat. Das Gleiche gilt für den Knoten 3 und seinen out-Wert. Bei dem Knoten d des Graphen G_2 kann man erkennen, dass ein Knoten gleichzeitig in T^{in} als auch T^{out} liegen kann.

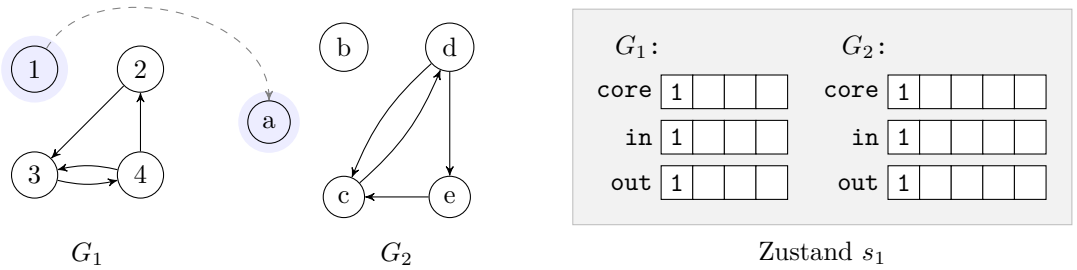
Im Gegensatz zu den ersten beiden Schritten sind die Terminalmengen auf dieser Ebene nicht leer. Bei genauer Betrachtung der angegebenen Mengen und dem aktuellen Zustand stellt man fest, dass ein Knoten genau dann in den Terminalmengen T^{in} oder T^{out} liegt, wenn er noch nicht abgebildet wurde und sein Wert im in- oder out-Array ungleich „0“ ist.

Zur Kandidatengenerierung werden auf dieser Rekursionsebene die Terminalmengen herangezogen. Falls sowohl T^{out} als auch T^{in} für beide Graphen Knoten enthalten, wird T^{out} bevorzugt. Nun fällt auf, dass $P(s_2) = \{(3, d)\}$ nur ein Paar enthält. Es wird also nur dieses eine Paar untersucht, obwohl drei Paare generiert werden können. An dieser Stelle beschneidet der VF2 den Suchraum. Die Terminalmengen umfassen alle an den bisherigen Match angrenzenden Knoten. Jeder gematchte Knoten muss mit seiner Nachbarschaft korrekt abbildbar sein, also muss es für einen Knoten aus dieser Nachbarschaft einen $feasible$ Zielknoten geben. Falls $(3, d)$ fehlschlägt, kann sofort gefolgert werden, dass es für die schon gematchten Knoten keine korrekte Abbildung der Nachbarschaft in diesem Rekursionsast gibt. Also kann man auf das Untersuchen von weiteren Knoten verzichten.

Der Test von $feasible(s_2, 3, d)$ schlägt schon mit der Prüfung der zweiten Regel fehl, da dem Knoten 3 eine Kante zum Knoten 2 fehlt. Anzumerken ist noch, dass der bisherige Match zusammen mit den Terminalmengen erstmalig alle Knoten von G_1 abdeckt, wodurch $\widetilde{V_1}(s_2) = \emptyset$ ist. Dadurch ist die fünfte Regel immer wahr.

Nach dem Backtracking von $(2, c)$

$$M(s_1) = \{(1, a)\} \qquad P(s_1) = \{\min(V_1 - M_1(s_1))\} \times (V_2 - M_2(s_1))$$
$$= \{\cancel{(2, b)}, \cancel{(2, c)}, (2, d), (2, e)\}$$
$$T_1^{in}(s_1) = \emptyset \qquad T_2^{in}(s_1) = \emptyset$$
$$T_1^{out}(s_1) = \emptyset \qquad T_2^{out}(s_1) = \emptyset$$
$$\widetilde{V_1}(s_1) = \{2, 3, 4\} \qquad \widetilde{V_2}(s_1) = \{b, c, d, e\}$$



G_1 G_2 Zustand s_1

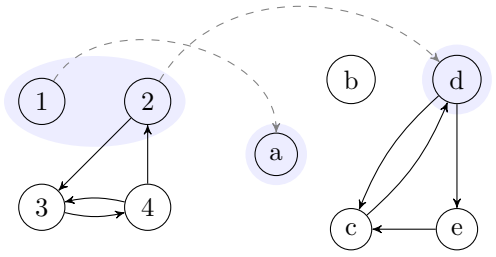
$$feasible(s_1, 2, d) \equiv rule_{pred}(s_1, 2, d) \wedge rule_{succ}(s_1, 2, d)$$
$$\wedge rule_{in}(s_1, 2, d) \wedge rule_{out}(s_1, 2, d) \wedge rule_{new}(s_1, 2, d)$$
$$\equiv true$$

Abbildung 3.7.: VF2-Beispiel: Zustand s_1 - nach Backtracking von $(2, c)$

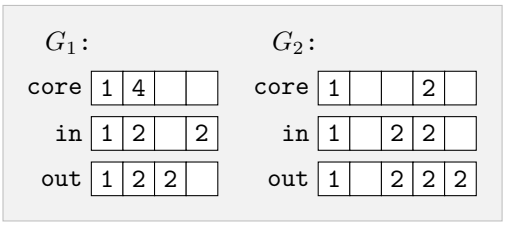
Der Algorithmus befindet sich wieder in der Rekursionstiefe zwei. Durch das Backtracking erhält er die Nachricht, dass der Match über $(2, c)$ fehlgeschlagen ist. Um den Zustand s_1 wiederherzustellen, werden alle auf dieser Ebene vorgenommenen Veränderungen rückgängig gemacht. Dazu werden alle Einträge der aktuellen Rekursionstiefe in den in- und out-Arrays mit „0“ sowie das Paar $(2, c)$ mit $NULL\_NODE$ überschrieben. Nachdem der Zustand s_1 wieder vorliegt, werden die restlichen Kandidaten untersucht. Dabei erscheint $(2, d)$ $feasible$. Bei genauer Betrachtung besitzt d zwar mehr Nachbarn als 2, diese liegen jedoch nicht in $M(s_1)$, weshalb dieser Umstand unproblematisch ist.

Nach dem Match von $(2, d)$

$$M(s_2) = \{(1,a),(2,d)\} \qquad P(s_2) = \{\min T_1^{out}(s_2)\} \times T_2^{out}(s_2) = \{(3,c),(3,e)\}$$
$$T_1^{in}(s_2) = \{4\} \qquad T_2^{in}(s_2) = \{c\}$$
$$T_1^{out}(s_2) = \{3\} \qquad T_2^{out}(s_2) = \{c,e\}$$
$$\widetilde{V_1}(s_2) = \emptyset \qquad \widetilde{V_2}(s_2) = \{b\}$$



G_1 G_2



Zustand s_2

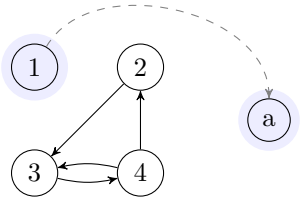
$$\begin{aligned}
feasible(s_2,3,c) &\equiv rule_{pred}(s_2,3,c) \wedge rule_{succ}(s_2,3,c) \\
&\quad \wedge rule_{in}(s_2,3,c) \wedge rule_{out}(s_2,3,c) \wedge rule_{new}(s_2,3,c) \\
&\equiv true \wedge false \wedge \ldots \\
&\equiv false \\
feasible(s_2,3,e) &\equiv true \wedge true \wedge false \wedge \ldots \\
&\equiv false
\end{aligned}$$

Abbildung 3.8.: VF2-Beispiel: Zustand s_2 - nach dem Match von 2 auf d

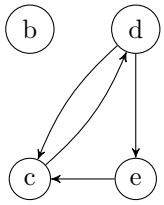
Die Aktualisierung des Zustandes und die Generierung der Kandidatenmenge erfolgt analog zum vorherigen Match von 2 auf c. $P(s_2)$ besteht auf dieser Ebene aus zwei Paaren. Außerdem schlägt $(3,c)$ durch die fehlende Kante von 3 zu 2 fehl. Des Weiteren fällt zum ersten Mal mit $(3,e)$ ein Paar wegen einer Terminalregel durch. Der Knoten 3 hat mit dem Knoten 4 sowohl einen Vorgänger als auch einen Nachfolger in den Terminalmengen, während e nur einen Nachfolger besitzt. Für den Knoten 3 kann also schon ausgeschlossen werden, dass seine Nachbarn aus den Terminalmengen abbildbar sind. Damit sind alle Kandidaten abgearbeitet.

Nach dem Backtracking von $(2, d)$

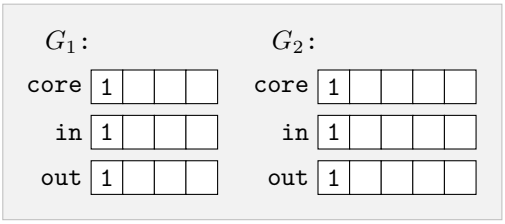
$$M(s_1) = \{(1, a)\} \qquad P(s_1) = \{\min(V_1 - M_1(s_1))\} \times (V_2 - M_2(s_1))$$
$$= \{\cancel{(2, b)}, \cancel{(2, c)}, \cancel{(2, d)}, (2, e)\}$$
$$T_1^{in}(s_1) = \emptyset \qquad T_2^{in}(s_1) = \emptyset$$
$$T_1^{out}(s_1) = \emptyset \qquad T_2^{out}(s_1) = \emptyset$$
$$\widetilde{V_1}(s_1) = \{2, 3, 4\} \qquad \widetilde{V_2}(s_1) = \{b, c, d, e\}$$



G_1



G_2



Zustand s_1

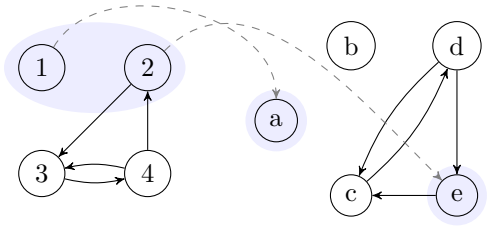
$$feasible(s_1, 2, e) \equiv rule_{pred}(s_1, 2, e) \wedge rule_{succ}(s_1, 2, e)$$
$$\wedge rule_{in}(s_1, 2, e) \wedge rule_{out}(s_1, 2, e) \wedge rule_{new}(s_1, 2, e)$$
$$\equiv true$$

Abbildung 3.9.: VF2-Beispiel: Zustand s_1 - nach Backtracking von $(2, d)$

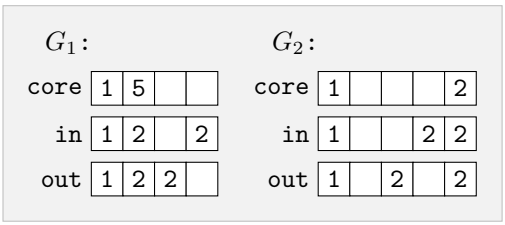
Nach dem Backtracking von $(2, d)$ bleibt auf dieser Ebene nur noch ein letzter Versuch, bis ein neuer Anlauf von Ebene eins gestartet werden muss. Das Paar $(2, e)$ erfüllt allerdings alle Regeln und wird dem aktuellen Zustand hinzugefügt.

Nach dem Match von $(2, e)$

$$M(s_2) = \{(1,a),(2,e)\} \qquad P(s_2) = \{\min T_1^{out}(s_2)\} \times T_2^{out}(s_2) = \{(3,c)\}$$
$$T_1^{in}(s_2) = \{4\} \qquad T_2^{in}(s_2) = \{d\}$$
$$T_1^{out}(s_2) = \{3\} \qquad T_2^{out}(s_2) = \{c\}$$
$$\widetilde{V_1}(s_2) = \emptyset \qquad \widetilde{V_2}(s_2) = \{b\}$$



G_1 G_2



Zustand s_2

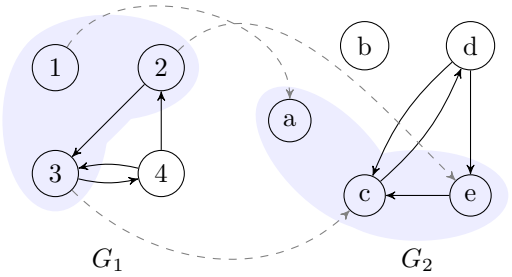
$$\begin{aligned} feasible(s_2, 3, c) &\equiv rule_{pred}(s_2, 3, c) \wedge rule_{succ}(s_2, 3, c) \\ &\quad \wedge rule_{in}(s_2, 3, c) \wedge rule_{out}(s_2, 3, c) \wedge rule_{new}(s_2, 3, c) \\ &\equiv true \end{aligned}$$

Abbildung 3.10.: VF2-Beispiel: Zustand s_2 - nach dem Match von 2 auf e

Wie schon im fünften Schritt bleibt durch die Suchraumbeschneidung per Terminalmenge nur ein Versuch. Allerdings erfüllt $(3, c)$ alle fünf Regeln auf Anhieb.

Nach dem Match von $(3, c)$

$$M(s_3) = \{(1,a),(2,e),(3,c)\} \qquad P(s_3) = \{\min T_1^{out}(s_3)\} \times T_2^{out}(s_3) = \{(4,d)\}$$
$$T_1^{in}(s_3) = \{4\} \qquad T_2^{in}(s_3) = \{d\}$$
$$T_1^{out}(s_3) = \{4\} \qquad T_2^{out}(s_3) = \{d\}$$
$$\widetilde{V_1}(s_3) = \emptyset \qquad \widetilde{V_2}(s_3) = \{b\}$$



G_1:

core	1	5	3	
in	1	2	3	2
out	1	2	2	3

G_2:

core	1		3		2
in	1		3	2	2
out	1		2	3	2

Zustand s_3

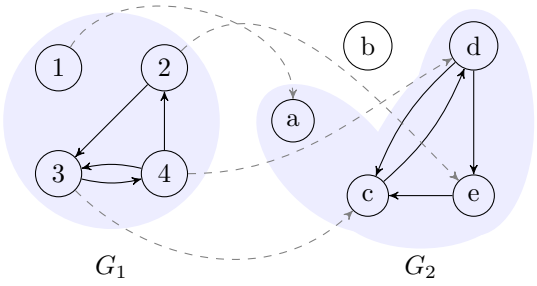
$$\begin{aligned} feasible(s_3,4,d) &\equiv rule_{pred}(s_3,4,d) \wedge rule_{succ}(s_3,4,d) \\ &\quad \wedge rule_{in}(s_3,4,d) \wedge rule_{out}(s_3,4,d) \wedge rule_{new}(s_3,4,d) \\ &\equiv true \end{aligned}$$

Abbildung 3.11.: VF2-Beispiel: Zustand s_3 - nach dem Match von 3 auf c

Bei der Zustandsaktualisierung durch die vorherige dritte Rekursionsebene kann man erkennen, dass bestehende Werte in den Arrays nicht überschrieben werden (dürfen). Dieser Umstand ist zwingend notwendig, da ansonsten bei etwaigem Backtracking ein fehlerhafter Zustand wiederhergestellt wird. Außerdem ist die korrekte Führung der Zu- und Abgänge zu den einzelnen Mengen Voraussetzung für die Repräsentation des Zustandes als SSR und somit für die Reduzierung des Speicherverbrauches.

Der gefundene Match zwischen G_1 und G_2

$$M(s_4) = \{(1,a),(2,e),(3,c),(4,d)\} \qquad P(s_4) = \emptyset$$
$$T_1^{in}(s_4) = \emptyset \qquad T_2^{in}(s_4) = \emptyset$$
$$T_1^{out}(s_4) = \emptyset \qquad T_2^{out}(s_4) = \emptyset$$
$$\widetilde{V_1}(s_4) = \emptyset \qquad \widetilde{V_2}(s_4) = \{b\}$$



G_1:

core	1	5	3	4
in	1	2	3	2
out	1	2	2	3

G_2:

core	1		3	4	2
in	1		3	2	2
out	1		2	3	2

Zustand s_4

Abbildung 3.12.: VF2-Beispiel: der gefundene Match zwischen G_1 und G_2

Nachdem der Teilmatch alle Knoten von G_1 abdeckt, ist ein fertiger Match gefunden worden. Dieser ist in dem core-Array von G_1 nachzulesen. An dem Zustand s_4 kann man erkennen, dass auf der letzten Ebene alle Felder in den Arrays von G_1 besetzt sind, während für G_2 Lücken existieren. Da der Algorithmus eine Untergraphisomorphie sucht, müssen auch nicht alle Knoten von G_2 getroffen werden. An dieser Stelle besteht nun die Möglichkeit, den gefunden Match zu akzeptieren oder ggf. abzulehnen. Der Algorithmus würde bei einer Ablehnung zurücktracken und einen anderen Match suchen. In diesem Beispiel wird der erste gefundene Match akzeptiert.

Der gefundene Match von G_1 in G_2 lautet:

$$M = \{(1,a),(2,e),(3,c),(4,d)\} \qquad \text{bzw.}$$
$$M : 1 \mapsto a$$
$$2 \mapsto e$$
$$3 \mapsto c$$
$$4 \mapsto d$$

4. Adaption des VF2 auf rekonfigurierbare Petrinetze

Der VF2 wird als Basis für die Entwicklung des Matchingalgorithmus verwendet. Im Vergleich zu den anderen vorgestellten Algorithmen besitzt er einige Vorteile.

Bei der Beschreibung der Anforderungen an den späteren Zielalgorithmus sind die einzelnen Kriterien in vier Kategorien eingeteilt worden (siehe Abschnitt 3.1). Die erste Kategorie betrifft das korrekte Matching von Petrinetzen. Durch die Grundstruktur des VF2 lassen sich die Kandidatengenerierung und die $feasible$-Prüfung in einfacher Weise auf die Bedürfnisse der Petrinetze anpassen. Die zweite Kategorie betrifft die Anforderungen aus der Sicht der Regelanwendung. Da das Matching in verschiedenen Kontexten erfolgen kann, muss sich der Zielalgorithmus auf die unterschiedlichen Anforderungen der einzelnen Matchingkontexte einstellen.

Bei der Suche eines Matches in einer Regeldatenbank ist die Gesamtlaufzeit von Bedeutung. Da der VF2 einzelne Graphen betrachtet, müssen die Regeln sequentiell getestet werden. Somit kann die Abhängigkeit zwischen der Datenbankgröße und der Gesamtlaufzeit nicht vermieden werden. Umso wichtiger ist die Laufzeit eines einzelnen Matchingvorganges. Dieser kann sowohl aus der Sicht der erfolgreichen Suche als auch der erfolglosen Suche betrachtet werden. Eine ausführliche Analyse über die Auswirkung des Matchingerfolges auf die Laufzeit wurde von Irniger und Bunke [IB03] durchgeführt. Dazu messen sie die Laufzeiten des Ullmann und des VF2 zur Filterung einer Graphdatenbank mit unterschiedlichen Graphen. Das Ergebnis dieser Untersuchung zeigt ein weitgehend einheitliches Bild. Während der VF2 bei der erfolgreichen Suche vor dem Ullmann liegt, sieht das bei der erfolglosen Suche umgekehrt aus. Allerdings fallen die Abstände im zweiten Fall deutlich geringer aus als im ersten. Im Allgemeinen zeigt der VF2 jedoch ein deutlich stabileres Laufzeitverhalten [vgl. IB03, S. 123 ff.].

Ein anderer Matchingkontext ist die Prüfung der negativen Anwendungsbedingungen. Da diese durchgeführt wird, sobald der Algorithmus für eine Regel einen vermeintlich gültigen Match gefunden hat, erfolgt sie parallel zum Regelmatching. Deshalb muss neben einer guten Laufzeit auch ein geringer Speicherverbrauch vorliegen. Diesen ge-

währleistet der VF2 durch die neu eingeführte Datenstruktur. Außerdem beinhaltet der zu prüfende Match einen Teilmatch der NAC. Dieser kann durch die Freiheiten der Kandidatengenerierung und der Prüfung mittels $feasible$ beim NAC-Matching übernommen werden. Dadurch ist ein effizienter Test der negativen Anwendungsbedingungen möglich.

Aus dem Bereich der Simulation ergibt sich eine weitere Anforderung. Die Bedingung an alle Kandidaten war, dass sie die Fairness auf eine geeignete Weise sicherstellen. Um die Fairness umzusetzen, wurden zwei verschiedene Strategien angesprochen. Die erste, dass der Algorithmus die Menge aller Matches erzeugen können muss, wird direkt erfüllt. Der VF2 kann (als Baumsuchalgorithmus) alle Matches zwischen den Graphen aufzählen. Die zweite Möglichkeit war, die Fairness in Form einer Pseudofairness durch einen nichtdeterministischen Ablauf zu simulieren. Die Einführung von Nichtdeterminismus ist bei dem VF2 durch die Kandidatengenerierung besonders einfach.

Nachdem die Gründe für die Wahl des VF2 dargelegt wurden, muss dieser auf die Petrinetze adaptiert werden. Der dabei entstehende Algorithmus wird im Folgenden als PNVF2 bezeichnet (Petrinetz-VF2).

4.1. Die vorgenommenen Veränderungen

Der PNVF2 sucht eine Untergraphisomorphie zwischen dem Quellnetz N_1 und dem Zielnetz N_2. Im Gegensatz zum VF2 muss der PNVF2 nicht nur für die Existenz der notwendigen Kanten sorgen, sondern auch die Eigenschaften der Netze beachten.

$$N_{i\in\{1,2\}} = (P_i, T_i, pre_i, post_i, m_i, cap_i, pname_i, tname_i, tlb_i, rnw_i)$$

$$pre_i : P_i \times T_i \to \mathbb{N}_0$$

$$post_i : P_i \times T_i \to \mathbb{N}_0$$

Die Definition eines dekorierten Petrinetzes besteht aus einem 10-Tupel. Die ersten beiden Mengen stehen für die Stellen P_i und Transitionen T_i. Die folgenden beiden Funktionen repräsentieren die Vorbereichskanten pre_i und die Nachbereichskanten $post_i$ des Netzes. Die Angaben erfolgen aus der Sicht der Transition. Das ist deshalb möglich, weil Petrinetze bipartite Graphen sind. Das Gewicht einer Kante bildet den dazugehörigen Funktionswert. Falls zwischen zwei Knoten keine Kante besteht, wird das Paar auf den Wert 0 abgebildet. Die nächsten beiden Objekte stehen für die Markierungen m_i und die Kapazitäten cap_i der Stellen. Die Dekorationen des Netzes werden durch die letzten vier Funktionen ausgedrückt. Zuerst werden Namen für die Stellen $pname_i$ und

Transitionen $tname_i$ vergeben und anschließend die Transitionen noch um Labels tlb_i und die dazugehörigen Labelerneuerungsfunktionen rnw_i erweitert.

Der gefundene Match M wird wieder je nach Kontext in der Form einer Funktion $M : P_1 \longrightarrow P_2, T_1 \longrightarrow T_2$ oder Relation $M \subseteq (P_1 \times P_2) \cup (T_1 \times T_2)$ angegeben. Die Zusammenhänge zwischen der Funktions- und Relationsform von M sowie zwischen den beiden Mengen M_1 und M_2 gelten entsprechend.

$$M(v) = w \;\equiv\; (v, w) \in M \qquad\qquad M_1 = \{v \mid (v, w) \in M\}$$
$$M_2 = \{w \mid (v, w) \in M\}$$

Die Grundstruktur des Matchingvorganges entspricht dem des VF2. Durch die Aufteilung der Knoten in Stellen und Transitionen besteht der Suchraum aus einer Kombination zwischen Stellen- und Transitionsabbildungen. Auf jeder Rekursionsebene werden nur Knoten einer Art betrachtet. Die Auswahl der zu untersuchenden Knotenart erfolgt nichtdeterministisch. Nachdem die Kandidaten generiert wurden, prüft der PNVF2 die einzelnen Paare auf ihre Gültigkeit. Wenn ein Stellen- bzw. Transitionspaar die gestellten Anforderungen erfüllt, wird es zum Teilmatch hinzugefügt. Anschließend werden die restlichen Knoten in den folgenden Rekursionsebenen untersucht.

4.1.1. Nichtdeterminismus und Berechnung der Kandidaten

Zur Umsetzung des Nichtdeterminismus muss auf jeder Ebene ein zufälliger freier Quellknoten auf zufällige passende Zielknoten abgebildet werden. Diese Strategie hat allerdings den Nachteil, dass sie sehr ineffizient ist. Bei der Wahl zufälliger Paare ignoriert ein Algorithmus die Struktur der Netze weitgehend. Dadurch schlägt das Matching in höheren Rekursionstiefen öfter fehl. Da die Effizienz durch die Regeldatenbank eine hohe Bedeutung hat, muss eine andere Lösung gefunden werden. Der VF2 verfolgt die Strategie, zusammenhängende Komponenten in das Zielnetz abzubilden. Dazu werden (soweit möglich) Knoten aus den Terminalmengen bevorzugt. Damit der PNVF2 eine angemessene Laufzeit besitzt, wird dieses Verhalten beibehalten. Der Nichtdeterminismus wird wie folgt umgesetzt: Der VF2 definiert als Basis sowohl Ordnungen über die Quellknoten als auch die Zielknoten. Für den PNVF2 müssen bei verschiedenen Matchingvorgängen verschiedene Matches entstehen können. Da die Abbildungsreihenfolge durch die Knotenordnungen bestimmt wird, werden die Stellen und Transitionen vor dem Matching permutiert. Allerdings geschieht das nur auf der Ebene ihrer jeweiligen

Ordnungen. Die Netze bleiben unverändert. Damit hat der Algorithmus die grundlegende Abbildungsreihenfolge der Knoten für den Matchingversuch festgelegt.

Aufgrund der Inkompatibilität der Stellen und Transition werden auf jeder Rekursionsebene nur Kandidaten einer Knotenart untersucht. Die Kandidatengenerierung sieht vor, dass der kleinste freie Quellknoten auf die freien Zielknoten abgebildet wird. Durch die Aufteilung der Knotenmenge sind zwei kleinste Knoten entstanden. An diesem Punkt besteht für den PNVF2 die Gelegenheit die abzubildende Knotenklasse zu wählen. Die Entscheidung ist für die Korrektheit unwichtig. Allerdings wirkt sich die Kandidatenwahl ggf. auf den späteren Match aus, da die Terminalmengen auf der nächsten Ebene anders aussehen können. Bevor die Wahl stattfinden kann, müssen die Terminal- und Restknotenmengen des PNVF2 entsprechend definiert werden.

$$
\begin{aligned}
T^{out}_{i\in\{1,2\}}(s_n) &= T^{out}_{P_i}(s_n) \cup T^{out}_{T_i}(s_n) & \qquad T^{in}_{i\in\{1,2\}}(s_n) &= T^{in}_{P_i}(s_n) \cup T^{in}_{T_i}(s_n) \\
T^{out}(s_n) &= T^{out}_1(s_n) \cup T^{out}_2(s_n) & \qquad \widetilde{P_1}(s_n) &= P_1 - M_1(s_n) - T^{term}_1(s_n) \\
T^{in}(s_n) &= T^{in}_1(s_n) \cup T^{in}_2(s_n) & \qquad \widetilde{T_1}(s_n) &= T_1 - M_1(s_n) - T^{term}_1(s_n) \\
T^{term}_1(s_n) &= T^{out}_1(s_n) \cup T^{in}_1(s_n) & \qquad \widetilde{P_2}(s_n) &= P_2 - M_2(s_n) - T^{term}_2(s_n) \\
T^{term}_2(s_n) &= T^{out}_2(s_n) \cup T^{in}_2(s_n) & \qquad \widetilde{T_2}(s_n) &= T_2 - M_2(s_n) - T^{term}_2(s_n)
\end{aligned}
$$

Die Aufteilung der Knotenmengen setzt sich in den Terminal- und Restknotenmengen fort. Beispielsweise wird die out-Menge von N_1 in eine out-Menge für Stellen $T^{out}_{P_1}$ und für Transition $T^{out}_{T_1}$ aufgespalten. In $T^{out}_{P_1}$ sind die Stellen enthalten, die das Ziel einer im Teilmatch startenden Kante sind. Die anderen Terminal- und Restknotenmengen ergeben sich entsprechend. Mit diesen Informationen kann der PNVF2 die geteilten speziellen Kandidatenmengen generieren.

$$
\begin{aligned}
P_{T^{out}_P}(s_n) &= \{\min T^{out}_{P_1}(s_n)\} \times T^{out}_{P_2}(s_n) & \qquad P_{\widetilde{P}}(s_n) &= \{\min(P_1 - M_1(s_n))\} \\
P_{T^{out}_T}(s_n) &= \{\min T^{out}_{T_1}(s_n)\} \times T^{out}_{T_2}(s_n) & & \quad \times (P_2 - M_2(s_n)) \\
P_{T^{in}_P}(s_n) &= \{\min T^{in}_{P_1}(s_n)\} \times T^{in}_{P_2}(s_n) & \qquad P_{\widetilde{T}}(s_n) &= \{\min(T_1 - M_1(s_n))\} \\
P_{T^{in}_T}(s_n) &= \{\min T^{in}_{T_1}(s_n)\} \times T^{in}_{T_2}(s_n) & & \quad \times (T_2 - M_2(s_n))
\end{aligned}
$$

Nachdem die Mengen vorliegen, muss sich der Algorithmus jeweils für eine Knotenart entscheiden. Daraus resultieren die speziellen Kandidatenmengen $P_{T^{out}}$, $P_{T^{in}}$ und $P_{\widetilde{V}}$.

$$
P_{T^{out}}(s_n) = \begin{cases} P_{T^{out}_P}(s_n) \\ P_{T^{out}_T}(s_n) \end{cases} \qquad
P_{T^{in}}(s_n) = \begin{cases} P_{T^{in}_P}(s_n) \\ P_{T^{in}_T}(s_n) \end{cases} \qquad
P_{\widetilde{V}} = \begin{cases} P_{\widetilde{P}}(s_n) \\ P_{\widetilde{T}}(s_n) \end{cases}
$$

Die Wahrscheinlichkeit der Entscheidung für eine bestimmte Knotenart entspricht ihrem Anteil an der Summe der gesamten Kandidaten. Falls beispielsweise vier Stellen- und acht Transitionspaare für die out-Mengen existieren ($|P_{T_P^{out}}(s_n)| = 4$ und $|P_{T_T^{out}}(s_n)| = 8$), verwendet der Algorithmus mit einer Wahrscheinlichkeit von 33% die Stellen und mit 66% die Transitionen. Die grundlegende Strategie des VF2 zur Wahl der Kandidatenmenge bleibt jedoch erhalten.

$$P(s_n) = \begin{cases} P_{T^{out}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| > 0 \\ P_{T^{in}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| = 0 \wedge |P_{T^{in}}(s_n)| > 0 \\ P_{\widetilde{V}}(s_n) & \text{falls} \quad |P_{T^{out}}(s_n)| = 0 \wedge |P_{T^{in}}(s_n)| = 0 \end{cases}$$

Als Erstes untersucht der PNVF2 die out-Mengen. Falls diese leer sind, geht er zu den Kandidaten der in-Mengen über. Als Letztes werden die übrigen Kandidaten betrachtet. Durch diese Abhängigkeit werden die unteren Kandidatenmengen erst generiert, wenn die oberen leer sind.

4.1.2. Vor- und Nachbereichserhaltung mit $feasible$

Die Erhaltung des Vor- und Nachbereichs der Transitionen ist das wichtigste Kriterium für einen korrekten Match. Bei einer Untergraphisomorphie auf allgemeinen Graphen kann der durch den Match induzierte Untergraph in G_2 zusätzliche Nachbarn besitzen, die nicht im Match enthalten sind. Anders ausgedrückt: Die Terminalmengen von M sind nicht leer. Für die Transitionen ist dieses Verhalten allerdings problematisch. Damit ein Match als korrekt angesehen werden kann, muss der Vor- und Nachbereich einer Transition exakt so erhalten bleiben, wie er im Quellnetz auftaucht. Daher darf die Transition im Zielnetz keine zusätzlichen Nachbarn besitzen.

$$card_{pre_{i\in\{1,2\}}} : T_i \longrightarrow \mathbb{N}_0 \qquad\qquad card_{post_i} : T_i \longrightarrow \mathbb{N}_0$$

Um die gleiche Anzahl an Nachbarn sicherzustellen, werden die Funktionen $card_{pre_i}$ und $card_{post_i}$ eingeführt. Sie bilden eine Transition auf die Anzahl ihrer Nachbarn ab. Also auf die Zahl der Paare (p_i, t_i), die für pre_i und $post_i$ größer sind als null. Im Gegensatz dazu stehen die Stellen. Sie dürfen im Zielnetz zusätzliche Nachbarn besitzen, aber auch das nicht in jedem Fall. Bei Stellen, die für die Erhaltung der Kontaktbedingung relevant sind, muss die gleiche Bedingung erfüllt werden wie bei den Transitionen.

Neben diesen strukturellen Zwängen muss ein Match auch die semantischen Besonderheiten der Petrinetze erfassen. Im Folgenden werden die Markierungen, Kapazitäten, Namen, Labels und die Labelerneuerungsfunktionen als semantische Eigenschaften betrachtet.

$$feasible(s_n, v, w) \equiv \begin{cases} feasible_P(s_n, v, w) & \text{falls } (v, w) \in P_1 \times P_2 \\ feasible_T(s_n, v, w) & \text{falls } (v, w) \in T_1 \times T_2 \end{cases}$$

Durch die Aufteilung der Knoten in zwei Klassen, ihren semantischen Eigenschaften und die unterschiedliche Behandlung der Nachbarn wird $feasible$ in zwei spezielle Prädikate aufgespalten.

$$feasible_P \equiv rule_{sem,P} \wedge rule_{pred,P} \wedge rule_{succ,P} \wedge rule_{in,P} \wedge rule_{out,P} \wedge rule_{new,P}$$
$$feasible_T \equiv rule_{sem,T} \wedge rule_{pred,T} \wedge rule_{succ,T} \wedge rule_{in,T} \wedge rule_{out,T} \wedge rule_{new,T}$$

Falls die zu untersuchenden Kandidaten ein Stellenpaar sind, verwendet der PNVF2 $feasible_P$ und im Falle der Transitionen $feasible_T$. Vor der Prüfung der strukturellen Korrektheit wird zuerst die semantische Abbildbarkeit getestet. Dabei gelten für die Stellen und Transitionen verschiedene Definitionen, ab wann sie abbildbar sind.

$$\begin{aligned} rule_{sem,P} \equiv\ & m_1(p_1) \leq m_2(p_2) \\ & \wedge\, cap_1(p_1) = cap_2(p_2) \\ & \wedge\, pname_1(p_1) = pname_2(p_2) \end{aligned} \qquad \begin{aligned} rule_{sem,T} \equiv\ & card_{pre_1}(t_1) = card_{pre_2}(t_2) \\ & \wedge\, card_{post_1}(t_1) = card_{post_2}(t_2) \\ & \wedge\, tname_1(t_1) = tname_2(t_2) \\ & \wedge\, tlb_1(t_1) = tlb_2(t_2) \\ & \wedge\, rnw_1(t_1) = rnw_2(t_2) \end{aligned}$$

Während die Regel für die Transitionen $rule_{sem,T}$ im jedem Matchingkontext in dieser Form verwendet wird, ist die Definition von $rule_{sem,P}$ weniger stabil. Beispielsweise kann ein Match mit exakter Tokenübereinstimmung benötigt werden. In diesem Fall muss die $\leq$-Beziehung durch die Gleichheit ersetzt werden. Ein anderer Einsatzbereich ist das Matching bei der Regelanwendung. Um die Kontaktbedingung der einzelnen Stellen einzuhalten, wird $rule_{sem,P}$ für die entsprechenden Stellen um die Kantenbedingung ($card$) ergänzt. Zur Vorstellung des Algorithmus wird im Folgenden aber von dem allgemeinen Matching ausgegangen.

Bei der Betrachtung der semantischen Regeln fällt auf, dass für jedes Paar eine Vielzahl von Kriterien getestet werden müssen. Da der PNVF2 eine rekursive Baumsuche betreibt, werden die gleichen Kandidatenpaare immer wieder (über verschiedene Pfade) untersucht. Das mehrfache Prüfen der semantischen Abbildbarkeit ist jedoch äußerst ineffizient. Außerdem sind bei stark gelabelten Petrinetzen ein Großteil der Knoten inkompatibel. Irniger und Bunke zeigen in ihrer Analyse, dass der Ullmann im Allgemeinen eine bessere Laufzeit bei der erfolglosen Suche aufweist [vgl. IB03, S. 123 ff.]. Um das wiederholte Prüfen zu vermeiden und schon vor dem Matching erkennen zu können, ob überhaupt ein Match existieren kann, werden Abbildbarkeitsmatrizen aufgestellt. Der Algorithmus unterscheidet zwischen einer $|P_1| \times |P_2|$-Stellen- und $|T_1| \times |T_2|$-Transitionsabbildbarkeitsmatrix. Die Werte der Matrizen zeigen an, ob der jeweilige Quellknoten aus N_1 auf den entsprechenden Zielknoten aus N_2 abgebildet werden kann. Dazu gelten dieselben Bedingungen, die eben für $rule_{sem,P}$ und $rule_{sem,T}$ beschrieben wurden. Mit der Hilfe dieser Matrizen kann auf die Kompatibilität zweier Netze geschlossen werden. Da es für jeden Quellknoten einen passenden Zielknoten geben muss, darf keine Zeile die Summe null besitzen. Falls das doch der Fall ist, kann auf das Matching verzichtet werden. Aber auch der umgekehrte Fall garantiert keine erfolgreiche Suche, da die Matrizen nicht die Struktur erfassen. Dies erfolgt erst beim Matching. Um die Laufzeit zu verbessern, wird die aufwändige Prüfung in den Regeln $rule_{sem,P}$ und $rule_{sem,T}$ durch einen Zugriff auf die entsprechende Matrix ersetzt.

Nachdem ein Kandidatenpaar die Überprüfung der semantischen Abbildbarkeit bestanden hat, wird die strukturelle Eignung getestet. Dazu werden die Vorgänger und Nachfolger der Knoten untersucht.

$$rule_{pred,P}(s_n, p_1, p_2) \equiv \forall t_1 \in M_1(s_n) \cap T_1 : post_1(p_1, t_1) = post_2(p_2, M(s_n)(t_1))$$
$$rule_{pred,T}(s_n, t_1, t_2) \equiv \forall p_1 \in M_1(s_n) \cap P_1 : pre_1(p_1, t_1) = pre_2(M(s_n)(p_1), t_2)$$

$$rule_{succ,P}(s_n, p_1, p_2) \equiv \forall t_1 \in M_1(s_n) \cap T_1 : pre_1(p_1, t_1) = pre_2(p_2, M(s_n)(t_1))$$
$$rule_{succ,T}(s_n, t_1, t_2) \equiv \forall p_1 \in M_1(s_n) \cap P_1 : post_1(p_1, t_1) = post_2(M(s_n)(p_1), t_2)$$

Die Regeln der beiden Knotenarten unterscheiden sich kaum. Der Algorithmus wechselt lediglich die Perspektive und verwendet zur Prüfung der Vorgänger der Stellen die Funktion $post_i$ und für die Transitionen die Funktion pre_i. Da die Kanten im Petrinetz aus der Sicht der Transition angegeben werden, ist diese Vorgehensweise notwendig. Die Gewichte der Kanten werden während des Vergleichs direkt mit geprüft. Anzumerken ist

noch, dass der bisherige Teilmatch $M(s_n)$ in den Regeln auf zwei verschiedene Weisen verwendet wird. Zum einen wird (beispielsweise in $rule_{pred,P}$) die Menge der bisher abgebildeten Transition $M_1(s_n) \cap T_1$ betrachtet und zum anderen der Match in seiner Funktionsform $M(s_n)(t_1)$ genutzt.

Um während des Matchingvorganges den Suchraum zu beschneiden, müssen noch die letzten drei Regeln betrachtet werden. Allerdings werden hier nur die beiden Spezialisierungen von $rule_{out}$ aufgeführt, da die anderen beiden Regeln einfach durch die Ersetzung von T^{out} mit T^{in} und $\widetilde{P}$ bzw. $\widetilde{T}$ hergeleitet werden können.

$$
\begin{aligned}
&rule_{out,P}(s_n, p_1, p_2) \\
&\quad \equiv |\{t_1 \mid t_1 \in T^{out}_{T_1} \wedge post_1(p_1, t_1) > 0\}| \leq |\{t_2 \mid t_2 \in T^{out}_{T_2} \wedge post_2(p_2, t_2) > 0\}| \\
&\quad \wedge |\{t_1 \mid t_1 \in T^{out}_{T_1} \wedge pre_1(p_1, t_1) > 0\}| \leq |\{t_2 \mid t_2 \in T^{out}_{T_2} \wedge pre_2(p_2, t_2) > 0\}|
\end{aligned}
$$

$$
\begin{aligned}
&rule_{out,T}(s_n, t_1, t_2) \\
&\quad \equiv |\{p_1 \mid p_1 \in T^{out}_{P_1} \wedge pre_1(p_1, t_1) > 0\}| = |\{p_2 \mid p_2 \in T^{out}_{P_2} \wedge pre_1(p_2, t_2) > 0\}| \\
&\quad \wedge |\{p_1 \mid p_1 \in T^{out}_{P_1} \wedge post_1(p_1, t_1) > 0\}| = |\{p_2 \mid p_2 \in T^{out}_{P_2} \wedge post_2(p_2, t_2) > 0\}|
\end{aligned}
$$

Wie schon bei den Vorgänger- und Nachfolgerregeln werden die Funktionen der Kanten in umgekehrter Reihenfolge betrachtet. Ein bedeutsamer Unterschied zwischen $rule_{out,P}$ und $rule_{out,T}$ ist die Art, wie die Anzahl der Nachbarn verglichen wird. Da die Untergraphisomorphie zwischen Petrinetzen an den Stellen ansetzt, sind für sie zusätzliche Nachbarn erlaubt. Bei allen Regeln, die die Transitionen betreffen, wird jedoch die Gleichheit gefordert. Dies erscheint im ersten Moment überflüssig zu sein, da die Erhaltung des Vor- und Nachbereichs schon mit der Abbildbarkeitsmatrix sichergestellt wird. Allerdings dienen die drei Regeln mehr der Suchraumbeschneidung als der Korrektheit. Sie fordern eine viel feingliedrige Gleichheit der Vorgänger und Nachfolger.

4.1.3. Aktualisierung des Zustandes

Der PNVF2 behält als Datenstruktur die State-Space-Representation (SSR) bei. Der einzige Unterschied besteht darin, dass je Netz sechs Arrays verwaltet werden. Dazu werden die core- und die in- und out-Arrays für die Stellen und Transitionen aufgespalten. Bei der Verwaltung der Terminalarrays gibt es keine Veränderung. Alleinig bei den core-Arrays muss die Verwendung der richtigen Ordnung beachtet werden. Die genaue Verwaltung der SSR wird in dem folgenden Beispiel illustriert.

4.2. Beispiel

4.2.1. Start des Matchings von N_1 nach N_2

$$M(s_0) = \emptyset$$

$$P_P(s_0) = \{\min(P_1 - M_1(s_0))\} \times (P_2 - M_2(s_0)) = \{(p_1, p_a), (p_1, p_b), (p_1, p_c)\}$$

$$P_T(s_0) = \{\min(T_1 - M_1(s_0))\} \times (T_2 - M_2(s_0)) = \{(t_1, t_a), (t_1, t_b), (t_1, t_c), (t_1, t_d)\}$$

$$\xrightarrow[\text{nichtdeterministisch}]{|P_P(s_0)| > 0 \wedge |P_T(s_0)| > 0} P(s_0) = P_P(s_0) = \{(p_1, p_a), (p_1, p_b), (p_1, p_c)\}$$

$$T^{in}_{P_1}(s_0) = \emptyset \qquad T^{out}_{P_1}(s_0) = \emptyset \qquad \widetilde{P_1}(s_0) = \{p_1, p_2\}$$

$$T^{in}_{T_1}(s_0) = \emptyset \qquad T^{out}_{T_1}(s_0) = \emptyset \qquad \widetilde{T_1}(s_0) = \{t_1, t_2, t_3\}$$

$$T^{in}_{P_2}(s_0) = \emptyset \qquad T^{out}_{P_2}(s_0) = \emptyset \qquad \widetilde{P_2}(s_0) = \{p_a, p_b, p_c\}$$

$$T^{in}_{T_2}(s_0) = \emptyset \qquad T^{out}_{T_2}(s_0) = \emptyset \qquad \widetilde{T_2}(s_0) = \{t_a, t_b, t_c, t_d\}$$

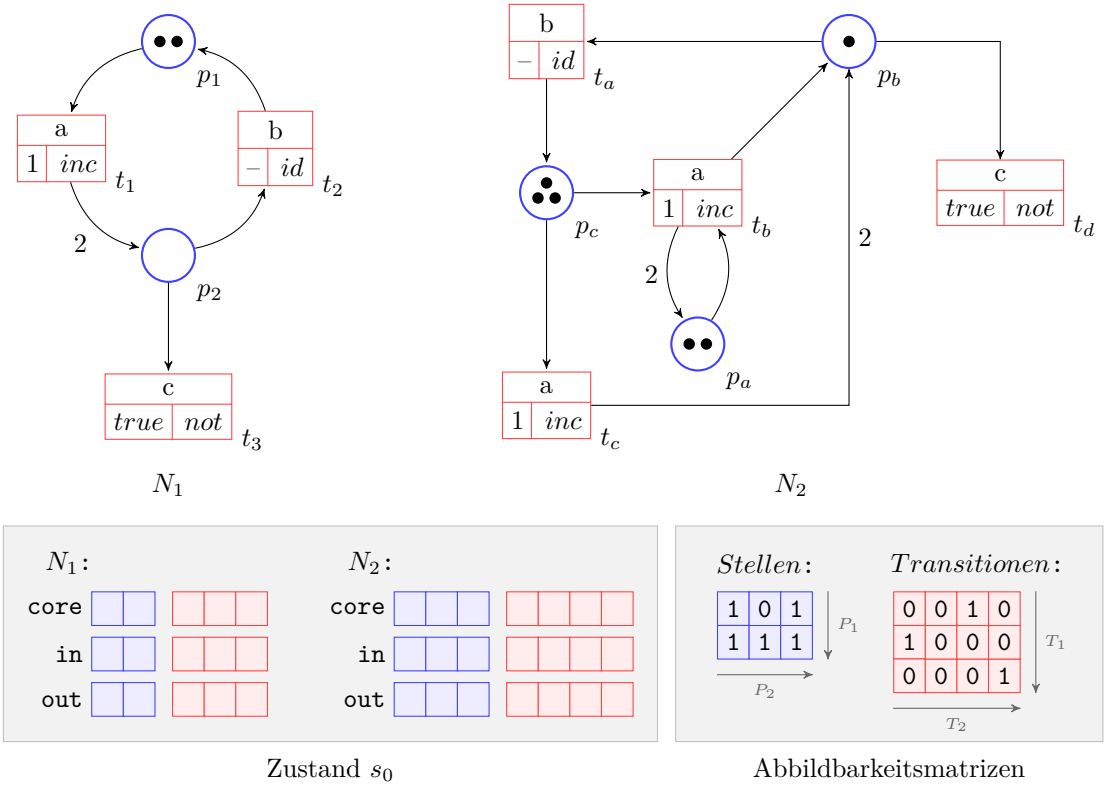


Abbildung 4.1.: PNVF2-Beispiel: Start des Matchings von N_1 nach N_2

In diesem Beispiel (siehe Abbildung 4.1) soll ein Vorkommen von N_1 in N_2 gefunden werden. Im Gegensatz zu dem in Abschnitt 3.3 vorgestellten VF2 und dem dabei durchgespielten Beispielablauf (siehe Unterabschnitt 3.3.2) werden jedoch Petrinetze gematcht.

Abbildung 4.2.: Notation des Beispiels

Die blauen Knoten repräsentieren die Stellen. Die dazugehörigen core-, in- und out-Arrays sowie die Abbildbarkeitsmatrix sind ebenfalls blau gekennzeichnet. Die Token (m) werden durch schwarze Kugeln dargestellt. Um das Beispiel nicht zu komplex werden zu lassen, wird auf die Kapazitäten (cap) und die Namen ($pname$) der Stellen verzichtet.

Die roten Kästchen repräsentieren die Transitionen. Im oberen Teil des Kästchens steht der Name ($tname$), links darunter das Label (tlb) und rechts die Labelerneuerungsfunktion (rnw) der Transition. Die verwendeten Funktionen lauten:

$id(x) = x$	Identität
$inc(x) = x + 1$	Zähler
$not : false \mapsto true,$ $true \mapsto false$	Schalter

Kanten ohne ausgezeichnetes Gewicht werden mit 1 angesetzt. Wie schon der VF2 benötigt der PNVF2 ebenfalls Ordnungen über die Knoten. Da Stellen und Transitionen nicht aufeinander abbildbar sind, werden für beide Knotenklassen verschiedene Ordnungen verwendet. Die Position eines Knotens wird rechts unten neben dem Knoten angezeigt. Beispielsweise hat im ersten Netz die Transition mit dem Namen b die Position 2. Um zu verdeutlichen, dass sich die vier Ordnungen nicht überschneiden, sind die Positionen der Stellen mit dem Prefix p und die der Transitionen mit t versehen. Außerdem werden im zweiten Netz Buchstaben anstelle von Ziffern verwendet. Ein weiterer Unterschied stellt das Zustandekommen der Ordnungen dar. Während der VF2 jede beliebige Ordnung als geeignet ansieht, fordert der PNVF2 vor jedem Matchingversuch ein Mischen der Knoten. Eine über die Lebenszeit der Netze stabile Ordnung erfüllt also die Forderung des VF2. Die Verwendung der gleichen Strategie für den PNVF2 führt im ungünstigsten

Fall dazu, dass immer wieder derselbe Match gefunden wird. Der PNVF2 soll jedoch bei wiederholten Matchingversuchen verschiedene Ergebnisse liefern können. Das Mischen der Knoten stellt damit das erste Standbein für den Nichtdeterminismus des Algorithmus dar. Die im Beispiel verwendete Ordnung ist folglich nur für diesen Versuch gültig.

Die vorberechneten Abbildbarkeitsmatrizen dienen der Vereinfachung der semantischen Prüfung während des Matchings. Der Aufwand für das erneute Prüfen derselben Kandidatenpaare wird damit auf einen Vergleich reduziert. Ein Stellenpaar besitzt genau dann den Eintrag *true* (dargestellt durch: 1), wenn die Namen und Kapazitäten übereinstimmen sowie die Markierung der Stelle aus dem Quellnetz kleiner oder gleich der Markierung der Zielstelle ist. Bei den Transitionen müssen die Namen, Labels und die Labelerneuerungsfunktionen übereinstimmen sowie beide Transitionen die gleiche Anzahl an Vorgängern und Nachfolgern besitzen. Das Gewicht der inzidenten Kanten und die Nachbarn der Knoten werden nicht betrachtet. Dies geschieht während des Matchings.

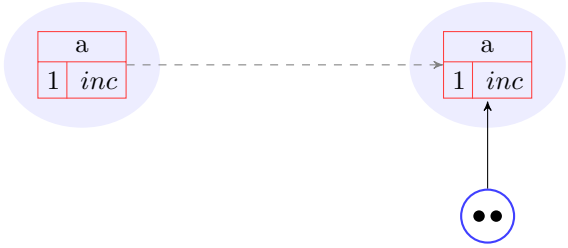


Abbildung 4.3.: ungültiger Match

Der Grund, warum bei den Transitionen die Anzahl der Nachbarn betrachtet wird, liegt in der Forderung der Vor- und Nachbereichserhaltung begründet. Für eine Untergraphisomorphie können die Knoten im Zielnetz zusätzliche Kanten zu Knoten besitzen, die nicht im fertigen Match enthalten sind (siehe Abbildung 4.3). Das kommt einer Vor- bzw. Nachbereichsveränderung gleich. Deshalb müssen alle Vorgänger und Nachfolger einer Transition im Match enthalten und damit die Anzahl gleich sein.

Die grundsätzliche Kandidatengenerierung erfolgt genauso wie beim VF2. Allerdings werden für die Stellen und Transitionen eigene Kandidatenmengen erzeugt. Falls beide Mengen Paare enthalten, entscheidet sich der Algorithmus nichtdeterministisch für eine der beiden. Die Festlegung auf eine der beiden Mengen beeinträchtigt nicht den Matchingvorgang. Der VF2 sieht alle Knoten als gleich an und unterwirft damit alle Knoten des Quellnetzes derselben Ordnung. Danach versucht er den kleinsten freien

Knoten des Quellnetzes auf alle freien Knoten des Zielnetzes abzubilden. Bei dem PNVF2 entstehen jedoch durch die Aufteilung der Knotenmenge und damit der Erzeugung von zwei Ordnungen auch bis zu zwei kleinste Knoten (eine Stelle und eine Transition). Für die Traversierung des Suchraumes wird aber genau ein kleinster Knoten benötigt. Deshalb kann und muss sich der Algorithmus für eine Menge entscheiden. Dies stellt das zweite Standbein für den Nichtdeterminismus des Algorithmus dar. Die Abbildung 4.1 verdeutlicht diesen Vorgang durch die Implikation unter den beiden speziellen Kandidatenmengen. Hervorzuheben ist außerdem, dass sich die Stellen- und Transitionskandidatenmengen bei der Kandidatengenerierung auf dieselben Knotenmengenarten beziehen. Die Strategie des VF2 sieht vor, dass zuerst die $T^{out}_{i\in\{1,2\}}$-, dann (falls wenigstens eine der beiden T^{out} Mengen leer ist) die T^{in}_i- und abschließend die Restknotenmengen untersucht werden. Der PNVF2 behält dieses Verhalten bei, daher werden entweder „$T^{out}_{P_i}$ und $T^{out}_{T_i}$“, „$T^{in}_{P_i}$ und $T^{in}_{T_i}$“ oder „$P_i - M_i$ und $T_i - M_i$“ untersucht.

Da bei der Prüfung von Kandidatenpaaren für die verschiedenen Knotenarten unterschiedliche Regeln genutzt werden, muss bei $feasible$ der Typ der beteiligten Knoten beachtet werden. Deshalb wird zwischen einem $feasible$ für Stellen ($feasible_P$) und Transitionen ($feasible_T$) unterschieden. Beide setzen sich aus sechs einzelnen Regeln zusammen.

$$feasible_P \equiv rule_{sem,P} \wedge rule_{pred,P} \wedge rule_{succ,P} \wedge rule_{in,P} \wedge rule_{out,P} \wedge rule_{new,P}$$
$$feasible_T \equiv rule_{sem,T} \wedge rule_{pred,T} \wedge rule_{succ,T} \wedge rule_{in,T} \wedge rule_{out,T} \wedge rule_{new,T}$$

Die erste Regel ($rule_{sem}$) prüft die semantische Abbildbarkeit des Paares durch einen Zugriff auf die jeweilige Abbildbarkeitsmatrix. Danach werden die fünf aus dem VF2 bekannten Regeln betrachtet. Die Vorgänger- und Nachfolgerregeln ($rule_{pred}$ und $rule_{succ}$) sind für beide Knotenarten abstrakt gesehen identisch. Bei den Regeln für neue Knoten ($rule_{new}$) und den Terminalregeln ($rule_{in}$ und $rule_{out}$) sieht das anders aus. Während es für Stellen ausreicht, dass das Ziel wenigstens so viele Vorgänger und Nachfolger besitzt wie die Quelle, müssen diese Werte bei Transitionen exakt übereinstimmen.

Bei einem Vergleich der Transitionsabbildbarkeitsmatrix mit den Regeln $rule_{in,T}$, $rule_{out,T}$ und $rule_{new,T}$ fällt auf, dass sowohl in den Regeln als auch in der Matrix die gleiche Anzahl an Nachbarn gefordert wird. Dies erscheint im ersten Moment redundant. Allerdings besitzen die Prüfungen verschiedene Aufgaben. Bei der Generierung der Matrix wird nur die Anzahl der Nachbarn betrachtet und nicht die Nachbarn selbst. Durch den Test auf $true$ wird sicher gestellt, dass nur die Transitionenpaare weiter

geprüft werden, bei denen eine Chance darauf besteht, dass sie $feasible_T$ erfüllen können. So wird die teure Prüfung der Vorgänger und Nachfolger vermieden.

Die Forderungen von $rule_{in,T}$, $rule_{out,T}$ und $rule_{new,T}$ sind feingliedriger. Sie besagen unter anderem, dass der Zielknoten exakt die gleiche Menge an Nachbarn in den Terminalmengen besitzen muss. Damit kann man zwar auch die Vor- und Nachbereichserhaltung sicherstellen, dies dient aber primär der Suchraumbeschneidung.

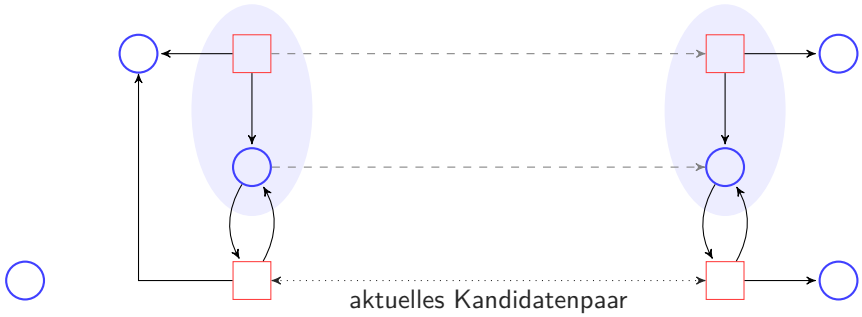


Abbildung 4.4.: unterschiedliche Anzahl an Nachbarn in den Terminalmengen

Beispielsweise könnten zwei Transitionen von der Menge der Nachbarn her aufeinander abbildbar sein, aber die eine könnte mehr Nachbarn in den Terminalmengen aufweisen als die andere (siehe Abbildung 4.4). So kann frühzeitig ausgeschlossen werden, dass die Transitionen bezüglich dem bisherigen Match kompatibel sind.

$$\begin{aligned} feasible_P(s_0, p_1, p_a) \equiv\ & rule_{sem,P}(s_0, p_1, p_a) \wedge rule_{pred,P}(s_0, p_1, p_a) \\ & \wedge rule_{succ,P}(s_0, p_1, p_a) \wedge rule_{in,P}(s_0, p_1, p_a) \\ & \wedge rule_{out,P}(s_0, p_1, p_a) \wedge rule_{new,P}(s_0, p_1, p_a) \\ \equiv\ & true \wedge true \wedge true \wedge true \wedge true \wedge true \\ \equiv\ & true \end{aligned}$$

Für diesen Matchingschritt wird angenommen, dass die Wahl auf die Kandidatenmenge $P_P(s_0)$ fällt. Somit sind auf dieser Ebene die Paare $\{(p_1, p_a), (p_1, p_b), (p_1, p_c)\}$ zu untersuchen (siehe Abbildung 4.1). Das erste Stellenpaar (p_1, p_a) erfüllt direkt – das in Unterabschnitt 4.1.2 vorgestellte – $feasible_P$. Durch den leeren Match besitzen momentan beide Stellen weder Vorgänger noch Nachfolger. Deshalb sind die unterschiedlichen Gewichte der angrenzenden Kanten irrelevant. Sie werden erst in einem späteren Schritt durch $rule_{pred,T}$ und $rule_{succ,T}$ betrachtet.

4.2.2. Nach dem Match von (p_1, p_a)

$$M(s_1) = \{(p_1, p_a)\}$$
$$P_P(s_1) = \{\min T_{P_1}^{out}(s_1)\} \times T_{P_2}^{out}(s_1) = \emptyset$$
$$P_T(s_1) = \{\min T_{T_1}^{out}(s_1)\} \times T_{T_2}^{out}(s_1) = \{(t_1, t_b)\}$$
$$\xrightarrow[\text{~~nicht~~deterministisch}]{|P_P(s_1)| = 0 \wedge |P_T(s_1)| > 0} P(s_1) = P_T(s_1) = \{(t_1, t_b)\}$$

$$T_{P_1}^{in}(s_1) = \emptyset \qquad T_{P_1}^{out}(s_1) = \emptyset \qquad \widetilde{P_1}(s_1) = \{p_2\}$$
$$T_{T_1}^{in}(s_1) = \{t_2\} \qquad T_{T_1}^{out}(s_1) = \{t_1\} \qquad \widetilde{T_1}(s_1) = \{t_3\}$$
$$T_{P_2}^{in}(s_1) = \emptyset \qquad T_{P_2}^{out}(s_1) = \emptyset \qquad \widetilde{P_2}(s_1) = \{p_b, p_c\}$$
$$T_{T_2}^{in}(s_1) = \{t_b\} \qquad T_{T_2}^{out}(s_1) = \{t_b\} \qquad \widetilde{T_2}(s_1) = \{t_a, t_c, t_d\}$$

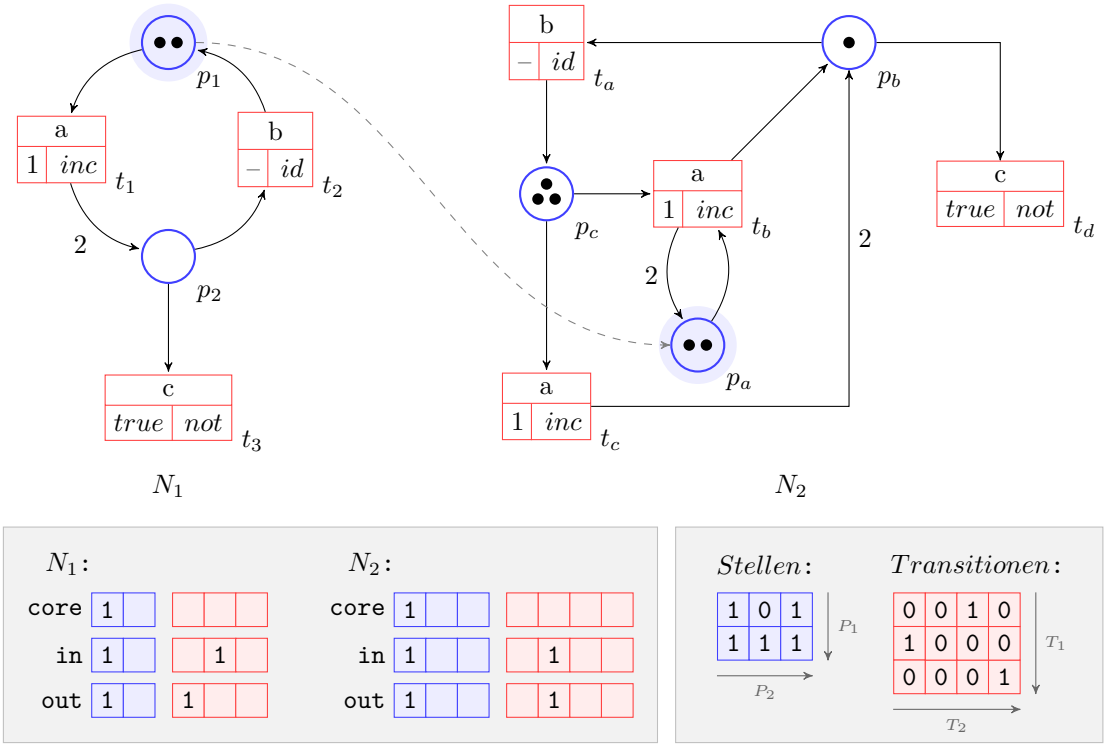


Abbildung 4.5.: PNVF2-Beispiel: nach dem Match von p_1 auf p_a

Die Arrays und die daraus resultierenden Mengen werden analog zum VF2 verwaltet. Mit der Hilfe der „1“ in den beiden blauen core-Arrays zeigt die jeweilige Stelle auf die Position der dazugehörigen Stelle im anderen Netz. Die Ziffern aller vier core-Arrays beziehen sich folglich auf die Positionen der Knoten in dem entsprechenden anderen gleichfarbigen core-Array und damit der Position des Knoten in seiner Ordnungsrelation. Falls also t_3 auf t_d abgebildet würde, enthielte das dritte Feld im ersten roten Array die „4“ und das vierte Feld im rechten Array die „3“ (siehe Abbildung 4.6).

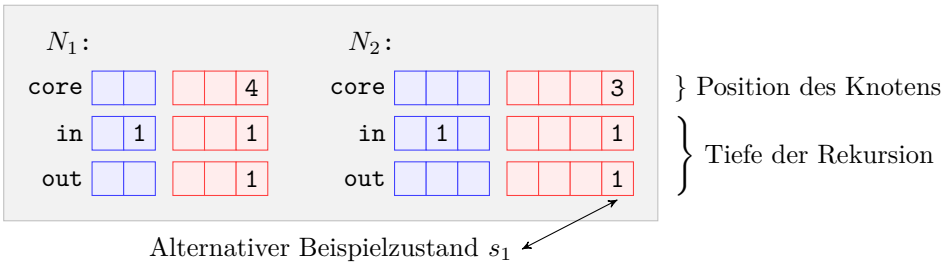


Abbildung 4.6.: Aufbau des Zustandes beim PNVF2

Die Arrays für die Terminalmengen verhalten sich identisch zum VF2. Demnach steht ein Eintrag für die Tiefe, in der der jeweilige Knoten in die Terminalmenge eintritt. Die Tiefe entspricht dabei der Nummer des wievielten Paares, welches gematcht werden soll. Außerdem entspricht die Nummer des entstehenden Zustandes s_n der Tiefe. In diesem (dem zweiten) Schritt ist die Tiefe die Zwei. Falls ein Knotenpaar gematcht wird, welches nicht aus den Terminalmengen stammt, werden die dazugehörigen Einträge in den Terminalarrays gesetzt. Diesen Vorgang kann man bei den in Abbildung 4.5 abgebildeten Stellen beobachten. Die Elemente der aktuellen Terminalmengen ergeben sich aus der Kombination der core- und Terminalarrays. Ein Knoten ist genau dann in einer bestimmten Terminalmenge enthalten, wenn er noch nicht abgebildet wurde und sein Eintrag in dem dazugehörigen Terminalarray gesetzt ist.

Durch die Abbildung von p_1 auf p_a beinhalten auf dieser Ebene einige Terminalmengen Knoten (siehe Abbildung 4.5). Der Algorithmus untersucht zur Kandidatengenerierung zuerst die out-Mengen. Für die Stellen ist das Kreuzprodukt aus $\min T_{P_1}^{out}(s_1)$ und $T_{P_2}^{out}(s_1)$ leer, aber für die Transitionen kann ein Paar mit $P_T(s_1) = \{(t_1, t_b)\}$ gefunden werden. Falls eine der beiden speziellen Kandidatenmengen leer ist, verwendet der Algorithmus deterministisch die nicht leere Menge. Deshalb gilt $P(s_1) = P_T(s_1) = \{(t_1, t_b)\}$. Erst wenn die Kandidatenmengen beider Knotenarten leer sind, greift der

PNVF2 auf die in- und danach schließlich die Restknotenmengen zurück. Dieses Vorgehen kann man direkt auf den VF2 zurückführen. Wenn man von den beiden verschiedenen Knotenarten abstrahiert, ergibt sich für N_1 eine einzelne out-Menge T_1^{out}, die sowohl die Stellen als auch die Transitionen enthält. Die Strategie des VF2 besagt, dass erst auf T^{in} zurückgegriffen wird, wenn mindestens eine out-Menge und damit die aus $minT_1^{out} \times T_2^{out}$ resultierende Kandidatenmenge leer ist. Daher greift der PNVF2 frühestens auf die in-Mengen zurück, wenn für die beiden speziellen Kandidatenmengen keine Paare generiert werden können.

Nachdem feststeht, dass es nur ein Kandidatenpaar gibt, beginnt der PNVF2 mit dem Test. Wie schon erwähnt, unterscheiden sich die Regeln für die Stellen und Transitionen. Die Prüfung des Paares auf $feasible_T$ scheitert schon mit dem Zugriff auf die Transitionsabbildbarkeitsmatrix.

$$\begin{aligned} feasible_T(s_1, t_1, t_b) \equiv\ & rule_{sem,T}(s_1, t_1, t_b) \wedge rule_{pred,T}(s_1, t_1, t_b) \\ & \wedge rule_{succ,T}(s_1, t_1, t_b) \wedge rule_{in,T}(s_1, t_1, t_b) \\ & \wedge rule_{out,T}(s_1, t_1, t_b) \wedge rule_{new,T}(s_1, t_1, t_b) \\ \equiv\ & false \wedge \ldots \\ \equiv\ & false \end{aligned}$$

Zwar sind die Transitionen von den Namen, den Labels und den Labelerneuerungsfunktionen her identisch, sie besitzen allerdings nicht die gleiche Anzahl an Nachbarn. Da die Kandidatenmenge erschöpft ist, kehrt der PNVF2 wieder auf die erste Ebene zurück.

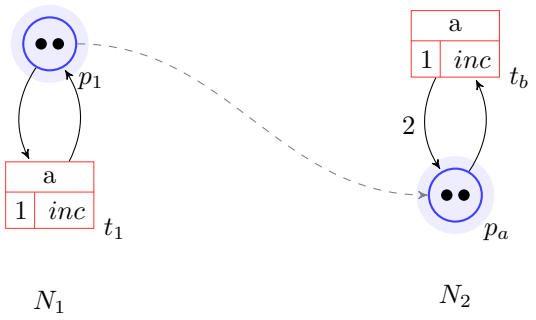


Abbildung 4.7.: Matching mit Kantengewichten

Um allerdings die Behandlung der Kantengewichte zu besprechen, nehmen wir an, dass die beiden Transitionen von der Nachbarschaft her fast kompatibel wären, wobei im

Zielnetz nur ein Kantengewicht anders wäre (siehe Abbildung 4.7). In dem Fall müssten $rule_{pred,T}(s_1, t_1, t_b)$ und $rule_{succ,T}(s_1, t_1, t_b)$ erfüllt werden. Nehmen wir ergänzend an, dass die Vorgängerkanten in beiden Netzen kein Problem darstellen, also käme es nur auf die beiden Nachfolgerkanten zu (p_1, p_a) an. Der VF2 besagt, dass die schon gematchten Nachfolger des Quellknotens auf Nachfolger des Zielknotens abgebildet sein müssen und umgekehrt. Der PNVF2 spezialisiert diese Forderung nun so, dass zusätzlich die Gewichte der Kanten übereinstimmen müssen. Also würde das angenommene Transitionspaar bei $rule_{succ,T}(s_1, t_1, t_b)$ durchfallen. Somit wird nicht nur für die korrekte Anzahl der Vor- und Nachbereichskanten, sondern auch für die Vor- und Nachbereichserhaltung der Transitionen gesorgt.

4.2.3. Zwischenschritte

Auf die detaillierte Darstellung der Zwischenschritte wird im Folgenden verzichtet. Sie lassen sich leicht aus den dazwischenliegenden Zuständen herleiten. Nachfolgend wird in den Abbildungen 4.8 bis 4.13 beispielhaft eine Zustandsabfolge dargestellt.

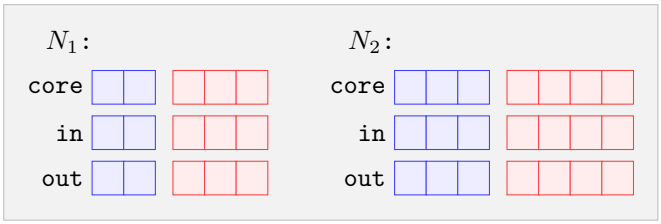
Nach dem Backtracking von (p_1, p_a)

$$M(s_0) = \emptyset$$

$$P_P(s_0) = \{\min(P_1 - M_1(s_0))\} \times (P_2 - M_2(s_0)) = \{(p_1, p_a), (p_1, p_b), (p_1, p_c)\}$$

$$P_T(s_0) = \{\min(T_1 - M_1(s_0))\} \times (T_2 - M_2(s_0)) = \{(t_1, t_a), (t_1, t_b), (t_1, t_c), (t_1, t_d)\}$$

$$\xrightarrow[\text{nichtdeterministisch}]{|P_P(s_0)| > 0 \wedge |P_T(s_0)| > 0} P(s_0) = P_P(s_0) = \{ \underbrace{\cancel{(p_1, p_a)}}_{\substack{\text{schon bearbeitet} \\ \text{(siehe Abbildung} \\ \text{4.1 und 4.5)}}} , (p_1, p_b), (p_1, p_c)\}$$

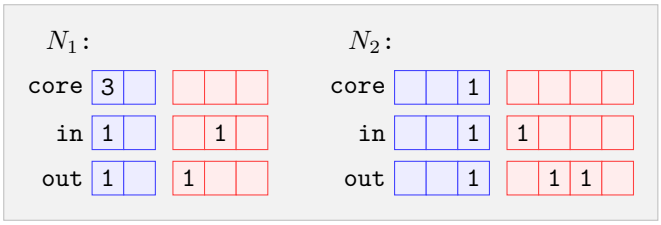


Zustand s_1

Abbildung 4.8.: PNVF2-Beispiel: Ausgangszustand s_0 - nach Backtracking von (p_1, p_a)

Nach dem Match von (p_1, p_c)

$$M(s_1) = \{(p_1, p_c)\}$$
$$P_P(s_1) = \{\min T_{P_1}^{out}(s_1)\} \times T_{P_2}^{out}(s_1) = \emptyset$$
$$P_T(s_1) = \{\min T_{T_1}^{out}(s_1)\} \times T_{T_2}^{out}(s_1) = \{(t_1, t_b), (t_1, t_c)\}$$
$$\xrightarrow[\text{~~nicht~~deterministisch}]{|P_P(s_1)| = 0 \wedge |P_T(s_1)| > 0} P(s_1) = P_T(s_1) = \{(t_1, t_b), (t_1, t_c)\}$$

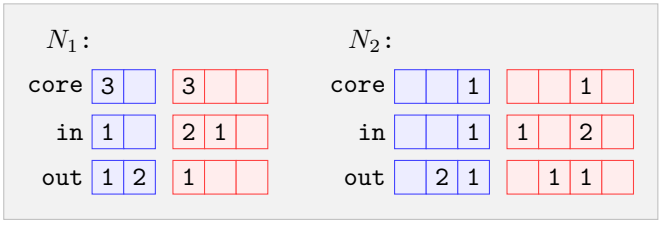


Zustand s_1

Abbildung 4.9.: PNVF2-Beispiel: Zustand s_1 - nach dem Match von p_1 auf p_c

Nach dem Match von (t_1, t_c)

$$M(s_2) = \{(p_1, p_c), (t_1, t_c)\}$$
$$P_P(s_2) = \{\min T_{P_1}^{out}(s_2)\} \times T_{P_2}^{out}(s_2) = \{(p_2, p_b)\}$$
$$P_T(s_2) = \{\min T_{T_1}^{out}(s_2)\} \times T_{T_2}^{out}(s_2) = \emptyset$$
$$\xrightarrow[\text{~~nicht~~deterministisch}]{|P_P(s_2)| > 0 \wedge |P_T(s_2)| = 0} P(s_2) = P_P(s_2) = \{(p_2, p_b)\}$$

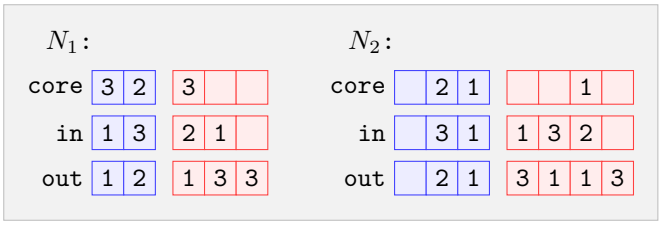


Zustand s_2

Abbildung 4.10.: PNVF2-Beispiel: Zustand s_2 - nach dem Match von t_1 auf t_c

Nach dem Match von (p_2, p_b)

$$M(s_3) = \{(p_1, p_c), (t_1, t_c), (p_2, p_b)\}$$
$$P_P(s_3) = \{\min T_{P_1}^{out}(s_3)\} \times T_{P_2}^{out}(s_3) = \emptyset$$
$$P_T(s_3) = \{\min T_{T_1}^{out}(s_3)\} \times T_{T_2}^{out}(s_3) = \{(t_2, t_a), (t_2, t_b), (t_2, t_d)\}$$
$$\xrightarrow[\text{~~nicht~~deterministisch}]{|P_P(s_3)| = 0 \wedge |P_T(s_3)| > 0} P(s_3) = P_T(s_3) = \{(t_2, t_a), (t_2, t_b), (t_2, t_d)\}$$

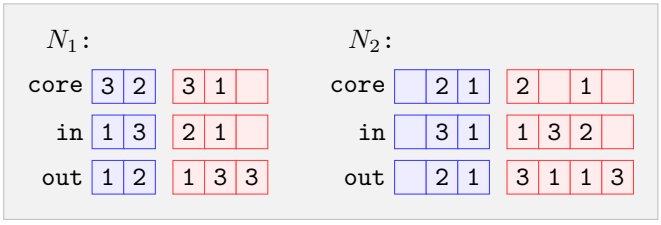


Zustand s_3

Abbildung 4.11.: PNVF2-Beispiel: Zustand s_3 - nach dem Match von p_2 auf p_b

Nach dem Match von (t_2, t_a)

$$M(s_4) = \{(p_1, p_c), (t_1, t_c), (p_2, p_b), (t_2, t_a)\}$$
$$P_P(s_4) = \{\min T_{P_1}^{out}(s_4)\} \times T_{P_2}^{out}(s_4) = \emptyset$$
$$P_T(s_4) = \{\min T_{T_1}^{out}(s_4)\} \times T_{T_2}^{out}(s_4) = \{(t_3, t_b), (t_3, t_d)\}$$
$$\xrightarrow[\text{~~nicht~~deterministisch}]{|P_P(s_4)| = 0 \wedge |P_T(s_4)| > 0} P(s_4) = P_T(s_4) = \{(t_3, t_b), (t_3, t_d)\}$$



Zustand s_4

Abbildung 4.12.: PNVF2-Beispiel: Zustand s_4 - nach dem Match von t_2 auf t_a

4.2.4. Der gefundene Match zwischen N_1 und N_2

$$T^{in}_{P_1}(s_5) = \emptyset \quad T^{in}_{T_1}(s_5) = \emptyset \quad T^{in}_{P_2}(s_5) = \emptyset \quad T^{in}_{T_2}(s_5) = \{t_b\}$$
$$T^{out}_{P_1}(s_5) = \emptyset \quad T^{out}_{T_1}(s_5) = \emptyset \quad T^{out}_{P_2}(s_5) = \emptyset \quad T^{out}_{T_2}(s_5) = \{t_b\}$$
$$\widetilde{P_1}(s_5) = \emptyset \quad \widetilde{T_1}(s_5) = \emptyset \quad \widetilde{P_2}(s_5) = \{p_a\} \quad \widetilde{T_2}(s_5) = \emptyset$$

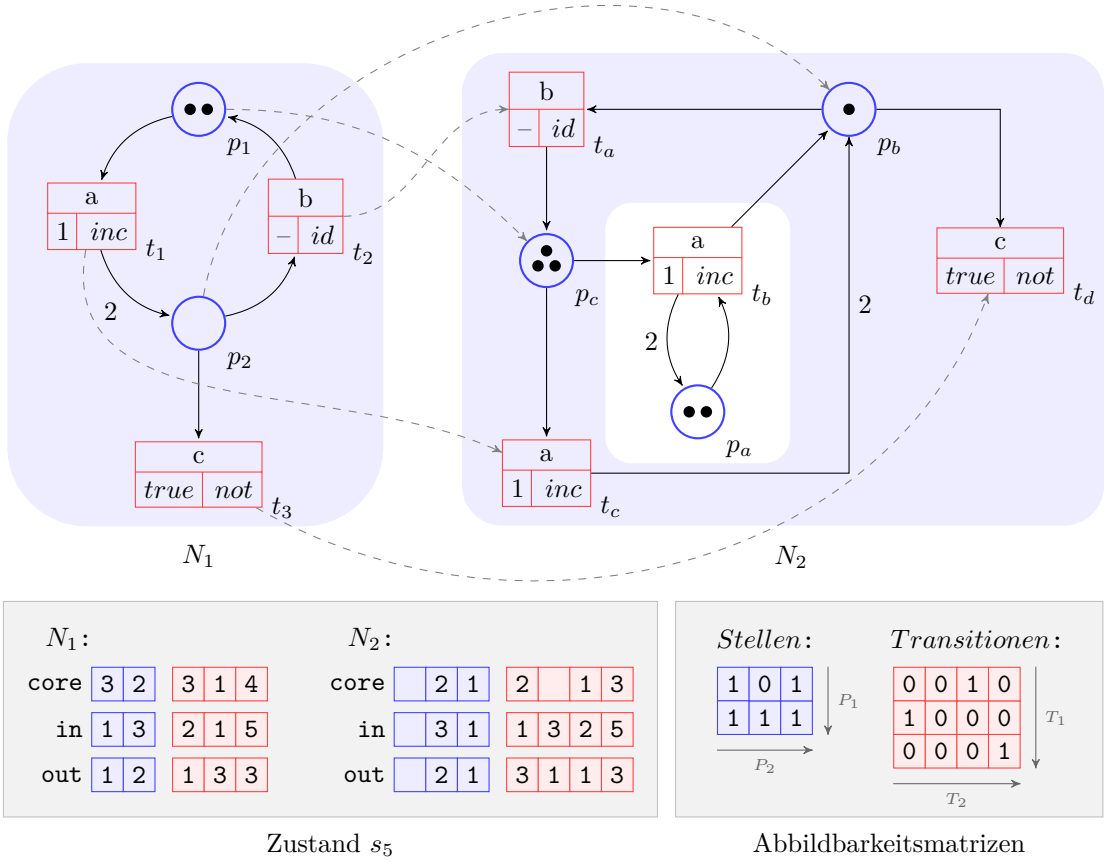


Abbildung 4.13.: PNVF2-Beispiel: der gefundene Match zwischen N_1 und N_2

Aufgrund der kleinen Netze und durch die Festlegung auf eine Kandidatenmenge erfolgen alle Schritte seit der ersten Ebene deterministisch. Da in diesem Beispiel der erste gefundene Match akzeptiert werden soll, ist der Matchingprozess erfolgreich beendet.

Der Match lautet: $M = \{(p_1, p_c), (p_2, p_b), (t_1, t_c), (t_2, t_a), (t_3, t_d)\}$ bzw.
$M : p_1 \mapsto p_c, p_2 \mapsto p_b, t_1 \mapsto t_c, t_2 \mapsto t_a, t_3 \mapsto t_d$

4.3. Korrektheit

Sowohl der VF2 als auch der PNVF2 garantieren die Korrektheit eines gefundenen Matches. Ein Match wird dann als korrekt angesehen, wenn die Struktur des Quellnetzes im Zielnetz erhalten bleibt und alle semantischen Eigenschaften kompatibel sind. Außerdem darf kein Zielknoten mehrere Quellknoten besitzen. Um diese Bedingungen zu erfüllen, prüfen die Algorithmen die Kandidaten, bevor sie sie zu dem jeweiligen Teilmatch hinzufügen. Die beim rekursiven Abstieg auftretende Kandidatenwahl sorgt für die Injektivität der Abbildung und $feasible$ für die strukturelle als auch semantische Kompatibilität. Da für den PNVF2 einige Veränderungen vorgenommen wurden, muss die behauptete Korrektheit bewiesen werden. Zum einfacheren Verständnis wird erst der VF2 untersucht und danach der PNVF2 bewiesen.

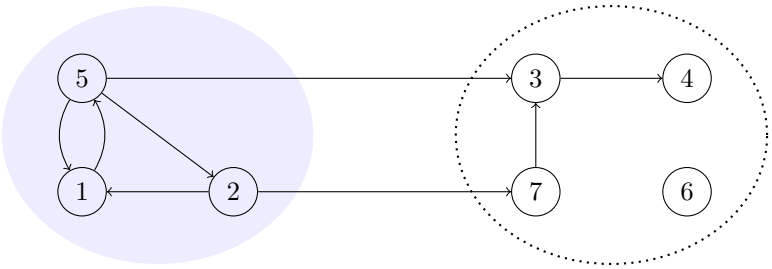
4.3.1. Beweis der Korrektheit des VF2

Damit die Korrektheit des VF2 (siehe Abschnitt 3.3) bewiesen werden kann, müssen zuerst die beteiligten Graphen etwas genauer definiert werden. In diesem Beweis wird angenommen, dass der Algorithmus eine Untergraphisomorphie von G_1 nach G_2 sucht. Die Betrachtung der Graphisomorphie wird ausgelassen, da sie einen Spezialfall der Untergraphisomorphie darstellt. Außerdem wird angenommen, dass die beteiligten Graphen weder Schleifen noch Mehrfachkanten enthalten. Aus diesen Annahmen können die benötigten Strukturen abgeleitet werden.

$$
\begin{array}{lll}
G_{i\in\{1,2\}} = (V_i, E_i) & G_{i,n} = (V_{i,n}, E_{i,n}) & V_{i,n} = M_i(s_n) \\
M(s_n) \subseteq V_1 \times V_2 & & E_{i,n} = E_i \cap (V_{i,n} \times V_{i,n}) \\
M_1(s_n) = \{v \mid (v,w) \in M(s_n)\} & \overline{G_{i,n}} = (\overline{V_{i,n}}, \overline{E_{i,n}}) & \overline{V_{i,n}} = V_i \setminus V_{i,n} \\
M_2(s_n) = \{w \mid (v,w) \in M(s_n)\} & & \overline{E_{i,n}} = E_i \cap (\overline{V_{i,n}} \times \overline{V_{i,n}})
\end{array}
$$

In der linken Spalte sind die bekannten Mengen aufgelistet. Ein Teilmatch wird durch $M(s_n)$ repräsentiert und die dazugehörigen Knoten werden in G_1 mit $M_1(s_n)$ und in G_2 mit $M_2(s_n)$ dargestellt. Die Relation $M(s_n)$ besitzt wieder die mengentheoretischen Eigenschaften einer Abbildung und wird im Folgenden auch teilweise in Form einer Funktion verwendet. Der Index n steht für die Anzahl der bisher abgebildeten Knoten. Da der Matchingprozess mit einem leeren Teilmatch beginnt, fällt n in das geschlossene

Intervall $[0,\ |V_1|\ ]$. In der rechten Spalte sind die Mengen für die beiden entstehenden Untergraphen aufgeführt. $G_{i,n}$ steht für den durch den Teilmatch $M(s_n)$ induzierten Untergraph in G_i, daher ist dessen Knotenmenge $V_{i,n}$ gleich $M_i(s_n)$. Der Graph $\overline{G_{i,n}}$ repräsentiert den induzierten Restgraphen und enthält die Knoten, die noch nicht abgebildet wurden (siehe Abbildung 4.14).



Teilmatch $M_1(s_3)$ und dessen Untergraph $G_{1,3}$

Untergraph $\overline{G_{1,3}}$ der nicht abgebildeten Knoten

Abbildung 4.14.: induzierte Untergraphen in G_1

Die Restgraphen werden zur Ermittlung der nächsten Kandidaten verwendet. Dieses Vorgehen weicht deutlich von der Kandidatenberechnung des VF2 ab, da dieser nur eine Teilmenge der insgesamt möglichen Kandidaten betrachtet (siehe Abschnitt 3.3.1). Allerdings soll mit diesem Beweis gezeigt werden, dass der inkrementelle Teilmatchaufbau in der Kombination mit $feasible$ zu einem korrekten Ergebnis führt. Das bedeutet anders ausgedrückt, dass ein vorhandener Teilmatch durch die Hinzunahme eines $feasible$ Paares immer einen neuen korrekten Teilmatch zur Folge hat. Die Strategie des VF2 dient bei der Suche einer Untergraphisomorphie alleinig der Suchraumbeschneidung und ist für die Korrektheit unwichtig. Daher wird im Folgenden die Menge aller möglichen Kandidaten betrachtet.

$$
\begin{aligned}
feasible(s_{n-1}, v, w) &\equiv rule_{pred}(s_{n-1}, v, w) \wedge rule_{succ}(s_{n-1}, v, w) \\
rule_{pred}(s_{n-1}, v, w) &\equiv \Big[\forall (v', v) \in E_{1,n} : \exists (w', w) \in E_{2,n} : (v', w') \in M(s_{n-1})\Big] \\
&\quad \wedge \Big[\forall (w', w) \in E_{2,n} : \exists (v', v) \in E_{1,n} : (v', w') \in M(s_{n-1})\Big] \\
rule_{succ}(s_{n-1}, v, w) &\equiv \Big[\forall (v, v') \in E_{1,n} : \exists (w, w') \in E_{2,n} : (v', w') \in M(s_{n-1})\Big] \\
&\quad \wedge \Big[\forall (w, w') \in E_{2,n} : \exists (v, v') \in E_{1,n} : (v', w') \in M(s_{n-1})\Big]
\end{aligned}
$$

Aus dem gleichen Grund bezieht sich $feasible$ nur auf die Vorgänger- und Nachfolgerregeln. Da der Zusammenhang zwischen einem gegebenen Teilmatch $M(s_{n-1})$, einem $feasible$ Paar (v, w) und dem entstehenden Teilmatch $M(s_n)$ untersucht wird, wurden die Indizes und die Definitionen der Regeln im Vergleich zum Abschnitt 3.3.1 leicht verändert. Um die Kompatibilität zweier Knoten bezüglich dem bisherigen Teilmatch $M(s_{n-1})$ zu bewerten, betrachten die Regeln die entstehenden Untergraphen $G_{i,n}$. Aufgrund der fehlenden Schleifen können die Vorgänger und Nachfolger der Kandidatenknoten nur in dem gegebenen Teilmatch $M(s_{n-1})$ liegen. Daher werden zwar die Kanten aus $E_{i,n}$ betrachtet, aber die Abbildungen in $M(s_{n-1})$ geprüft.

$$match(s_n) \equiv total(s_n) \wedge eindeutig(s_n) \wedge strukturerhaltend(s_n)$$

$$\begin{aligned} total(s_n) \wedge eindeutig(s_n) \equiv &\Big[\forall v \in V_{1,n} : \exists! w \in V_{2,n} : (v, w) \in M(s_n)\Big] \\ &\wedge \Big[\forall w \in V_{2,n} : \exists! v \in V_{1,n} : (v, w) \in M(s_n)\Big] \end{aligned}$$

$$\begin{aligned} strukturerhaltend(s_n) \equiv &\Big[\forall (v', v'') \in E_{1,n} : \exists (w', w'') \in E_{2,n} : (v', w'), (v'', w'') \in M(s_n)\Big] \\ &\wedge \Big[\forall (w', w'') \in E_{2,n} : \exists (v', v'') \in E_{1,n} : (v', w'), (v'', w'') \in M(s_n)\Big] \end{aligned}$$

Die Korrektheit eines Teilmatches wird durch das Prädikat $match$ getestet. Zum Bestehen muss $M(s_n)$ zum einen die Bedingungen an eine Abbildung erfüllen (Totalität und Eindeutigkeit auf den Untergraphen) und zum anderen die Strukturerhaltung sicherstellen. Zur Prüfung, ob $M(s_n)$ die Struktur der beiden Graphen erhält, wird die korrekte Abbildung der Start- und Endknoten der Kanten untersucht.

Theorem:

Für $n \in \mathbb{N}$ mit $n \leq |V_1|$, $feasible(s_{n-1}, v, w)$ mit $(v, w) \in \overline{V_{1,(n-1)}} \times \overline{V_{2,(n-1)}}$ und $M(s_n) = M(s_{n-1}) \cup \{(v, w)\}$ gilt:

$$match(s_{n-1}) \wedge feasible(s_{n-1}, v, w) \longrightarrow match(s_n)$$

Beweis durch Induktion

Induktionsanfang: $(n = 1)$

Da die Graphen $G_{1,0} = G_{2,0} = (\emptyset, \emptyset)$ weder Knoten noch Kanten enthalten und

$M(s_0) = \emptyset$ der leere Match ist, gilt $match(s_0)$. Gegeben sei ein Paar $(v, w) \in \overline{V_{1,0}} \times \overline{V_{2,0}}$, welches $feasible(s_0, v, w)$ erfüllt. So folgt aus $|V_{1,1}| = |V_{2,1}| = |M(s_1)| = 1$ und $|E_{1,1}| = |E_{2,1}| = 0$, dass $M(s_1) = \{(v, w)\}$ sowohl total, eindeutig als auch strukturerhaltend ist.

Induktionsschritt: $(n - 1 \longrightarrow n)$

Gegeben sei ein $feasible$ Paar $(v, w) \in \overline{V_{1,(n-1)}} \times \overline{V_{2,(n-1)}}$. Des Weiteren gilt $match(s_{n-1}) \equiv true$, da für $n - 1$ die Induktionsbehauptung gelte.

ZZ $match(s_n)$:
Aus der IB folgt, dass $M(s_{n-1})$ total und eindeutig ist. Da $M(s_n) = M(s_{n-1}) \cup \{(v, w)\}$ und weder $v \notin V_{1,(n-1)}$ noch $w \notin V_{2,(n-1)}$ ist sowie $V_{1,n} = V_{1,(n-1)} \cup \{v\}$ und $V_{2,n} = V_{2,(n-1)} \cup \{w\}$ gilt, ist $M(s_n)$ ebenso total und eindeutig.

Weil $M(s_{n-1})$ laut der IB schon strukturerhaltend ist und weder v noch w enthält, kann die Strukturerhaltung von $M(s_n)$ nur durch das Hinzukommen von Kanten zwischen v und Knoten in $V_{1,(n-1)}$ oder w und Knoten in $V_{2,(n-1)}$ verletzt werden.

1. Fall $G_{1,n}$:

Angenommen es existiere eine Kante (v', v'') in $E_{1,n}$, für die keine entsprechende Kante $(M(s_n)(v'), M(s_n)(v''))$ in $E_{2,n}$ existiert. Aus der IB folgt, dass entweder $v = v'$ oder $v = v''$ ist. Eine in $G_{2,n}$ fehlende oder falsch dorthin abgebildete Kante widerspricht allerdings der Forderung von $feasible$. Also kann es keine solche Kante (v', v'') geben.

2. Fall $G_{2,n}$:

Die Argumentation erfolgt analog für w.

Aufgrund von $feasible$ kann es keine strukturbrechende Kante in $G_{1,n}$ und $G_{2,n}$ geben. Daher sind die beiden Graphen isomorph und $match(s_n)$ gilt.

Korrektheit der Untergraphisomorphie zwischen G_1 und G_2

Da die Induktion über $\mathbb{N}$ mit $n = |V_1|$ endet und $G_{1,n} = G_{1,|V|} = G$ gilt, folgt aus dem vorherigen Beweis, dass $M(s_{|V|})$ ein gültiger Match zwischen G_1 und einem Untergraphen von G_2 ist.

4.3.2. Beweis der Korrektheit des PNVF2

Nachdem das Beweisschema verdeutlicht wurde, soll auf die Petrinetze eingegangen werden. Der PNVF2 sucht einen Match zwischen den beiden dekorierten Petrinetzen N_1 und N_2. Der Spezialfall der Graphisomorphie wird (wie schon beim VF2) ausgelassen.

$$\begin{aligned} N_{i\in\{1,2\}} = (P_i, T_i, pre_i, post_i, \quad & pre_i : P_i \times T_i \to \mathbb{N}_0 \quad & card_{pre_i} : T_i \to \mathbb{N}_0 \\ m_i, cap_i, pname_i, \quad & post_i : P_i \times T_i \to \mathbb{N}_0 \quad & card_{post_i} : T_i \to \mathbb{N}_0 \\ tname_i, tlb_i, rnw_i) & & \end{aligned}$$

Die beiden Funktionen pre_i und $post_i$ beschreiben die Vor- und Nachbereichskanten aus der Sicht der Transition. Eine Kante besteht genau dann, wenn der Funktionswert größer ist als 0. Dieser bildet gleichzeitig das Gewicht der Kante. Die Anzahl der Nachbarn einer Transition wird durch die beiden Funktionen $card_{pre_i}$ und $card_{post_i}$ angegeben. Zum Beweis der Korrektheit müssen noch die einzelnen Unternetze definiert werden.

$$\begin{aligned} Q_{i,n} = (P_{i,n}, T_{i,n}, pre_{i,n}, post_{i,n}) \quad & P_{i,n} = P_i \cap M_i(s_n) \quad & pre_{i,n} : P_{i,n} \times T_{i,n} \to \mathbb{N}_0, \\ M(s_n) \subseteq (P_1 \times P_2) \cup (T_1 \times T_2) \quad & T_{i,n} = T_i \cap M_i(s_n) \quad & (p,t) \mapsto pre_i(p,t) \\ M_1(s_n) = \{v \mid (v,w) \in M(s_n)\} \quad & \overline{P_{i,n}} = P_i \setminus P_{i,n} \quad & post_{i,n} : P_{i,n} \times T_{i,n} \to \mathbb{N}_0, \\ M_2(s_n) = \{w \mid (v,w) \in M(s_n)\} \quad & \overline{T_{i,n}} = T_i \setminus T_{i,n} \quad & (p,t) \mapsto post_i(p,t) \end{aligned}$$

Um den Beweis übersichtlicher zu gestalten, werden nur vereinfachte Unternetze betrachtet. Da diese zur Prüfung der Strukturerhaltung aufgestellt werden, ist der Verzicht auf die semantischen Eigenschaften unproblematisch. Um sie zu prüfen, wird an den jeweiligen Stellen auf das Ausgangsnetz zurückgegriffen. Die Bedingungen an den Teilmatch $M(s_n)$ und seine beiden Knotenmengen $M_1(s_n)$ und $M_2(s_n)$ gelten entsprechend zum Beweis des VF2 (siehe Unterabschnitt 4.3.1). Die Restknotenmengen der Stellen $\overline{P_{i,n}}$ und Transitionen $\overline{T_{i,n}}$ werden in diesem Beweis ohne das induzierte Unternetz aufgeführt.

$$feasible(s_{n-1}, v, w) \equiv \begin{cases} feasible_P(s_{n-1}, v, w) & \text{falls } (v,w) \in P_1 \times P_2 \\ feasible_T(s_{n-1}, v, w) & \text{falls } (v,w) \in T_1 \times T_2 \end{cases}$$

$$\begin{aligned} feasible_P(s_{n-1}, p_1, p_2) &\equiv rule_{sem,P} \wedge rule_{pred,P} \wedge rule_{succ,P} \\ feasible_T(s_{n-1}, t_1, t_2) &\equiv rule_{sem,T} \wedge rule_{pred,T} \wedge rule_{succ,T} \end{aligned}$$

Der PNVF2 verwendet für die verschiedenen Knotenklassen verschiedene Regeln, deshalb wird zwischen einem $feasible$ für die Stellen $feasible_P$ und Transitionen $feasible_T$ unterschieden. In der Beschreibung des Algorithmus besteht jedes $feasible$ aus sechs Regeln (siehe Unterabschnitt 4.1.2). Die ersten drei dienen Sicherstellung der semantischen Abbildbarkeit und der Strukturerhaltung. Die letzten drei Regeln sind nur für die Graphisomorphie und die Suchraumbeschneidung relevant. Daher werden in diesem Beweis nur die jeweils ersten drei Regeln betrachtet.

$$
\begin{aligned}
rule_{sem,P} \equiv\ & m_1(p_1) \leq m_2(p_2) \\
& \wedge\, cap_1(p_1) = cap_2(p_2) \\
& \wedge\, pname_1(p_1) = pname_2(p_2)
\end{aligned}
\qquad
\begin{aligned}
rule_{sem,T} \equiv\ & card_{pre_1}(t_1) = card_{pre_2}(t_2) \\
& \wedge\, card_{post_1}(t_1) = card_{post_2}(t_2) \\
& \wedge\, tname_1(t_1) = tname_2(t_2) \\
& \wedge\, tlb_1(t_1) = tlb_2(t_2) \\
& \wedge\, rnw_1(t_1) = rnw_2(t_2)
\end{aligned}
$$

Die jeweils erste Regel von $feasible_P$ und $feasible_T$ stellen $rule_{sem,P}$ und $rule_{sem,T}$ dar. Diese sorgen dafür, dass nur die Stellen und Transitionen aufeinander abgebildet werden, die semantisch kompatibel sind. Die Definitionen der Regeln entsprechen genau denen, die in der Beschreibung des PNVF2 aufgeführt werden. Da die semantischen Eigenschaften nicht von der Struktur abhängig sind, müssen die Regeln auch nicht auf diese Untergraphbetrachtung angepasst werden. In der Vorstellung des PNVF2 wurde auf die Veränderlichkeit von $rule_{sem,P}$ für die verschiedenen Matchingkontexte hingewiesen (siehe Unterabschnitt 4.1.2). Für diesen Beweis wird von dem allgemeinen Matching ausgegangen. Da bei einer Anpassung von $rule_{sem,P}$ auch $match$ (auf die gleiche Weise) angepasst werden muss, ist das Ergebnis dieses Beweises auf die anderen Fälle übertragbar.

$$
\begin{aligned}
rule_{pred,P}(s_{n-1}, p_1, p_2) &\equiv \forall t_1 \in T_{1,(n-1)} : post_{1,n}(p_1, t_1) = post_{2,n}(p_2, M(s_n)(t_1)) \\
rule_{pred,T}(s_{n-1}, t_1, t_2) &\equiv \forall p_1 \in P_{1,(n-1)} : pre_{1,n}(p_1, t_1) = pre_{2,n}(M(s_n)(p_1), t_2)
\end{aligned}
$$

$$
\begin{aligned}
rule_{succ,P}(s_{n-1}, p_1, p_2) &\equiv \forall t_1 \in T_{1,(n-1)} : pre_{1,n}(p_1, t_1) = pre_{2,n}(p_2, M(s_n)(t_1)) \\
rule_{succ,T}(s_{n-1}, t_1, t_2) &\equiv \forall p_1 \in P_{1,(n-1)} : post_{1,n}(p_1, t_1) = post_{2,n}(M(s_n)(p_1), t_2)
\end{aligned}
$$

Die letzten beiden Regeln $rule_{pred}$ und $rule_{succ}$ stellen die Erhaltung der Struktur sicher, indem sie die Vorgänger und Nachfolger der Stellen oder Transitionen prüfen. Dazu

untersuchen Sie alle Kanten zwischen Knoten aus dem bisherigen Teilmatch $M(s_{n-1})$ und dem zu prüfenden Kandidatenpaar. Die Gewichte der bestehenden Kanten werden durch die Art der Kantenrepräsentation automatisch mit getestet.

$$match(s_n) \equiv total(s_n) \wedge eindeutig(s_n) \wedge strukturerhaltend(s_n) \wedge semantischOK(s_n)$$

Das Prädikat zur Prüfung der Korrektheit eines Matches *match* besitzt im Vergleich zum vorherigen Beweis eine zusätzliche Bedingung. Eine gegebene Relation $M(s_n)$ muss neben den mengentheoretischen Abbildungseigenschaften und der Strukturerhaltung auch noch semantisch korrekt sein.

$$\begin{aligned} total(s_n) \wedge eindeutig(s_n) \equiv\ & \forall p_1 \in P_{1,n} : \exists! p_2 \in P_{2,n} : (p_1, p_2) \in M(s_n) \\ & \wedge \forall t_1 \in T_{1,n} : \exists! t_2 \in T_{2,n} : (t_1, t_2) \in M(s_n) \\ & \wedge \forall p_2 \in P_{2,n} : \exists! p_1 \in P_{1,n} : (p_1, p_2) \in M(s_n) \\ & \wedge \forall t_2 \in T_{2,n} : \exists! t_1 \in T_{1,n} : (t_1, t_2) \in M(s_n) \end{aligned}$$

$$\begin{aligned} strukturerhaltend(s_n) \equiv\ & \forall p_1 \in P_{1,n} : \forall t_1 \in T_{1,n} : \Big[\\ & pre_{1,n}(p_1, t_1) = pre_{2,n}(M(s_n)(p_1), M(s_n)(t_1)) \\ & \wedge post_{1,n}(p_1, t_1) = post_{2,n}(M(s_n)(p_1), M(s_n)(t_1)) \Big] \end{aligned}$$

$$\begin{aligned} semantischOK(s_n) \equiv\ & \forall (p_1, p_2) \in M(s_n) \cap (P_1 \times P_2) : \Big[\\ & m_1(p_1) \leq m_2(p_2) \\ & \wedge cap_1(p_1) = cap_2(p_2) \\ & \wedge pname_1(p_1) = pname_2(p_2) \Big] \\ & \wedge \forall (t_1, t_2) \in M(s_n) \cap T_1 \times T_2) : \Big[\\ & card_{pre_1}(t_1) = card_{pre_2}(t_2) \\ & \wedge card_{post_1}(t_1) = card_{post_2}(t_2) \\ & \wedge tname_1(t_1) = tname_2(t_2) \\ & \wedge tlb_1(t_1) = tlb_2(t_2) \\ & \wedge rnw_1(t_1) = rnw_2(t_2) \Big] \end{aligned}$$

Theorem:

Für $n \in \mathbb{N}$ mit $n \leq |P_1| + |T_1|$, $feasible(s_{n-1}, v, w)$ mit $(v, w) \in (\overline{P_{1,(n-1)}} \times \overline{P_{2,(n-1)}}) \cup (\overline{T_{1,(n-1)}} \times \overline{T_{2,(n-1)}})$ und $M(s_n) = M(s_{n-1}) \cup \{(v, w)\}$ gilt:

$$match(s_{n-1}) \wedge feasible(s_{n-1}, v, w) \longrightarrow match(s_n)$$

Beweis durch Induktion

Induktionsanfang: $(n = 1)$

Da die Petrinetze $Q_{1,0}$ und $Q_{2,0}$ weder Stellen, Transition noch Kanten enthalten, ist $M(s_0) = \emptyset$ ein gültiger Match. Gegeben sei ein $feasible$ Paar $(v, w) \in (\overline{P_{1,0}} \times \overline{P_{2,0}}) \cup (\overline{T_{1,0}} \times \overline{T_{2,0}})$.

ZZ $match(s_1)$:

Für $(v, w) \in P_1 \times P_2$ **:**

Aus $P_{1,1} = \{v\}$, $P_{2,1} = \{w\}$, $T_{1,1} = T_{2,1} = \emptyset$ und $M(s_1) = \{(v, w)\}$ folgt, dass $M(s_1)$ sowohl total als auch eindeutig ist. Da $Q_{i,1}$ bipartite transitionslose Graphen sind, enthalten sie auch keine Kanten. Also ist $M(s_1)$ ebenso strukturerhaltend.

Weil $M(s_1)$ nur das Paar (v, w) enthält und die semantischen Forderungen von $match$ gerade denen von $feasible_P$ entsprechen, ist $M(s_1)$ ebenso eine semantisch korrekte Abbildung.

Für $(v, w) \in T_1 \times T_2$ **:**

Die Korrektheit wird analog belegt.

Induktionsschritt: $(n - 1 \longrightarrow n)$

Gegeben sei ein $feasible$ Paar $(v, w) \in (\overline{P_{1,(n-1)}} \times \overline{P_{2,(n-1)}}) \cup (\overline{T_{1,(n-1)}} \times \overline{T_{2,(n-1)}})$. Des Weiteren gilt $match(s_{n-1}) \equiv true$, da für $n - 1$ die Induktionsbehauptung gelte.

ZZ $match(s_n)$:

Für $(v, w) \in P_1 \times P_2$ **:**

Totalität und Eindeutigkeit: Aus der IB folgt, dass $M(s_{n-1})$ linkstotal und rechtseindeutig ist. Da $M(s_n) = M(s_{n-1}) \cup \{(v, w)\}$, $v \notin P_{1,(n-1)}$ und $P_{1,n} = P_{1,(n-1)} \cup \{v\}$ sind, ist $M(s_n)$ ebenso linkstotal und rechtseindeutig. Die Rechtstotalität und Linkseindeutigkeit von $M(s_n)$ ergibt sich entsprechend.

Strukturerhaltung: $M(s_{n-1})$ ist laut der IB schon strukturerhaltend und enthält weder v noch w. Dadurch kann die Strukturerhaltung von $M(s_n)$ nur durch das Hinzukommen von Kanten zwischen v und Transitionen in $T_{1,(n-1)}$ bzw. w und Transitionen in $T_{2,(n-1)}$ verletzt werden.

1. Fall: $t_1 \in T_{1,(n-1)}$
Angenommen, es existiere eine Kante:

$$pre_{1,n}(v', t_1) \neq pre_{2,n}(M(s_n)(v'), M(s_n)(t_1)) \quad \text{bzw.}$$
$$post_{1,n}(v', t_1) \neq post_{2,n}(M(s_n)(v'), M(s_n)(t_1))$$

Aus der IB folgt, dass $v' = v$ und $M(s_n)(v') = w$ ist, weil alle anderen Stellen schon strukturerhaltend abgebildet wurden. Eine in $Q_{2,n}$ fehlende oder falsch dorthin abgebildete Kante widerspricht allerdings der Forderung von $feasible_P$. Also kann es keine solche Kante geben.

2. Fall: $t_2 \in T_{2,(n-1)}$
Die Argumentation erfolgt analog für w.

Semantische Korrektheit: Weil $M(s_{n-1})$ laut IB schon semantisch korrekt ist, kann diese für $M(s_n)$ nur durch das Paar (v, w) verletzt werden. Da die semantischen Forderungen von *match* gerade denen von $feasible_P$ entsprechen, ist $M(s_n)$ ebenso semantisch korrekt.

Für $(v, w) \in T_1 \times T_2$ **:**
Die Korrektheit wird analog belegt.

Korrektheit der die Untergraphisomorphie zwischen N_1 und N_2

Die Induktion über $\mathbb{N}$ endet mit $n = |P_1| + |T_1|$, wobei $Q_{1,n}$ alle Stellen, Transitionen und Kanten von N_1 abdeckt. Für einen gültigen Match müssen der Vor- und Nachbereich der Transitionen exakt erhalten bleiben. *match* fordert, dass die Anzahl der Kanten der Transitionspaare in N_i identisch sind. Aus der Eindeutigkeit, der Strukturkerhaltung und der gleichen Anzahl von Nachbarn folgt, dass die Nachbarn sämtlicher gematchter Transitionen von N_2 in $M(s_n)$ enthalten sind. Also ist $M(s_n)$ ein gültiger Match zwischen N_1 und N_2.

4.4. Laufzeitkomplexität

Zur korrekten Lösung des Matchingproblems stehen eine Vielzahl von Algorithmen zur Verfügung. Wenn für eine Applikation die Wahl getroffen werden muss, welcher Vertreter verwendet werden soll, spielen neben der Korrektheit noch weitere Eigenschaften eine Rolle. Eine der bedeutendsten Eigenschaften ist die Komplexität. Sie beschreibt in abstrakter Form, welchen Aufwand der Algorithmus zur Lösung des Problems betreiben muss. Dazu trifft sie eine Aussage darüber, wie der Ressourcenverbrauch mit steigender Problemgröße skaliert. Der Begriff der Problemgröße ist dabei von zentraler Bedeutung. Die Aufwände verschiedener Algorithmen lassen sich nur vergleichen, wenn ihre Definitionen der Problemgröße kompatibel sind. Des Weiteren kann die falsche Wahl der Problemgröße die gesamte Analyse unbrauchbar machen [vgl. Wag03, S. 15 - 16]. Im Falle des Matchingproblems ist die Größe meist die Anzahl der Knoten der beteiligten Graphen. Aber auch das Matching von verschiedenen Graphen der gleichen Größe ist für sich gesehen nicht immer gleich aufwändig. Als Beispiel diene die Suche eines Vorkommens in zwei kantenlosen oder fast vollständigen Graphen [vgl. Cor+99, S. 4]. Daraus lassen sich verschiedene Komplexitätsbetrachtungen ableiten. Im Folgenden wird zwischen dem Best-Case und Worst-Case unterschieden [vgl. Cor+10, S. 27 ff.].

Damit Aussagen über die Komplexitäten der Algorithmen getroffen werden können, müssen diese analysiert werden. Bei der Betrachtung ihrer Beschreibungen fällt auf, dass man zwischen zwei Arten von Operationen differenzieren kann. Zum einen nutzen die Algorithmen die Operationen der zugrundeliegenden Modelle und zum anderen führen sie eigene komplexe Operationen ein. Jede Grundoperation besitzt im jeweiligen Modell einen bestimmten Aufwand [vgl. Wag03, S. 15 - 16]. In der folgenden Analyse wird davon ausgegangen, dass die Ausführung einer Grundoperation in $\Theta(1)$ erfolgt. Dies bezieht sich unter anderem auf das Lesen und Setzen von Einträgen in der State-Space-Representation (SSR) [vgl. Cor+01a, S. 4 - 5], den Matrizen sowie auf die Zugriffe auf die Datenstrukturen der beteiligten Graphen. Es wird also angenommen, dass die Graphen in einer Form vorliegen, bei der in konstanter Zeit auf die einzelnen Knoten und die Menge ihrer Nachbarn zugegriffen werden kann. Außerdem wird für den PNVF2 vorausgesetzt, dass die Vergleiche der Namen, Labels und anderer Eigenschaften in konstanter Zeit stattfinden.

Die komplexen Operationen werden in Abhängigkeit zur Problemgröße auf die Grundoperationen zurückgeführt. Anders ausgedrückt, bestimmt die Problemgröße die Häufigkeit, mit der eine Grundoperation ausgeführt wird. Deshalb stellt die Summe der

Aufwände sämtlicher durchzuführender Grundoperationen den Aufwand des Algorithmus dar [vgl. Wag03, S. 15 - 16].

4.4.1. Rekursionsgleichung

Zur Abschätzung und zum Beweis der Komplexität wird zuerst eine Rekursionsgleichung benötigt. Diese hat die Form einer rekursiv definierten Funktion. Dabei bildet sie die Problemgrößen auf die entsprechende Laufzeit ab [vgl. Cor+10, S. 67 ff.].

Rekursionsgleichung für den VF2

Im Falle des Matchings mit dem VF2 sind die Problemgrößen die Werte n und m, welche für die jeweils abzubildende Knotenanzahl von G_1 und G_2 stehen. Natürlich kann nur ein Match gefunden werden, wenn m wenigstens genauso groß ist wie n. Im Folgenden wird also angenommen, dass $n \leq m$ ist.

Zur Ermittlung der Rekursionsgleichung müssen den einzelnen Teiloperationen Aufwände zugeordnet werden. Der VF2 versucht auf jeder Ebene einen Quellknoten in eine Menge möglicher Zielknoten abzubilden. Bei Erfolg wird ein neuer Teilmatch berechnet, der auf der nächsten Rekursionsebene weiter untersucht wird.

```
1  PROCEDURE Match(s)
2     INPUT:   an intermediate state s; the initial state s_0 has M(s_0) = ∅
3     OUTPUT:  the mappings between the two graphs
4
5     IF M(s) covers all the nodes of G_1 THEN
6        OUTPUT M(s)
7     ELSE
8        Compute the set P(s) of the pairs candidate for inclusion in M(s)
9        FOREACH p in P(s)
10          IF the feasibility rules succeed for the inclusion of p in M(s) THEN
11             Compute the state s' obtained by adding p to M(s)
12             CALL Match(s')
13          END IF
14       END FOREACH
15       Restore data structures
16    END IF
17 END PROCEDURE Match
```

Quelle: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs [Cor+04, S. 1368][1]

Abbildung 4.15.: VF2-Algorithmus

[1] In [Cor+04] wird ein Match von G_2 nach G_1 gesucht. Da hier die Betrachtung in umgekehrter Reihenfolge erfolgt, wurde der Pseudocode so angepasst, dass als Quellgraph G_1 verwendet wird.

Auf jeder Ebene müssen die folgenden wiederkehrenden Aktionen durchgeführt werden:

1. Der Algorithmus muss entscheiden, welche Menge (Terminalmenge oder Restknotenmenge) er als Basis für die Kandidatengenerierung verwendet (Zeile 8).
2. Der kleinste noch nicht abgebildete Quellknoten muss gefunden werden (Zeile 8).
3. Die noch nicht abgebildeten Zielknoten müssen identifiziert werden (Zeile 8).
4. Die Kombination aus dem Quellknoten und einem Zielknoten bildet ein Kandidatenpaar. Für jedes dieser Paare sind die folgenden Schritte zu unternehmen (Zeile 9):
   a) Aus der Sicht des Quellknotens muss geprüft werden, ob alle Vorgänger und Nachfolger korrekt abgebildet wurden. Außerdem müssen die Werte für die Lookahead-Prüfung generiert werden (Zeile 10).
   b) Das Gleiche muss noch einmal für den Zielknoten getan werden (Zeile 10).
   c) Nachdem die Lookahead-Werte generiert wurden, muss geprüft werden, ob der Zielknoten ausreichend Nachbarn besitzt (Zeile 10).
   d) Das gültige Paar wird zum aktuellen Zustand hinzugefügt (Zeile 11).
   e) Danach vollzieht der Algorithmus einen Rekursionsschritt, um die übrigen Knoten abzubilden (Zeile 12).
   f) Falls das Matching fehlschlägt, müssen die vorgenommen Änderungen rückgängig gemacht werden (Zeile 15).

Die benötigte Rekursionsgleichung kann aus den aufgelisteten Schritten hergeleitet werden. Um dieses übersichtlicher zu gestalten, werden die einzelnen Teilschritte als Nebenfunktionen T_i aufgeführt:

$$T_1(n,m) = c_1 n \quad \text{(Finden des kleinsten freien Quellknotens)}$$
$$T_2(n,m) = c_2 m \quad \text{(Finden der möglichen Zielknoten)}$$
$$T_3(n,m) = c_3(n-1) \quad \text{(Prüfung des Quellknoten sowie Lookahead)}$$
$$T_4(n,m) = c_4(m-1) \quad \text{(Prüfung des Zielknoten sowie Lookahead)}$$
$$T_5(n,m) = c_5 + c_5' n + c_5'' m \quad \text{(Hinzufügen des Paares zum aktuellen Zustand)}$$
$$T_6(n,m) = c_6 + c_6' n + c_6'' m \quad \text{(Wiederherstellen des letzten Zustandes)}$$

Der Algorithmus verwendet als Datenstruktur die SSR. Um den kleinsten freien Quellknoten zu finden sowie die Zielknoten zu identifizieren, muss der VF2 über beide core-Arrays iterieren. Daraus ergeben sich die Laufzeiten $\mathcal{O}(n)$ und $\mathcal{O}(m)$ für T_1 und T_2. Da ein Knoten im Maximalfall Kanten zu jedem seiner Nachbarn besitzt, beziehen sich die Funktionen für die *feasible*-Prüfung T_3 und T_4 auf $(n-1)$ und $(m-1)$.

$$T_{i\in\{5,6\}}(n,m) = \underbrace{c_i}_{\text{core-Arrays}} + \underbrace{c_i' n}_{\text{in- und out-Arrays in } G_1} + \underbrace{c_i'' m}_{\text{in- und out-Arrays in } G_2}$$

Die Aktualisierung der SSR wird in T_5 und T_6 behandelt. Wenn ein Paar zum aktuellen Zustand hinzugefügt oder aus diesem entfernt wird, müssen die Terminalmengen und der bisherige Teilmatch aktualisiert werden. Die Veränderungen der core-Arrays betreffen nur das Feld des jeweiligen Kandidatenknotens und sind somit konstant. Bei der Aktualisierung der Terminalmengen müssen im schlimmsten Fall alle Felder der in- und out-Arrays modifiziert werden [vgl. Cor+04, S. 1369]. Dies tritt beispielsweise ein, wenn bei einem vollständigen Graphen das erste Paar betrachtet wird. Die Anzahl an Operationen für jeden Schritt hängen zwar von der Problemgröße ab, sind aber konstant. Diese werden durch die positiven (konstanten) Faktoren c repräsentiert. Mit Hilfe dieser Nebenfunktionen kann die Rekursionsgleichung des VF2 aufgestellt werden:

$$\begin{aligned}
T(0,m) &= c \\
T(n,m) &= c + \sum_{i=1}^{2} T_i(n,m) + m \cdot \left[\sum_{i=3}^{6} T_i(n,m) \right] + m \cdot T(n-1,m-1) \\
&= c + c_1 n + c_2 m + m \cdot \Big[c_3(n-1) + c_4(m-1) + c_5 + c_5' n + c_5'' m \\
&\qquad + c_6 + c_6' n + c_6'' m \Big] + m \cdot T(n-1,m-1)
\end{aligned}$$

Wie zu erkennen ist, besteht die Gleichung aus dem Rekursionsanker $T(0,m)$ sowie der rekursiv definierten Funktion $T(n,m)$. Für $T(n,m)$ gilt, dass n immer größer ist als null. Die beiden Summen repräsentieren die Nebenfunktionen T_i. Der Faktor m vor der zweiten Summe und dem Rekursionschritt stellt die Anzahl der möglichen Zielknoten dar. Diese kann im Maximalfall m betragen. Der konstante Summand c steht für die Wahl der Ausgangsmengen (siehe Schritt 1) sowie einigen anderen konstanten Operationen.

Rekursionsgleichung für den PNVF2

Die Rekursionsgleichung für den angepassten Algorithmus ist etwas komplexer. Während der VF2 einen Match zwischen zwei allgemeinen Graphen sucht, betrachtet der PNVF2 Petrinetze. Petrinetze sind bipartite Graphen. Also müssen zwei verschiedene Arten von Knoten abgebildet werden. Daraus ergeben sich die vier Problemgrößen:

p_1	als Anzahl der zu matchenden Stellen des Netzes N_1
t_1	als Anzahl der zu matchenden Transitionen des Netzes N_1
p_2	als Anzahl der zu matchenden Stellen des Netzes N_2
t_2	als Anzahl der zu matchenden Transitionen des Netzes N_2

Im Folgendem wird angenommen, dass $p_1 \leq p_2$ und $t_1 \leq t_2$ ist, da ansonsten kein Match möglich wäre. Um bei jedem Durchlauf das Finden eines anderen Matches zu ermöglichen, werden die Ordnungen über die Stellen und Transitionen zufällig erstellt. Dazu werden die Knotenmengen zu Beginn permutiert:

$$T_{shuffle}(p_1, t_1, p_2, t_2) = cp_1 + cp_2 + ct_1 + ct_2$$

Für den Aufwand von $T_{shuffle}$ wird angenommen, dass ein geeignetes Permutationsverfahren verwendet wird, welches für jede Knotenmenge in $\mathcal{O}(n)$ arbeitet. Ein solches In-Place-Verfahren wird beispielsweise in [Cor+10, S. 127 - 128] beschrieben. Ein weiterer Unterschied zum VF2 ist, dass vor dem eigentlichen Matching die Abbildbarkeitsmatrizen für die Stellen und Transitionen generiert werden.

$$T_{matrizen}(p_1, t_1, p_2, t_2) = cp_1p_2 + dt_1t_2 + e$$

Die Vorberechnung der semantischen Abbildbarkeit erspart im späteren Matching die wiederholte semantische Prüfung derselben Kandidatenpaare. Außerdem besteht an dieser Stelle die Möglichkeit zu prüfen, ob für jede Stelle und jede Transition des Quellnetzes passende Knoten im Zielnetz existieren. Falls für einen Quellknoten kein passender Zielknoten existieren sollte, kann der Algorithmus den Matchingversuch ganz unterlassen.

Sollten für alle Knoten genügend unterstützende Zielknoten vorhanden sein, geht der Algorithmus in den Matchingteil über. Die Schritte entsprechen dabei weitestgehend denen des VF2. Allerdings vervielfachen sich die Formeln, da die verschiedenen Knoten-

klassen entsprechend gewürdigt werden müssen. Den Anfang bilden die Gleichungen für den Rekursionsanker und Netze, welche nur aus Knoten einer Klasse bestehen:

$$
\begin{aligned}
T_{match}(0,0,p_2,t_2) &= e \\
T_{match}(p_1,0,p_2,t_2) &= c + c_1p_1 + c_2p_2 + c_5p_2 + c_6p_2 + p_2 \cdot T_{match}(p_1-1,0,p_2-1,t_2) \\
T_{match}(0,t_1,p_2,t_2) &= d + d_1t_1 + d_2t_2 + d_5t_2 + d_6t_2 + t_2 \cdot T_{match}(0,t_1-1,p_2,t_2-1)
\end{aligned}
$$

Der allgemeine Fall des Matchings, bei dem sowohl Stellen als auch Transitionen präsent sind, teilt sich durch den Nichtdeterminismus in zwei Fälle auf:

$$
T_{match}(p_1,t_1,p_2,t_2) = \begin{cases} T_{match,P}(p_1,t_1,p_2,t_2) & \text{wenn Stellen gematcht werden} \\ T_{match,T}(p_1,t_1,p_2,t_2) & \text{wenn Transitionen gematcht werden} \end{cases}
$$

$$
\begin{aligned}
T_{match,P}(p_1,t_1,p_2,t_2) = c + c_1p_1 + c_2p_2 + p_2 \cdot \Big[& c_3(t_1-1) + c_4(t_2-1) + c_5 + c_5't_1 \\
& + c_5''t_2 + c_6 + c_6't_1 + c_6''t_2 \Big] + p_2 \cdot T_{match}(p_1-1,t_1,p_2-1,t_2)
\end{aligned}
$$

$$
\begin{aligned}
T_{match,T}(p_1,t_1,p_2,t_2) = d + d_1t_1 + d_2t_2 + t_2 \cdot \Big[& d_3(p_1-1) + d_4(p_2-1) + d_5 + d_5'p_1 \\
& + d_5''p_2 + d_6 + d_6'p_1 + d_6''p_2 \Big] + t_2 \cdot T_{match}(p_1,t_1-1,p_2,t_2-1)
\end{aligned}
$$

Der Algorithmus entscheidet sich auf jeder Ebene nichtdeterministisch für einen der beiden Fälle. Die entstehenden Kosten für diese Entscheidung sind in den konstanten Summanden c und d enthalten. Durch die Ersetzung der wiederholten semantischen Prüfung eines Kandidatenpaares mit einem Zugriff auf die Abbildbarkeitsmatrix entsprechen die Gleichungen optisch einem zweigeteiltem VF2.

$$
T(p_1,t_1,p_2,t_2) = T_{shuffle}(p_1,t_1,p_2,t_2) + T_{matrizen}(p_1,t_1,p_2,t_2) + T_{match}(p_1,t_1,p_2,t_2)
$$

Die Laufzeit des PNVF2 setzt sich somit aus der Laufzeit für die Permutation, der Matrizengenerierung und dem Matching zusammen.

4.4.2. Beweismethode

Nachdem die Rekursionsgleichungen erfolgreich aufgestellt wurden, müssen sie gelöst werden. Zur Berechnung einer geeigneten Lösung werden in der Literatur verschiedene Verfahren beschrieben. Nach [Cor+10, S. 67 ff.] stehen beispielsweise die Substitutions-

und die Mastermethode zur Verfügung. Aber nicht alle Verfahren sind geeignet, um zu einer gültigen Lösung zu gelangen. Die Mastermethode benötigt eine Rekursionsgleichung in der Form:

$$T(n) = a \cdot T(n/b) + f(n)$$

Dabei sind sowohl a als auch b konstante Faktoren [vgl. Cor+10, S. 96]. Offensichtlich entsprechen die Rekursionsgleichungen des VF2 und des PNVF2 nicht diesem Schema. Lässt man die verschiedene Anzahl an Parametern außen vor, fällt direkt die Inkompatibilität der Faktoren vor der rekursiven Funktion $T(n/b)$ auf. Die Mastermethode benötigt einen konstanten Faktor, während die entsprechenden Werte m, p_2 und t_2 der beiden Algorithmen mit zunehmender Rekursionstiefe kleiner werden und damit nicht konstant sind. Die Mastermethode ist also auf dieses Problem nicht anwendbar.

Die Substitutionsmethode verwendet das Prinzip einer vollständigen Induktion. Dazu muss zuerst eine geeignete Laufzeit erraten werden, welche danach mittels Induktion bewiesen wird. Der Begriff der Substitution spielt dabei auf das Ersetzen der rekursiven Funktion im Induktionsschritt an [vgl. Cor+10, S. 85 ff.]. Als Beispiel diene ein Algorithmus mit der Rekursionsgleichung:

$$\begin{aligned} T(0) &= c \\ T(n) &= T(n-1) + n \end{aligned}$$

Die Gleichung legt nahe, dass der Algorithmus in $\mathcal{O}(n^2)$ liegt. Bei einem Beweis dieser Vermutung könnte folgender Induktionschritt vollzogen werden (ZZ $n-1 \longrightarrow n$):

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\underset{IB}{\leq} c(n-1)^2 + n && \text{Anwendung der IB mittels Substitution der rekursiven Funktion} \\ &= cn^2 - 2cn + c + n \\ &\leq cn^2 - 2cn + cn + cn && \text{da } c \text{ und } n \geq 1 \\ &= cn^2 \end{aligned}$$

Allerdings besitzt diese Methode den Nachteil, dass zuerst eine gute Schätzung abgegeben werden muss. Das kann für eine komplexe Rekursionsgleichung sehr schwierig werden. Als Ausweg besteht die Möglichkeit zuerst, einen Rekursionsbaum aufzustellen. Die Schätzung mittels Rekursionsbaum hat außerdem den Vorteil, dass relativ einfach eine

scharfe Schranke gefunden werden kann [vgl. Cor+10, S. 89 ff.]. Im Folgenden werden bei den einfachen Rekursionsgleichungen Schätzungen „erraten“ und erst bei den komplexen Gleichungen Rekursionsbäume aufgestellt.

4.4.3. Best-Case

Bei der Analyse der Algorithmen wird zwischen ihrem Verhalten im besten und im schlechtesten Fall unterschieden. Für eine Problemgröße bedeutet der Best-Case, dass der Algorithmus mit den geringsten Kosten ausgeführt wird, die möglich sind. Dazu müssen die gegebenen Probleminstanzen bestimmte Eigenschaften haben [vgl. Cor+10, S. 27 ff.].

Best-Case für den VF2

Bei dem VF2 tritt der Best-Case ein, wenn zwei gleich große Graphen untersucht werden, für die im rekursiven Abstieg pro Ebene immer nur ein Kandidatenpaar $feasible$ erfüllt [vgl. Cor+99, S. 4]. Eine weitere Möglichkeit ist, dass beide Graphen keine Kanten besitzen. Aufgrund der fehlenden Kanten ist jede mögliche Knotenzuordnung auch ein gültiger Match. Da dadurch auch alle möglichen Teilmatches $feasible$ sind, tritt auf dem Weg bis zum Rekursionsanker auch kein Backtracking auf.

Spezialisierung der allgemeinen Rekursionsgleichung auf den Best-Case

$$
\begin{aligned}
T_{best}(0,m) &= 1 \\
T_{best}(n,m) &= c + c_1 n + c_2 m + 1 \cdot \Big[0c_3 + 0c_4 + c_5 + 0c_5' + 0c_5'' \\
&\qquad + c_6 + 0c_6' + 0c_6''\Big] + 1 \cdot T_{best}(n-1, m-1) \\
&= c + c_1 n + c_2 m + c_5 + c_6 + T_{best}(n-1, m-1) && \text{kantenlose Graphen} \\
&= c + c_1 n + c_2 n + c_5 + c_6 + T_{best}(n-1, n-1) && \text{da } n = m \\
&\geq c_1 n + c_2 n + T_{best}(n-1, n-1) \\
&\geq 2n + T_{best}(n-1, n-1) && \text{da } c_1 + c_2 \geq 2
\end{aligned}
$$

Die Rekursionsgleichung wurde auf den geschilderten Spezialfall übertragen. Da kein Knoten Nachbarn besitzt, fallen die meisten Summen weg. Zum Schluss wird die Gleichung einmal nach unten abgeschätzt. Dies dient der Vereinfachung der folgenden Rechenschritte. Die Abschätzung beeinträchtigt nicht die Genauigkeit des Ergebnisses.

Abschätzung des Aufwands

$$\begin{aligned}\sum_{i=1}^{n}(2i)+1 &= 2\cdot\frac{n(n+1)}{2}+1\\ &= n^2+n+1\\ &> n^2\\ &\longrightarrow T_{best}(n,m)\in\Omega(n^2)\end{aligned}$$

Zur Abschätzung der entstehenden Kosten für ein beliebiges n wird eine Summe über sämtliche Rekursionsebenen hinweg gebildet. Da die entstehende Summe als Komplexitätsmaß viel zu genau ist und im Allgemeinen durch einen Summanden dominiert wird, wird sie noch einmal nach unten abgeschätzt.

Theorem:
Für $n = m \geq 1 = n_0$ und $c = 1$ gelte:

$$T_{best}(n,m) \geq 2n + T_{best}(n-1,n-1) \geq cn^2$$

Beweis der Schätzung per Substitution

Induktionsanfang: $(n = 1 = n_0)$

$$\begin{aligned}T_{best}(n,m) &= T_{best}(1,1)\\ &\geq 2n + T_{best}(n-1,n-1)\\ &= 2 + T_{best}(0,0)\\ &= 2+1\\ &> 1\cdot 1^2\\ &= cn^2\end{aligned}$$

Induktionsschritt: $(n-1 \longrightarrow n)$

$$\begin{aligned}
T_{best}(n,m) &= T_{best}(n,n) \\
&\geq 2n + T_{best}(n-1,n-1) \\
&\underset{IB}{\geq} 2n + c(n-1)^2 && \text{IB per Substitution} \\
&= 2n + cn^2 - 2cn + c \\
&= n^2 + 2n - 2n + 1 \\
&= n^2 + 1 \\
&> cn^2
\end{aligned}$$

Best-Case für den PNVF2

Für den PNVF2 muss die Betrachtung differenzierter erfolgen. Zum einen sind die anfallenden Kosten, wenn gar kein Match möglich ist, von Interesse und zum anderen die Kosten für einen erfolgreichen Match.

$$T(p_1,t_1,p_2,t_2) = T_{shuffle}(p_1,t_1,p_2,t_2) + T_{matrizen}(p_1,t_1,p_2,t_2) + T_{match}(p_1,t_1,p_2,t_2)$$

Die Laufzeit des PNVF2 setzt sich aus den Aufwänden für die Permutation, die Matrizengenerierung und das Matching zusammen. Bei der Ausführung werden als Erstes die Knoten in $\Theta(p_1+p_2+t_1+t_2)$ permutiert und zwar unabhängig davon, ob ein Match gefunden werden kann. Danach werden die beiden beschriebenen Fälle betrachtet.

Der erste Fall kann bei der Aufstellung der semantischen Abbildbarkeitsmatrizen erkannt werden. Falls schon für den ersten Knoten kein unterstützender Zielknoten gefunden wird, bricht der Algorithmus in $\Theta(p_1+p_2+t_1+t_2)$ ab. Sollte allerdings erst der letzte Knoten keinen unterstützenden Zielknoten besitzen, benötigt der Algorithmus eine Laufzeit von $\Theta(p_1p_2+t_1t_2)$.

Wenn die Prüfung erfolgreich bestanden wird, versucht der Algorithmus einen Match zu finden. Da T_{match} im Allgemeinen teurer ist als $T_{shuffle}$ und $T_{matrizen}$, werden in der Betrachtung des Best-Case und Worst-Case nur die jeweiligen Teilfunktionen $T_{match,best}$ und $T_{match,worst}$ untersucht. Zur Adaption der Gleichungen auf den Spezialfall müssen wieder Annahmen über die Eigenschaften der beiden Petrinetze getroffen werden. Diese ergeben sich analog zum VF2. Es werden also gleich große kantenlose Netze untersucht, bei denen jeweils alle Stellen und Transitionen die gleichen Eigenschaften (Markierungen, Labels usw.) aufweisen.

Spezialisierung der allgemeinen Rekursionsgleichung auf den Best-Case

$$
\begin{aligned}
T_{match,best}(0,0,p_2,t_2) &= 1 \\
T_{match,best}(p_1,0,p_2,t_2) &= c + c_1p_1 + c_2p_2 + c_5 + c_6 + 1 \cdot T_{match,best}(p_1-1,0,p_2-1,t_2) \\
&\geq 2p_1 + T_{match,best}(p_1-1,0,p_1-1,0) \\
T_{match,best}(0,t_1,p_2,t_2) &= d + d_1t_1 + d_2t_2 + d_5 + d_6 + 1 \cdot T_{match,best}(0,t_1-1,p_2,t_2-1) \\
&\geq 2t_1 + T_{match,best}(0,t_1-1,0,t_1-1)
\end{aligned}
$$

$$
T_{match,best}(p_1,t_1,p_2,t_2) = \begin{cases} T_{match,P,best}(p_1,t_1,p_2,t_2) & \text{bei Stellen} \\ T_{match,T,best}(p_1,t_1,p_2,t_2) & \text{bei Transitionen} \end{cases}
$$

$$
\begin{aligned}
T_{match,P,best}(p_1,t_1,p_2,t_2) &= c + c_1p_1 + c_2p_2 + 1 \cdot \Big[0c_3 + 0c_4 + c_5 + 0c_5' + 0c_5'' \\
&\quad + c_6 + 0c_6' + 0c_6''\Big] + 1 \cdot T_{match,best}(p_1-1,t_1,p_2-1,t_2) \\
&\geq 2p_1 + T_{match,best}(p_1-1,t_1,p_1-1,t_1) \\
T_{match,T,best}(p_1,t_1,p_2,t_2) &= d + d_1t_1 + d_2t_2 + 1 \cdot \Big[0d_3 + 0d_4 + d_5 + 0d_5' + 0d_5'' \\
&\quad + d_6 + 0d_6' + 0d_6''\Big] + 1 \cdot T_{match,best}(p_1,t_1-1,p_2,t_2-1) \\
&\geq 2t_1 + T_{match,best}(p_1,t_1-1,p_1,t_1-1)
\end{aligned}
$$

Durch die bipartiten Eigenschaften der Petrinetze und dem Nichtdeterminimus des Algorithmus erscheint die spezialisierte Rekursionsgleichung komplexer als sie ist. Da auch für die Petrinetze der Best-Case ein kantenloses Netz ist, sind alle Summen für die Nachbarschaftsprüfung und die meisten Summen der Zustandsaktualisierung entfallen.

Abschätzung des Aufwands

$$
\begin{aligned}
\sum_{i=1}^{p_1}(2i) + \sum_{i=1}^{t_1}(2i) + 1 &= 2 \cdot \frac{p_1(p_1+1)}{2} + 2 \cdot \frac{t_1(t_1+1)}{2} + 1 \\
&= p_1^2 + p_1 + t_1^2 + t_1 + 1 \\
&> p_1^2 + t_1^2 \\
&\longrightarrow T_{match,best}(p_1,t_1,p_2,t_2) \in \Omega(p_1^2 + t_1^2)
\end{aligned}
$$

Der Algorithmus entscheidet sich auf jeder Rekursionsebene nichtdeterministisch, welche Knotenart gematcht werden soll. Bei der Aufstellung der Aufwandssumme wird dieser Umstand nicht mehr hervorgehoben. Alle möglichen Rekursionsabläufe werden durch die Kommutativität der Addition und dem nicht eintretenden Backtracking korrekt repräsentiert.

Theorem:
Für $p_1 = p_2$, $t_1 = t_2$, $p_1 + t_1 \geq 1$ und $c = 1$ gelte:

$$\begin{aligned}
T_{match,best}(0,0,p_2,t_2) &= 1 \\
T_{match,best}(p_1,0,p_2,t_2) &\geq 2p_1 + T_{match,best}(p_1-1,0,p_1-1,0) \\
T_{match,best}(0,t_1,p_2,t_2) &\geq 2t_1 + T_{match,best}(0,t_1-1,0,t_1-1) \\
T_{match,P,best}(p_1,t_1,p_2,t_2) &\geq 2p_1 + T_{match,best}(p_1-1,t_1,p_1-1,t_1) \\
T_{match,T,best}(p_1,t_1,p_2,t_2) &\geq 2t_1 + T_{match,best}(p_1,t_1-1,p_1,t_1-1)
\end{aligned}$$

$$\begin{aligned}
T_{match,best}(p_1,t_1,p_2,t_2) &= \begin{cases} T_{match,P,best}(p_1,t_1,p_2,t_2) & \text{bei Stellen} \\ T_{match,T,best}(p_1,t_1,p_2,t_2) & \text{bei Transitionen} \end{cases} \\
&\geq cp_1^2 + ct_1^2
\end{aligned}$$

Beweis der Schätzung per Substitution

Induktionsanfang:

1. Fall: ($p_1 = p_2 = 1$, $t_1 = t_2 = 0$)

$$\begin{aligned}
T_{match,best}(p_1,t_1,p_2,t_2) &= T_{match,best}(1,0,1,0) \\
&\geq 2 \cdot 1 + T_{match,best}(0,0,0,0) \\
&= 2 + 1 \\
&> 1 \cdot 1^2 + 1 \cdot 0^2 \\
&= cp_1^2 + ct_1^2
\end{aligned}$$

2. Fall: ($p_1 = p_2 = 0$, $t_1 = t_2 = 1$)
Der Beweis für den zweiten Fall erfolgt analog zum Ersten.

Induktionsschritt:

1. Fall: $(p_1 - 1 \longrightarrow p_1)$

$$\begin{aligned} T_{match,best}(p_1, t_1, p_2, t_2) &\geq 2p_1 + T_{match,best}(p_1 - 1, t_1, p_1 - 1, t_1) \\ &\underset{IB}{\geq} 2p_1 + c(p_1 - 1)^2 + ct_1^2 \\ &= 2p_1 + (cp_1^2 - 2cp_1 + c) + ct_1^2 \\ &= p_1^2 + 2p_1 - 2p_1 + t_1^2 + 1 \\ &= p_1^2 + t_1^2 + 1 \\ &> cp_1^2 + ct_1^2 \end{aligned}$$

2. Fall: $(t_1 - 1 \longrightarrow t_1)$
Der Beweis für den zweiten Fall erfolgt analog zum Ersten.

4.4.4. Worst-Case

Worst-Case für den VF2

Der Worst-Case sind zwei fast vollständige Graphen, bei denen der Suchraum vollständig durchsucht werden muss. Als Beispiel diene ein Zielgraph mit vielen Symmetrien [vgl. Cor+99, S. 4]. Dieser Fall kann eintreten, wenn die Abbildung der Knoten im letzten Rekursionsschritt fehlschlägt oder der Benutzer alle gefundenen Matches ablehnt. Im Gegensatz zum Best-Case kann m deutlich höher ausfallen als n.

Spezialisierung der allgemeinen Rekursionsgleichung auf den Worst-Case

$$\begin{aligned} T_{worst}(0, m) &= c' \\ T_{worst}(n, m) &= c + c_1 n + c_2 m + m \cdot \Big[c_3(n-1) + c_4(m-1) + c_5 + c_5' n + c_5'' m \\ &\quad + c_6 + c_6' n + c_6'' m\Big] + m \cdot T_{worst}(n-1, m-1) \\ &\leq c + c_1 n + c_2 m + m \cdot \Big[c_3 n + c_4 m + c_5 + c_5' n + c_5'' m + c_6 + c_6' n + c_6'' m\Big] \\ &\quad + m \cdot T_{worst}(n-1, m-1) \\ &= c + c_1 n + c_2 m + c_3 nm + c_4 m^2 + c_5 m + c_5' nm + c_5'' m^2 + c_6 m + c_6'' nm \\ &\quad + c_6' m^2 + m \cdot T_{worst}(n-1, m-1) \end{aligned}$$

$$
\begin{aligned}
&\leq cm^2 + c_1m^2 + c_2m^2 + c_3m^2 + c_4m^2 + c_5m^2 + c_5'm^2 + c_5''m^2 + c_6m^2 \\
&\quad + c_6'm^2 + c_6''m^2 + m \cdot T_{worst}(n-1, m-1) \\
&= c'm^2 + m \cdot T_{worst}(n-1, m-1)
\end{aligned}
$$

Die Ausgangsgleichung für den Worst-Case deckt sich mit der allgemeinen Rekursionsgleichung. Da mit dieser keine übersichtliche Rechnung möglich ist und einige Summanden deutlich schneller wachsen als andere, wird sie nach oben abgeschätzt. Für die zweite Formel ist bekannt, dass sowohl n als auch m großer sind als 0. Außerdem ist n in jedem Fall kleiner oder gleich m. Der größte Term, der m betrifft, ist m^2. Mit diesem Wissen wird der Teil der Formel, der nicht den Rekursionsaufruf betrifft, auf $c'm^2$ abgeschätzt. c' ist die Summe aller konstanten Operationen auf der jeweiligen Rekursionsebene.

Abschätzung des Aufwands

Die Gleichung ist allerdings noch zu komplex, um ungezielt eine Summenformel aufzustellen. Zur Entwicklung einer geeigneten Abschätzung wird im Folgenden zuerst der Rekursionsbaum erstellt, danach die Formel abgeleitet und anschließend vereinfacht.

Rekursionsbaum

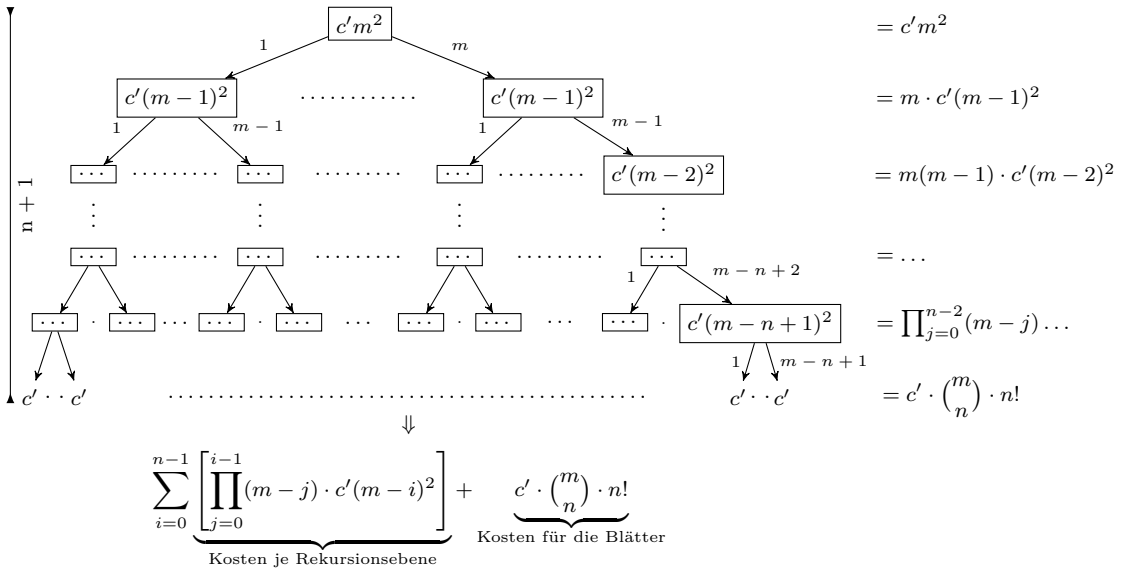


Abbildung 4.16.: VF2-Rekursionsbaum für den Worst-Case

Der Rekursionsbaum nach [Cor+10, S. 89 ff.] stellt eine vollständige Traversierung des Suchraumes grafisch dar. Angenommen, es soll ein Match von einem Graphen G_1 in einem Graphen G_2 gefunden werden. Wie anfangs erwähnt, entspricht n der Anzahl der ungematchten Knoten in G_1 und m der Anzahl der ungematchten Knoten in G_2. Der Matchingprozess beginnt in der Wurzel des Baumes. Es wurde noch kein Knoten abgebildet. Nachdem mit den Kosten $c'm^2$ die $feasible$ Paare gefunden wurden, geht der Algorithmus für jedes Paar in die nächste Rekursionsebene über. Dieser Vorgang wird durch die Kanten von der Wurzel zu ihren Kindern veranschaulicht. Da mit jedem abgebildeten Knoten die Menge der freien Knoten kleiner wird, sinken auf jeder Ebene die Kosten. Während auf der ersten Ebene noch $c'm^2$ Operationen je Baumknoten benötigt werden, sind es auf der vorletzten Ebene nur noch $c'(m-n+1)^2$ Operationen. Die Anzahl der jeweiligen Kinder hängt vom Zielgraphen ab und ist nicht konstant. Außerdem wird sie auf jeder Ebene kleiner. Wenn ein Blatt erreicht wird, ist ein gültiger Match gefunden worden. Die Tiefe eines Blattes beträgt $n+1$, da zuerst n Knoten gematcht werden müssen, bevor der Rekursionsanker erreicht wird. Auf der rechten Seite sind die aufsummierten Kosten je Ebene aufgelistet. Dazu werden die Kosten aller Baumknoten der entsprechenden Ebene aufsummiert. Das Produkt vor dem Term $c'(m-...)^2$ ist die Anzahl der Baumknoten in dieser Tiefe. Bei jedem Rekursionsschritt wird dieses Produkt um einen Faktor ergänzt. Durch die Aufmultiplizierung bis zum Rekursionsanker werden schließlich $\binom{m}{n} \cdot n!$ Blätter erzeugt.

$$\underbrace{\sum_{i=0}^{n-1} \left[\prod_{j=0}^{i-1} (m-j) \cdot c'(m-i)^2 \right]}_{\text{Kosten je Rekursionsebene}} + \underbrace{c' \cdot \binom{m}{n} \cdot n!}_{\text{Kosten für die Blätter}}$$

Die abgeleitete Formel besteht aus zwei Summanden. Die Summe steht für die Kosten der oberen Ebenen und der Binomialkoeffizient für die der Blätter. Das Summenargument repräsentiert die Kosten einer bestimmten Ebene. Da die Wurzel mit $c'm^2$ startet, dieser Wert allerdings mit zunehmender Tiefe kleiner wird, läuft die Summe von 0 bis $n-1$ und zieht den aktuellen Index von m ab. Das Produkt vor den Knotenkosten steht für die Anzahl der Baumknoten je Ebene.

Vereinfachung der Abschätzung

$$\begin{aligned}
&\sum_{i=0}^{n-1}\left[\prod_{j=0}^{i-1}(m-j)\cdot c'(m-i)^2\right] + c'\cdot\binom{m}{n}\cdot n! \\
&\quad\leq c'm^2\cdot\sum_{i=0}^{n-1}\left[\prod_{j=0}^{i-1}(m-j)\right] + c'\cdot\binom{m}{n}\cdot n! \\
&\quad= c'm^2\cdot\sum_{i=0}^{n-1}\left[\binom{m}{i}\cdot i!\right] + c'\cdot\binom{m}{n}\cdot n! \\
&\quad\leq c'm^2\cdot n\cdot\binom{m}{n-1}\cdot(n-1)! + c'\cdot\binom{m}{n}\cdot n! \\
&\quad\leq c'nm^2\cdot\binom{m}{n}\cdot n! + c'\cdot\binom{m}{n}\cdot n! \\
&\quad= (c'nm^2+c')\cdot\binom{m}{n}\cdot n! \\
&\quad\leq 2c'nm^2\cdot\binom{m}{n}\cdot n! \\
&\quad\longrightarrow T_{worst}(n,m)\in\mathcal{O}(nm^2\cdot\binom{m}{n}\cdot n!)
\end{aligned}$$

Theorem:
Für $n \geq 1 = n_0$ und $c = 2c'$ gelte:

$$T_{worst}(n,m) \leq c'm^2 + m\cdot T_{worst}(n-1,m-1) \leq cnm^2\cdot\binom{m}{n}\cdot n!$$

Beweis der Schätzung per Substitution

Induktionsanfang: $(n = 1 = n_0)$

$$\begin{aligned}
T_{worst}(n,m) = T_{worst}(1,m) &\leq c'm^2 + m\cdot T_{worst}(0,m-1) \\
&= c'\cdot(m^2+m) \\
&\leq 2c'\cdot(m^2\cdot m) \\
&= cm^2\cdot m \\
&= cnm^2\cdot\binom{m}{n}\cdot n!
\end{aligned}$$

Induktionsschritt: $(n-1 \longrightarrow n)$

$$
\begin{aligned}
T_{worst}(n,m) &\leq c'm^2 + m \cdot T_{worst}(n-1, m-1) \\
&\underset{IB}{\leq} c'm^2 + m \cdot c(n-1)(m-1)^2 \cdot \binom{m-1}{n-1} \cdot (n-1)! \\
&\leq cm^2 + m \cdot c(n-1)(m-1)^2 \cdot \binom{m-1}{n-1} \cdot (n-1)! \\
&= c\left[m^2 + m \cdot (n-1)(m-1)^2 \cdot \binom{m-1}{n-1} \cdot (n-1)!\right] \\
&= c\left[m^2 + m \cdot (n-1)(m-1)^2 \cdot \frac{(m-1)!}{((m-1)-(n-1))!}\right] \\
&= c\left[m^2 + (n-1)(m-1)^2 \cdot \frac{m(m-1)!}{(m-1-n+1)!}\right] \\
&= c\left[m^2 + (n-1)(m-1)^2 \cdot \frac{m!}{(m-n)!}\right] \\
&\leq c\left[m^2 + (n-1)m^2 \cdot \binom{m}{n} \cdot n!\right] \\
&\leq c\left[m^2 \cdot \binom{m}{n} \cdot n! + (n-1)m^2 \cdot \binom{m}{n} \cdot n!\right] \\
&= cnm^2 \cdot \binom{m}{n} \cdot n!
\end{aligned}
$$

Worst-Case für den PNVF2

Die schlechteste Laufzeit wird erreicht, wenn alle Stellen und Transitionen identisch gelabelt sind. Im Folgenden wird angenommen, dass zwei vollständige Petrinetze untersucht werden und der vollständige Suchraum durchsucht wird.

Spezialisierung der allgemeinen Rekursionsgleichung auf den Worst-Case

$$
\begin{aligned}
T_{match,worst}(0,0,p_2,t_2) &= e \\
T_{match,worst}(p_1,0,p_2,t_2) &= c + c_1p_1 + c_2p_2 + c_5p_2 + c_6p_2 \\
&\quad + p_2 \cdot T_{match,worst}(p_1-1,0,p_2-1,t_2) \\
&\leq ep_2 + p_2 \cdot T_{match,worst}(p_1-1,0,p_2-1,t_2)
\end{aligned}
$$

$$\begin{aligned} T_{match,worst}(0,t_1,p_2,t_2) &= d + d_1t_1 + d_2t_2 + d_5t_2 + d_6t_2 \\ &\quad + t_2 \cdot T_{match,worst}(0,t_1-1,p_2,t_2-1) \\ &\leq et_2 + t_2 \cdot T_{match,worst}(0,t_1-1,p_2,t_2-1) \end{aligned}$$

$$T_{match,worst}(p_1,t_1,p_2,t_2) = \begin{cases} T_{match,P,worst}(p_1,t_1,p_2,t_2) & \text{bei Stellen} \\ T_{match,T,worst}(p_1,t_1,p_2,t_2) & \text{bei Transitionen} \end{cases}$$

$$\begin{aligned} T_{match,P,worst}(p_1,t_1,p_2,t_2) &= c + c_1p_1 + c_2p_2 + p_2 \cdot \Big[c_3(t_1-1) + c_4(t_2-1) + c_5 \\ &\quad + c_5't_1 + c_5''t_2 + c_6 + c_6't_1 + c_6''t_2\Big] \\ &\quad + p_2 \cdot T_{match,worst}(p_1-1,t_1,p_2-1,t_2) \\ &\leq ep_2t_2 + p_2 \cdot T_{match,worst}(p_1-1,t_1,p_2-1,t_2) \end{aligned}$$

$$\begin{aligned} T_{match,T,worst}(p_1,t_1,p_2,t_2) &= d + d_1t_1 + d_2t_2 + t_2 \cdot \Big[d_3(p_1-1) + d_4(p_2-1) + d_5 \\ &\quad + d_5'p_1 + d_5''p_2 + d_6 + d_6'p_1 + d_6''p_2\Big] \\ &\quad + t_2 \cdot T_{match,worst}(p_1,t_1-1,p_2,t_2-1) \\ &\leq ep_2t_2 + t_2 \cdot T_{match,worst}(p_1,t_1-1,p_2,t_2-1) \end{aligned}$$

Durch die zwei Knotenklassen dominiert bei den Abschätzungen nicht das Quadrat einer bestimmten Problemgröße die Kosten für den jeweiligen Rekursionsschritt (wie m^2 beim VF2), sondern das Produkt p_2t_2. Der konstante Faktor e gilt für alle Gleichungen und schätzt die maximal auftretende Anzahl an konstanten Operationen nach oben ab. Dadurch werden im Folgenden die Sonderfallbehandlungen für die konstanten Faktoren vermieden [vgl. Cor+10, S. 37].

Abschätzung des Aufwands

Erstellung der Aufwandssumme

Bei der Ableitung einer geeigneten Aufwandssumme steht man vor dem Problem, den Nichtdeterminismus korrekt abzubilden. Da bei einem Netz mit Stellen und Transitionen unzählige Ausführungspfade möglich sind, wird zuerst ein einfacher Fall betrachtet. Es

wird angenommen, dass der PNVF2 als Erstes die Stellen und danach die Transitionen abbildet. Aus der Analyse des VF2 ist schon eine Aufwandssumme für allgemeine Graphen bekannt. Sie lautet:

$$\sum_{i=0}^{n-1} \underbrace{\left[\prod_{j=0}^{i-1}(m-j)\cdot c'(m-i)^2\right]}_{\text{Kosten je Rekursionsebene}} + \underbrace{c'\cdot\binom{m}{n}\cdot n!}_{\text{Kosten für die Blätter}}$$

Diese kann auf den PNVF2 adaptiert werden. Das Ergebnis sieht (unter der Berücksichtigung des einfachen Falles) wie folgt aus:

$$\underbrace{\sum_{i=0}^{p_1-1}\left[\prod_{j=0}^{i-1}(p_2-j)\cdot e(p_2-i)t_2\right]}_{\text{Kosten für die Stellen}} + \binom{p_2}{p_1}\cdot p_1!\cdot \underbrace{\sum_{i=0}^{t_1-1}\left[\prod_{j=0}^{i-1}(t_2-j)\cdot e(t_2-i)\right]}_{\text{Kosten für die Transitionen}}$$

$$+\underbrace{e\cdot\binom{p_2}{p_1}\cdot p_1!\cdot\binom{t_2}{t_1}\cdot t_1!}_{\text{Kosten für die Blätter}}$$

Der erste Summand entspricht der Summe in der Aufwandsabschätzung des VF2. Nachdem alle Stellen abgebildet wurden, geht der Algorithmus in die nächste Rekursionsebene über. Im VF2 sind dies die Blätter. Im PNVF2 beginnt dort allerdings das Matching der Transitionen. Deswegen ist der zweite Summand ein Produkt aus dem Binomialkoeffizienten der Stellen und der Summe der Kosten für die Abbildung der Transitionen. Der dritte Summand entspricht den Kosten der Blätter.

Vereinfachung der Abschätzung

$$\begin{aligned}
&\sum_{i=0}^{p_1-1}\left[\prod_{j=0}^{i-1}(p_2-j)\cdot e(p_2-i)t_2\right] + \binom{p_2}{p_1}\cdot p_1!\cdot\sum_{i=0}^{t_1-1}\left[\prod_{j=0}^{i-1}(t_2-j)\cdot e(t_2-i)\right]\\
&\qquad + e\cdot\binom{p_2}{p_1}\cdot p_1!\cdot\binom{t_2}{t_1}\cdot t_1!\\
&\leq ep_2t_2\cdot\sum_{i=0}^{p_1-1}\left[\prod_{j=0}^{i-1}(p_2-j)\right] + et_2\cdot\binom{p_2}{p_1}\cdot p_1!\cdot\sum_{i=0}^{t_1-1}\left[\prod_{j=0}^{i-1}(t_2-j)\right]\\
&\qquad + e\cdot\binom{p_2}{p_1}\cdot p_1!\cdot\binom{t_2}{t_1}\cdot t_1!
\end{aligned}$$

$$= ep_2t_2 \cdot \sum_{i=0}^{p_1-1} \left[\binom{p_2}{i} \cdot i! \right] + et_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \sum_{i=0}^{t_1-1} \left[\binom{t_2}{i} \cdot i! \right] + e \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!$$

$$\leq ep_2t_2 \cdot p_1 \cdot \binom{p_2}{p_1 - 1} \cdot (p_1 - 1)! + et_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot t_1 \cdot \binom{t_2}{t_1 - 1} \cdot (t_1 - 1)!$$
$$+ e \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!$$

$$\leq ep_2t_2 \cdot p_1 \cdot \binom{p_2}{p_1} \cdot p_1! + et_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot t_1 \cdot \binom{t_2}{t_1} \cdot t_1! + e \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!$$

$$= \left[ep_1p_2t_2 + et_1t_2 \cdot \binom{t_2}{t_1} \cdot t_1! + e \cdot \binom{t_2}{t_1} \cdot t_1! \right] \cdot \binom{p_2}{p_1} \cdot p_1!$$

$$\leq [ep_1p_2t_2 + et_1t_2 + e] \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!$$

Die Aufwandssumme für den einfachen Fall wurde erfolgreich vereinfacht. Allerdings stellt das Ergebnis keine gute Abschätzung für den Worst-Case-Aufwand des PNVF2 dar. Hätte die Strategie umgekehrt ausgesehen, dass zuerst die Transitonen und danach die Stellen abgebildet worden wären, sähen die Koeffizienten anders aus. Anstelle der Produkte $p_1p_2t_2$ und t_1t_2 ständen dort $p_2t_1t_2$ und p_1p_2. Um alle Pfade, die durch den Nichtdeterminismus möglich sind, abzudecken, wird das Ergebnis noch mal nach oben abgeschätzt.

$$\begin{aligned}
&[ep_1p_2t_2 + et_1t_2 + e] \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\leq [ep_1p_2t_1t_2 + ep_1p_2t_1t_2 + ep_1p_2t_1t_2] \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&= cp_1p_2t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1! \qquad \text{mit } c = 3e \in \mathbb{N} \\
&\longrightarrow T_{match,worst}(p_1, t_1, p_2, t_2) \in \mathcal{O}(p_1p_2t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!)
\end{aligned}$$

Theorem:
Für $1 \leq p_1 \leq p_2$, $1 \leq t_1 \leq t_2$ und $c = 3e$ gelte:

$$
\begin{aligned}
T_{match,worst}(0,0,p_2,t_2) &= e \\
T_{match,worst}(p_1,0,p_2,t_2) &\leq ep_2 + p_2 \cdot T_{match,worst}(p_1-1,0,p_2-1,t_2) \\
T_{match,worst}(0,t_1,p_2,t_2) &\leq et_2 + t_2 \cdot T_{match,worst}(0,t_1-1,p_2,t_2-1) \\
T_{match,P,worst}(p_1,t_1,p_2,t_2) &\leq ep_2t_2 + p_2 \cdot T_{match,worst}(p_1-1,t_1,p_2-1,t_2) \\
T_{match,T,worst}(p_1,t_1,p_2,t_2) &\leq ep_2t_2 + t_2 \cdot T_{match,worst}(p_1,t_1-1,p_2,t_2-1)
\end{aligned}
$$

$$
\begin{aligned}
T_{match,worst}(p_1,t_1,p_2,t_2) &= \begin{cases} T_{match,P,worst}(p_1,t_1,p_2,t_2) & \text{bei Stellen} \\ T_{match,T,worst}(p_1,t_1,p_2,t_2) & \text{bei Transitionen} \end{cases} \\
&\leq cp_1p_2t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!
\end{aligned}
$$

Beweis der Schätzung per Substitution

Induktionsanfang: $(p_1 = t_1 = 1)$

1. Fall: Match der Stelle

$$
\begin{aligned}
T_{match,worst}(1,1,p_2,t_2) &\leq ep_2t_2 + p_2 \cdot T_{match,worst}(0,1,p_2-1,t_2) \\
&= ep_2t_2 + p_2 \cdot \Big[et_2 + t_2 \cdot T_{match,worst}(0,0,p_2-1,t_2-1)\Big] \\
&= ep_2t_2 + p_2 \cdot \Big[et_2 + t_2 \cdot e\Big] \\
&= ep_2t_2 + ep_2t_2 + ep_2t_2 \\
&= 3ep_2t_2 \\
&= cp_2t_2 \\
&\leq cp_1p_2t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!
\end{aligned}
$$

2. Fall: Match der Transition

Der Beweis für den zweiten Fall erfolgt analog zum Ersten.

Induktionsschritt:

1. Fall: $(p_1 - 1 \longrightarrow p_1)$

$$
\begin{aligned}
&T_{match,worst}(p_1, t_1, p_2, t_2) \\
&\quad \leq ep_2t_2 + p_2 \cdot T_{match,worst}(p_1 - 1, t_1, p_2 - 1, t_2) \\
&\quad \underset{IB}{\leq} ep_2t_2 + p_2 \cdot c(p_1 - 1)(p_2 - 1)t_1t_2 \cdot \binom{p_2 - 1}{p_1 - 1} \cdot (p_1 - 1)! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad \leq cp_2t_2 + p_2 \cdot c(p_1 - 1)(p_2 - 1)t_1t_2 \cdot \binom{p_2 - 1}{p_1 - 1} \cdot (p_1 - 1)! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad = c \cdot \left[p_2t_2 + p_2 \cdot c(p_1 - 1)(p_2 - 1)t_1t_2 \cdot \binom{p_2 - 1}{p_1 - 1} \cdot (p_1 - 1)! \cdot \binom{t_2}{t_1} \cdot t_1! \right] \\
&\quad \leq c \cdot \left[p_2 + p_2 \cdot (p_1 - 1)(p_2 - 1) \cdot \binom{p_2 - 1}{p_1 - 1} \cdot (p_1 - 1)! \right] \cdot t_1t_2 \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad = c \cdot \left[p_2 + p_2 \cdot (p_1 - 1)(p_2 - 1) \cdot \frac{(p_2 - 1)!}{(p_2 - 1 - p_1 + 1)!} \right] \cdot t_1t_2 \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad = c \cdot \left[p_2 + (p_1 - 1)(p_2 - 1) \cdot \frac{p_2!}{(p_2 - p_1)!} \right] \cdot t_1t_2 \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad = c \cdot \left[p_2 + (p_1 - 1)(p_2 - 1) \cdot \binom{p_2}{p_1} \cdot p_1! \right] \cdot t_1t_2 \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad \leq c \cdot \left[p_2 + (p_1 - 1)(p_2 - 1) \right] \cdot t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad \leq c \cdot \left[p_2 + (p_1 - 1)p_2 \right] \cdot t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1! \\
&\quad = cp_1p_2t_1t_2 \cdot \binom{p_2}{p_1} \cdot p_1! \cdot \binom{t_2}{t_1} \cdot t_1!
\end{aligned}
$$

2. Fall: $(t_1 - 1 \longrightarrow t_1)$

Der Beweis für den zweiten Fall erfolgt analog zum Ersten.

4.5. Speicherkomplexität

4.5.1. Speicherkomplexität des VF2

Da die Graphen nicht zum Algorithmus gehören und damit deren Speicherverbrauch uninteressant ist, bezieht sich die Analyse nur auf die verwendeten Datenstrukturen des VF2. Als Repräsentation des aktuellen Matchingzustandes verwendet er die State-Space-Representation (SSR). Alle Rekursionsschritte nutzen dieselbe SSR-Instanz. Es wird also nur einmal Speicher benötigt. Für beide Graphen werden je drei Arrays verwaltet, die die Länge n bzw. m besitzen. Neben der SSR wird auf jeder Rekursionsebene noch etwas zusätzlicher Speicher verbraucht, welcher allerdings konstant ist. Die maximale Tiefe der Rekursion beträgt $n + 1$, daher kann man diesen Zusatzspeicher mit cn abschätzen. Daraus ergibt sich eine Speicherkomplexität von $\Theta(n + m)$ [vgl. Cor+04, S. 1369].

4.5.2. Speicherkomplexität des PNVF2

Der PNVF2 setzt eine zum VF2 vergleichbare SSR ein. Wenn man von den Knotenarten abstrahiert, kann man n mit $p_1 + t_1$ und m mit $p_2 + t_2$ ansetzen. Daneben wird ebenfalls etwas Speicher für jeden Rekursionsschritt benötigt. Daraus ließe sich eine Komplexität von $\Theta(p_1 + p_2 + t_1 + t_2)$ ableiten. Allerdings werden vor dem Matching die semantischen Abbildbarkeitsmatrizen für die Stellen und Transitionen generiert, dadurch verschlechtert sich die Speicherkomplexität auf $\Theta(p_1 p_2 + t_1 t_2)$.

An dieser Stelle wurde ein Kompromiss zwischen der Laufzeit- und Speicherkomplexität geschlossen. Zwei Knoten wiederholt auf semantische Abbildbarkeit zu prüfen, kann durch die vielen Labels schnell sehr teuer werden. Neben der Verringerung der Laufzeit im Matchingprozess war außerdem wichtig, dass mit der Hilfe der Matrizen erkannt werden kann, ob ein Match unmöglich ist. Im optimalen Fall kann der PNVF2 auf den Matchingversuch verzichten.

5. Implementierung

Im folgenden Kapitel wird die Integration des PNVF2 in ReConNet beschrieben. Dazu werden die bisher abstrakten Definitionen der einzelnen Teilfunktionen und Datenstrukturen (siehe Kapitel 4) in konkreten Programmcode übersetzt. Dadurch entstehen weitere Anforderungen an den Algorithmus. Diese resultieren aus den sprachspezifischen Eigenheiten von Java sowie aus ReConNet (siehe Kapitel 1) selbst. Damit diese Anforderungen erfüllt werden können, muss der PNVF2 an den betroffenen Stellen angepasst oder auch konkretisiert werden. Beispielsweise wird in der abstrakten Definition nicht die Umsetzung des Nichtdeterminismus in einer Zielsprache besprochen. Auch bei der tatsächlichen Repräsentation der Petrinetze besteht relativ viel Spielraum. Im Folgenden wird für [Blu13] auf die einzelnen Veränderungen und deren Konsequenzen eingegangen.

5.1. Architektur von ReConNet

Die Umsetzung der rekonfigurierbaren Petrinetze erfolgt in ReConNet in fünf Komponenten, davon sind einige fachlich und andere technisch orientiert. Die Zusammenhänge zwischen den einzelnen Komponenten sollen mit der Abbildung 5.1 verdeutlicht werden. Da das Programm in Java 7 implementiert ist, werden die Komponenten im Code als Packages und ihre Schnittstellen durch eine Kombination aus Java-Interfaces und Klassen repräsentiert.

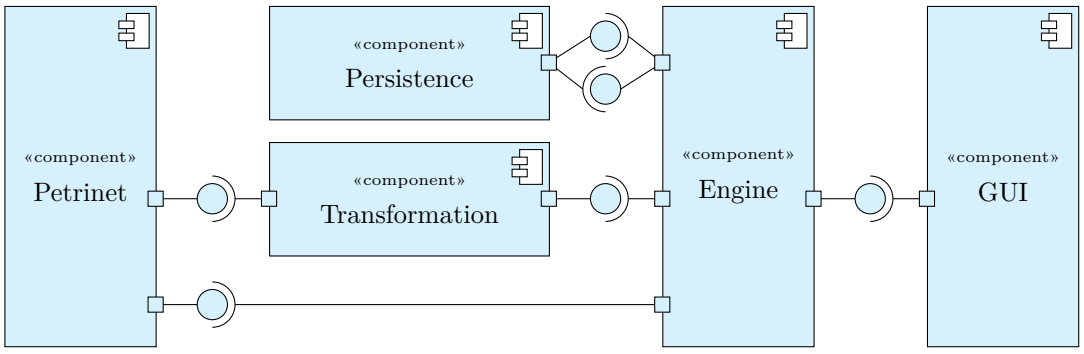


Abbildung 5.1.: ReConNet Komponentendiagramm

Die beiden fachlichen Komponenten sind *Petrinet* und *Transformation*. Die Petrinetzkomponente bildet das Fundament von ReConNet. In ihr werden die Petrinetze, ihre Teilstrukturen und die darauf aufbauenden Operationen umgesetzt. Das betrifft sowohl die einzelnen Klassen für die Stellen und die Transitionen als auch die verschiedenen Kantenarten und natürlich die Petrinetze selbst. Die Transformationskomponente liegt eine Ebene darüber. Sie bietet unter anderem eine Implementierung der Regeln und der darauf aufbauenden Netztransformationen. Außerdem beinhaltet sie den – für die Anwendung einer Regel notwendigen – Matchingalgorithmus. Mit der Hilfe dieser beiden Komponenten können bereits die verschiedenen Szenarien (siehe Unterabschnitt 1.1.1) modelliert werden. Allerdings werden zusätzliche technische Komponenten benötigt, die diese Modellierung unterstützen und die Simulation der Szenarien steuern.

Die drei technischen Komponenten sind *Persistence*, *GUI* und *Engine*. Die Persistenzkomponente dient der Speicherung der einzelnen Petrinetze und Regeln. Um eine gewisse Kompatibilität zu anderen Petrinetzwerkzeugen zu gewährleisten, wird als Speicherformat PNML (Petri Net Markup Language [vgl. Pnm]) verwendet. Allerdings bietet der PNML-Standard keine Möglichkeit der Repräsentation der Regeln und wurde deswegen an den entsprechenden Stellen speziell für die Verwendung in ReConNet angepasst [vgl. Pad+12, S. 7 - 8]. Die Kompatibilität zu anderen Programmen bleibt somit zwar für die Petrinetze erhalten, die Regeln gehen jedoch bei einem Werkzeugwechsel verloren.

Auf der obersten Ebene liegt die grafische Benutzeroberfläche. Sie wird in der Komponente *GUI* umgesetzt und bildet den Einstiegspunkt für den Benutzer. Um die Oberfläche von den fachlichen Komponenten abzukoppeln, liegt zwischen diesen die *Engine*. Sie ist die zentrale Steuerungseinheit von ReConNet. Beispielsweise wird in der *Engine* die Simulation umgesetzt. Dazu schickt die *GUI* der *Engine* eine Nachricht, die die Art der Simulation und die Anzahl der zu vollziehenden Schritte enthält. Nachdem die *Engine* die Nachricht erhalten hat, führt sie entsprechend viele Schaltschritte mit *Petrinet* und Regelanwendungen mit *Transformation* durch.

5.2. Übersicht über die Struktur

Der grobe Aufbau des PNVF2 und seine Einbettung in ReConNet wird in Abbildung 5.2 dargestellt. Um das Diagramm nicht zu komplex werden zu lassen, wird auf die Wiedergabe der Komponentengrenzen und die höheren Klassen, die den PNVF2 verwenden, verzichtet. Die grafische Darstellung teil sich in zwei Teile auf. Auf der linken Seite des

PNVF2 sind die verwendeten Ausgangsstrukturen aufgelistet und auf der rechten Seite die möglichen Ergebnisse.

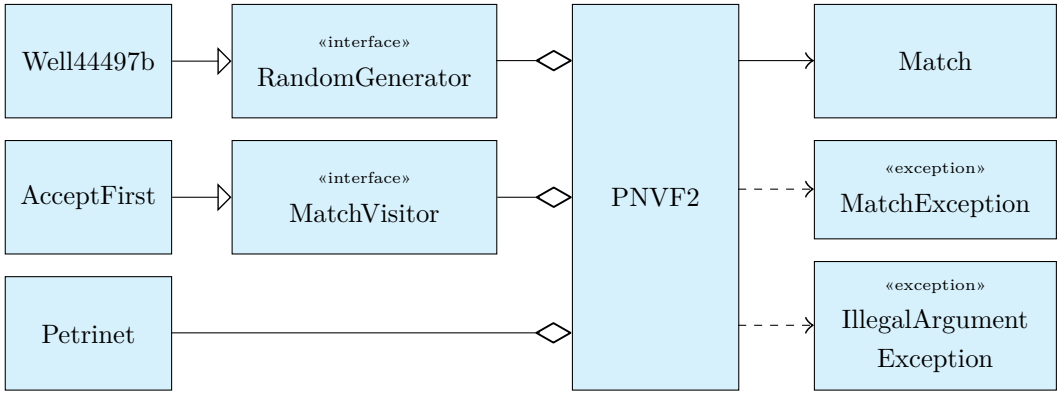


Abbildung 5.2.: vereinfachtes Klassendiagramm des PNVF2

Damit ein Matchingversuch unternommen werden kann, benötigt der PNVF2 zuerst die beiden beteiligten Petrinetze, einen Zufallszahlengenerator und einen $MatchVisitor$. Der $MatchVisitor$ entscheidet, ob ein gefundener Match als gültig angesehen wird. Falls kein spezieller Visitor übergeben wird, verwendet der PNVF2 den Standardvisitor $AcceptFirst$. Dadurch wird der erste gefundene Match direkt vom Algorithmus akzeptiert. Die möglichen Ergebnisse eines Matchingvorganges sind neben einem $Match$ auch eine $MatchException$ oder $IllegalArgumentException$. Falls kein gültiger Match gefunden werden kann oder alle durch einen Visitor abgelehnt werden, wirft der PNVF2 die $MatchException$. Die $IllegalArgumentException$ tritt auf, wenn der PNVF2 auf eine unkorrekte Weise aufgerufen wird. Das ist beispielsweise der Fall, wenn während des Matchingvorganges die PNVF2-Instanz ein weiteres Mal verwendet wird.

5.3. Umsetzung des PNVF2

Bei der Umsetzung des PNVF2 müssen verschiedene Probleme bewältigt werden. Das Erste betrifft die Realisierung des Nichtdeterminismus. Da handelsübliche Computer deterministische Maschinen sind, muss der Nichtdeterminismus auf eine geeignete Weise erzeugt werden. Das nächste Problem betrifft die Erzeugung der Knotenordnungen. Der PNVF2 geht davon aus, dass Ordnungen über die Knoten definiert werden können und er auf die einzelnen Knoten anhand ihrer Position zugreifen kann. Allerdings bietet das Petrinetzmodell in ReConNet keine der beiden Eigenschaften. Nachdem die Ordnungen

feststehen, müssen die Kandidaten generiert werden. Auch an dieser Stelle offenbart sich die abstrakte Definition des PNVF2, denn zur Art und Weise der Berechnungen der einzelnen Kandidatenmengen trifft er keine Aussage. Er verwendet sie lediglich in ihrer Mengenform. Danach wird die Umsetzung von $feasible$ geklärt. Abschließend wird noch auf die Zustandsaktualisierung und die Verwendung des Matchingalgorithmus selbst eingegangen.

Den Ausgangspunkt für die Umsetzung des PNVF2 bildet die in C++ geschriebene vflib [Cor+01b], die neben dem VF und dem Ullmann auch eine Implementierung des VF2 beinhaltet. Die im Folgenden beschriebenen Methoden zur Kandidatengenerierung, der Prüfung auf $feasible$ und der Zustandsaktualisierung orientieren sich an dieser Referenzimplementierung. Da sich die Methoden jedoch im Detail unterscheiden, weicht die Umsetzung des PNVF2 teilweise deutlich ab.

5.3.1. PNVF2-Klasse

Der PNVF2 wird in ReConNet in der gleichnamigen Klasse PNVF2 umgesetzt. Ein Auszug der Variablen und der Methoden ist der Abbildung 5.3 zu entnehmen. Da dieser Auszug schon eine beachtliche Größe aufweist, verzichtet die Darstellung auf die Aufzählung der Hilfsvariablen und -methoden. Außerdem sind die einzelnen Einträge nach Bereich gruppiert. Weil die State-Space-Represenation (SSR) Teil des Zustands einer PNVF2-Instanz ist, sind die Instanzen nicht für eine parallele Nutzung geeignet. Das betrifft sowohl den Zugriff aus verschiedenen Threads als auch einen mögliche Aufruf innerhalb eines $MatchVisitor$s.

5.3.2. Nichtdeterminismus und PRNGs

Eine Kernanforderung an den PNVF2 ist sein nichtdeterministischer Ablauf (siehe Unterabschnitt 3.1.3). Bei der Umsetzung besteht allerdings das Problem, dass Java und die zugrundeliegende Maschine deterministisch agieren. Damit die Anforderung dennoch erfüllt werden kann, verwendet der PNVF2 zur Steuerung der Knotenpermutation und der Kandidatenauswahl einen Zufallszahlengenerator (RNG). Aber das verlagert nur das Problem, denn echte Zufallszahlen sind im Allgemeinen nur eingeschränkt verfügbar. Für die Erzeugung von echten Zufallszahlen verwenden Computer meist die Informationen von den angeschlossenen Geräten (z.B. das Rauschen) oder sie nutzen spezielle Hardware. Allerdings ist die Anzahl der erzeugbaren Zahlen je Zeiteinheit durch die Eigenheiten (und ggf. die Nutzung) der Hardware begrenzt. Die limitierte Verfügbarkeit von echten

Zufallszahlen schränkt ihren Einsatzbereich deutlich ein. Da in der Praxis eine Vielzahl von Anwendungen Zufallszahlen benötigen, bieten die verschiedenen Sprachen Pseudozufallsgeneratoren (PRNG). Diese berechnen aus einer Menge bekannter Ausgangszahlen deterministisch eine neue Pseudozufallszahl. Die bei einem Durchlauf eines Algorithmus entstehende Zahlensequenz richtet sich nach dem Ausgangswert des Generators [vgl. PD08, S. 345 ff.]. Auf den ersten Blick erscheint dieses Vorgehen ungeeignet, da Zufall benötigt wird, aber ein PRNG nur Pseudozufall bietet. Allerdings stellt jede Anwendung ihre eigenen Anforderungen an die Güte der erzeugten Zahlensequenzen. Beispielsweise reicht für viele Programme schon eine scheinbare Gleichverteilung der erzeugten Zahlen.

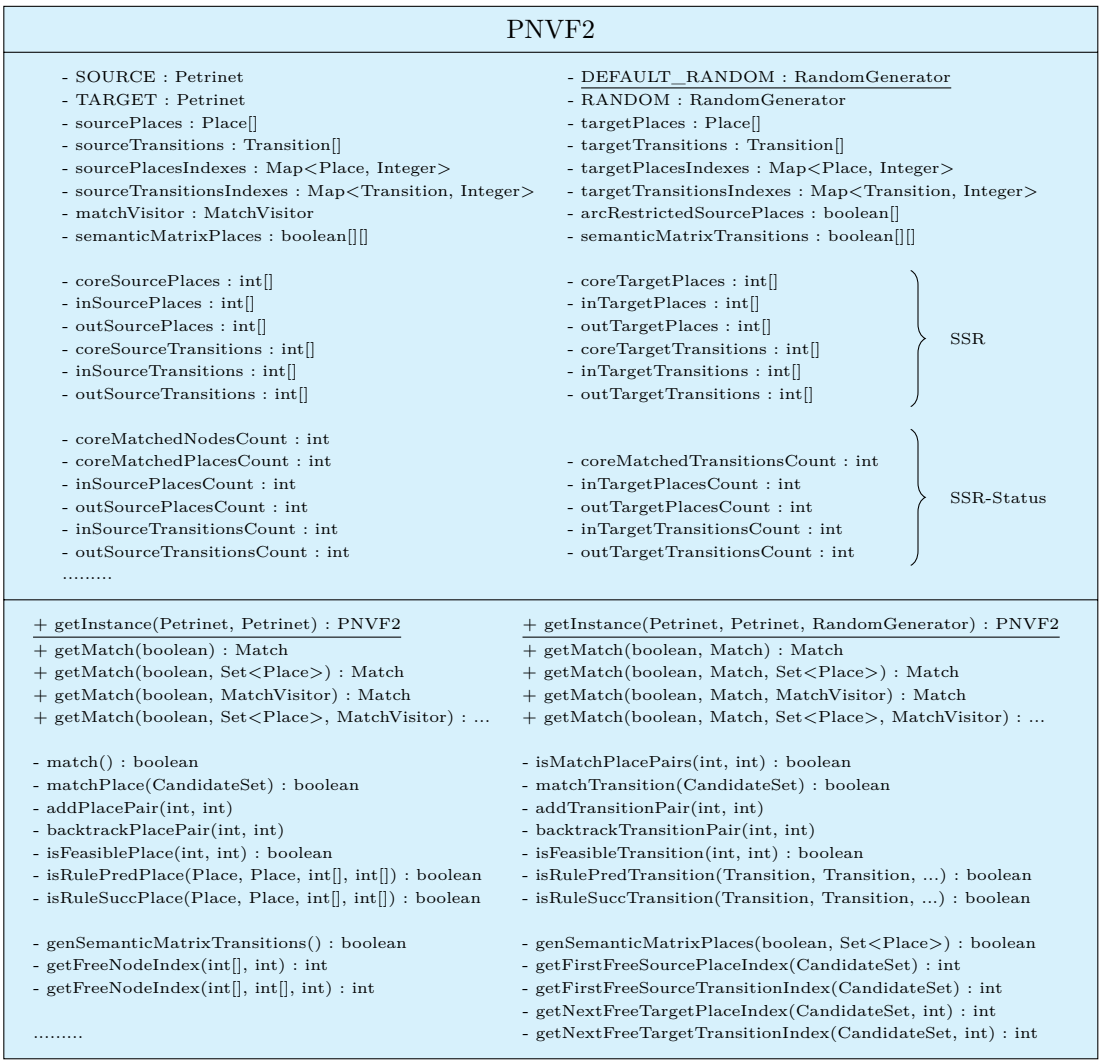


Abbildung 5.3.: Ausschnitt aus der PNVF2-Klasse

Da der PNVF2 die Zufallszahlen zur Steuerung des Ablaufs benötigt und dieser in einer Simulationsumgebung eingesetzt wird, sind die Anforderungen an den PRNG besonders hoch. So können drei Kriterien aufgestellt werden. Das erste Kriterium betrifft die Periodenlänge. Bei der Suche eines Matches werden eine Vielzahl von Zahlen benötigt. Eine zu kurze Periodenlänge sorgt dafür, dass die gleiche Zahlensequenz immer wieder verwendet wird und der PNVF2 somit nicht mehr zufällig agiert. Aus der Menge der benötigten Zahlen folgt auch gleich die nächste Anforderung. Der verwendete PRNG darf nicht zu langsam sein. Geht man davon aus, dass gegebenenfalls bei jedem Rekursionsschritt eine nichtdeterministische Entscheidung getroffen werden muss, entspricht die Menge an benötigten Zufallszahlen ungefähr der Anzahl der Knoten des Rekursionsbaumes. Unter der Annahme, dass zwei gleich große Netze untersucht werden, muss der PRNG bis zu $cp_2!t_2!$ Zahlen liefern. Selbst für kleine Netze mit jeweils acht Stellen und Transitionen werden schon über eine Milliarde Zahlen benötigt. Das letzte und wichtigste Kriterium ist natürlich die Qualität der erzeugten Zahlen.

Im Bereich der PRNGs stehen eine Vielzahl von verschiedenen Algorithmen zur Verfügung. Eine Gegenüberstellung von über 70 Vertretern wurde von L'Ecuyer und Simard bei der Entwicklung ihrer Testbibliothek für Zufallszahlengeneratoren vorgenommen [vgl. LS07, S. 28 - 29]. Diese Gegenüberstellung umfasst neben Informationen zur Periodenlänge und der Laufzeit der Algorithmen auch die Ergebnisse von bis zu über 200 durchgeführten statistischen Tests. Die untersuchten Algorithmen schließen nicht nur Zufallszahlengeneratoren ein, die in Java umgesetzt sind, sondern PRNGs vieler verschiedener Systeme. Beispielsweise werden auch die Generatoren von Mathematica und Matlab betrachtet.

Einer der ersten aufgelisteten Algorithmen ist der PRNG von Java (Java.util.Random), welcher in ca. 30 der durchgeführten Tests durchfällt. Die Kombination aus den fehlgeschlagenen Tests und seiner kurzen Periodenlänge zeigt, dass er für den Einsatz in ReConNet ungeeignet ist. Unter den übrigen Algorithmen sind einige, die sämtliche statistischen Tests bestehen. Allerdings muss eine gute Kombination für alle drei Anforderungen gefunden werden. Ein Kandidat, der alle Kriterien ausreichend gut erfüllt, ist der Mersenne Twister [MN98] von Matsumoto. Er bietet eine Periodenlänge von $2^{19937} - 1$ und kann 10^8 Zufallszahlen in ca. 2 bis 4 Sekunden generieren. Die Erzeugung von guten Pseudozufallszahlen sorgt für seine weite Verbreitung [vgl. LS07, S. 33]. Aber auch dieser PRNG zeigt Schwächen. Bei der Analyse fällt er bei zwei Tests durch. Allerdings besteht kein PRNG des gleichen Typs diese beiden Tests. Außerdem weist er ein Problem auf, wenn er mit einem Wert initialisiert wird, der zu viele Nullen enthält.

Im Falle solch einer Initialisierung müssen erst bis zu ca. 700.000 Zahlen abgefragt werden, bevor er anfängt gute Pseudozufallszahlen zu liefern. Ein anderer Algorithmus, den dieses Problem weniger stark betrifft, ist der WELL [PLM06; PLM08]. Er fällt in die gleiche Gruppe wie der Mersenne Twister. Allerdings kann er den Zustand einer ungünstigen Initialisierung schon nach ca. 1.000 Zahlen verlassen [vgl. PLM06, S. 11]. Außerdem bietet er verschiedene Periodenlängen von bis zu $2^{44497} - 1$.

Für die Umsetzung des Nichtdeterminismus wird in ReConNet der WELL44497b aus den Apache Commons Bibliotheken [Apa] verwendet (siehe Abbildung 5.2). Dazu hält der PNVF2 die Referenzen zu zwei PRNGs. Der Erste ($DEFAULT\_RANDOM$, siehe Abbildung 5.3) stellt den Standardgenerator aller PNVF2-Instanzen dar. Um einen ggf. schlechten Startzustand zu verlassen, werden bei der Erzeugung des Standardgenerators zuerst 2.000 Zahlen abgefragt. Der zweite PRNG ($RANDOM$) wird beim Matching verwendetet. Falls bei der Initialisierung des PNVF2 kein spezieller PRNG übergeben wird, ist $RANDOM$ gleich dem Standardgenerator.

5.3.3. Knotenordnung

Für die Steuerung des Matchingablaufes muss der PNVF2 Ordnungen über die verschiedenen Stellen- und Transitionenmengen definieren (siehe Unterabschnitt 4.1.1). Außerdem muss ein wahlfreier Zugriff auf einen Knoten anhand seiner Position und umgekehrt auf die Position anhand des Knotens möglich sein. Das ist eine grundlegende Voraussetzung dafür, dass die einzelnen Zustände nicht über verschiedene Matchingpfade erreichbar sind und bei der Prüfung der SSR die Vor- und Nachbereichserhaltung sichergestellt werden kann. Allerdings bietet die Repräsentation der Petrinetze in ReConNet keine Möglichkeit, die Stellen und Transitionen zu ordnen und über ihre Positionen auf die einzelnen Knoten zuzugreifen. Deshalb werden die Ordnungen im PNVF2 umgesetzt. Dazu besitzt die PNVF2-Klasse zusätzliche Instanzvariablen (siehe Abbildung 5.3). Durch die insgesamt vier Knotenmengen werden acht Datenstrukturen benötigt. Die ersten vier Datenstrukturen sind Stellen- und Transitionsarrays ($sourcePlaces$, $sourceTransitions$, ...). Die Position eines Knoten in einem dieser Arrays entspricht seiner Position in der Ordnung. Der Zugriff in umgekehrter Richtung wird mittels HashMaps ($sourcePlacesIndexes$, $sourceTransitionsIndexes$, ...) ermöglicht. Die Kombinationen dieser Arrays und der Maps bilden die vier Stellen- und Transitionsordnungen.

Dieser zusätzliche Aufwand beeinflusst natürlich die Laufzeit- und Speicherkomplexität. Durch die acht Datenstrukturen wird zusätzlicher Speicher im Bereich von $\Theta(p_1+p_2+t_1+t_2)$ benötigt. Allerdings skaliert der Speicherverbrauch schon mit $\Theta(p_1p_2 + t_1t_2)$. Daher

wirken sich die zusätzlichen Datenstrukturen – bis auf eine Erhöhung des konstanten Faktors – nicht auf die Speicherkomplexität aus. Bei der Laufzeitkomplexität sieht der Sachverhalt schon problematischer aus. Der Zugriff auf einen der vier Arrays ist in $\Theta(1)$ möglich, daher fallen die Arrays nicht ins Gewicht. Allerdings weist eine HashMap im Worst-Case eine Zugriffszeit von $\Theta(n)$ auf. Dieses Laufzeitverhalten verbessert sich unter der Annahme des gleichmäßigen Hashings im Mittel zu $\Theta(1)$ [vgl. Cor+10, S. 255]. Da eine Java HashMap ihre Größe bei einer zu starken Befüllung automatisch anpasst und ihre Hashfunktion bei gleichmäßiger Hashverteilung für eine begrenzte Kollisionszahl sorgt [vgl. Has], werden die Zugriffe auf die HashMaps auch mit $\Theta(1)$ angenommen. Die Laufzeitkomplexität bleibt damit ebenso erhalten.

5.3.4. Kandidatengenerierung

Die Kandidatengenerierung erfolgt in der Implementierung des PNVF2 in drei Schritten. Als Erstes wählt der Algorithmus eine der drei speziellen Kandidatenmengen. Zur Wahl stehen Paare aus den out-, in- und den Restknotenmengen. Als nächstes muss sich der Algorithmus für eine der beiden Knotenarten (Stellen oder Transitionen) entscheiden. Der dritte Schritt bildet die eigentliche Berechnung der Kandidaten. Das Vorgehen weicht etwas von der beschriebenen Kandidatengenerierung des PNVF2 (siehe Unterabschnitt 4.1.1) ab. Dort wird angenommen, dass die Kandidaten aller Mengen bekannt sind. Die Berechnung sämtlicher Paare ist allerdings ziemlich ineffizient. Diese Anpassung steht nicht im Widerspruch zur abstrakten Definition, sondern konkretisiert die Berechnungsmethode nur in einigen Punkten. Eine vergleichbare Strategie der Referenzimplementierung [Cor+01b] ist in der Methode $VF2SubState :: NextPair$ nachzulesen.

1. Schritt: Entscheidung für eine Knotenmenge

Damit sich der Algorithmus gezielt für eine Knotenmenge entscheiden kann, wird in zusätzlichen Instanzvariablen der Status der SSR festgehalten (siehe Abbildung 5.3). Der Vorgang soll am Beispiel der Stellen des Quellnetzes verdeutlicht werden.

Dieser Teil der SSR setzt sich aus den Arrays $coreSourcePlaces$, $inSourcePlaces$ und $outSourcePlaces$ zusammen. Damit der PNVF2 nicht vergebens versuchen muss, Kandidaten zu generieren, enthalten die Variablen $coreMatchedPlacesCount$, $inSourcePlacesCount$ und $outSourcePlacesCount$ Informationen über die Belegung der jeweiligen Arrays. Bei dem core-Array entspricht der Wert der Anzahl der bisher abgebildeten Stellen. Für die Terminalarrays haben die Werte eine

andere Bedeutung. Dort sagen sie aus, wie viele Knoten bisher zu dem jeweiligen Terminalarray hinzugefügt worden sind.

$$|T_{P_1}^{out}| = outSourcePlacesCount - coreMatchedPlacesCount$$

Um an die aktuelle Mächtigkeit der jeweiligen Terminalmenge zu gelangen, muss der Algorithmus nur noch die Anzahl der bisher abgebildeten Stellen von der Anzahl der Knoten des entsprechenden Terminalarrays abziehen.

Die Entscheidung für eine spezielle Kandidatenmenge erfolgt genau so, wie sie in der Definition des PNVF2 dargelegt wird (siehe Unterabschnitt 4.1.1). Zuerst berechnet der Algorithmus die Mächtigkeiten der vier out-Terminalmengen. Falls durch die Berechnung festgestellt werden kann, dass für die beiden Knotenarten jeweils wenigstens eine out-Menge keine Knoten enthalten wird, geht der Algorithmus zuerst zu den in-Mengen und danach schließlich den Restknotenmengen über.

2. Schritt: Wahl der Knotenart

Nachdem sich der PNVF2 für eine spezielle Kandidatenmenge entschieden hat, muss noch die abzubildende Knotenart gewählt werden. Die Wahl wird mit der Methode $isMatchPlacePairs$ umgesetzt. Damit auf der aktuellen Rekursionsebene Stellen untersucht werden, muss einer der beiden folgenden Fälle eintreten.

1. Auf dieser Ebene existieren keine Transitionspaare (deterministisch).
2. Die Zufallszahl aus dem Intervall [0, Anzahl Stellenpaare + Anzahl Transitionspaare) ist kleiner als die Anzahl der Stellenpaare (nichtdeterministisch).

3. Schritt: Generierung der Kandidaten

Die eigentliche Berechnung der Kandidaten erfolgt während des Matchingschrittes. Dazu werden auf jeder Rekursionsebene die Knotenindizes der momentan untersuchten Kandidatenknoten gespeichert. Um an diese Indizes zu gelangen, stehen vier Methoden zur Verfügung. Mit der Hilfe von $getFirstFreeSourcePlaceIndex$ oder $getFirstFreeSourceTransitionIndex$ wird zu Beginn eines Matchingschrittes der erste freie Quellknoten gesucht. Danach müssen die Zielknoten identifiziert werden. Die Suche der freien Zielknoten wird in $getNextFreeTargetPlaceIndex$ und $getNextFreeTargetTransitionIndex$ umgesetzt. Damit bei der Kandidatengenerierung nicht die Menge aller Paare im vornherein berechnet werden muss, besitzen die letzten beiden Methoden einen zusätzlichen Parameter. Dieser gibt den Index des aktuell untersuchten Zielknotens an. Dadurch kann die Suche an der

entsprechenden Stelle fortgesetzt werden. Außerdem sorgt diese Vorgehensweise dafür, dass beim Matchingvorgang auf jeder Rekursionsebene nur Speicher für zwei Zahlen benötigt wird.

5.3.5. Abbildbarkeitsmatrizen

Da die Suche eines Matches in verschiedenen Kontexten erfolgen kann, muss sich der PNVF2 auf die unterschiedlichen Anforderungen einstellen können (siehe Unterabschnitt 3.1.2). In der abstrakten Definition des PNVF2 wird allerdings nur das allgemeine Matching behandelt (siehe Unterabschnitt 4.1.2). Damit der Algorithmus die Suche in den verschiedenen Matchingkontexten effizient betreiben kann, weicht die Implementierung in einigen Bereichen von der Definition ab. Die beiden Matrizen werden als zweidimensionale Booleanarrays $semanticMatrixPlaces$ und $semanticMatrixTransitions$ dargestellt. Die Berechnung der Belegung der beiden Arrays wird in den dazugehörigen Methoden $genSemanticMatrixTransitions$ und $genSemanticMatrixPlaces$ durchgeführt (siehe Abbildung 5.3).

Die Werte der Transitionsabbildbarkeitsmatrix entsprechen denen aus der Definition des PNVF2. Zur Reaktion auf den jeweiligen Matchingkontext wird somit nur die Stellenabbildbarkeitsmatrix angepasst. Dazu besitzt die Methode $genSemanticMatrixPlaces$ zwei Parameter. Der erste Parameter zeigt an, ob für diesen Matchingvorgang eine exakte Tokenübereinstimmung gesucht wird. Falls der Wert *true* lautet, wird anstelle der Einhaltung der $\leq$-Beziehung die Gleichheit der Markierungen gefordert. Der andere Parameter dient dem effizienten Matching der Regeln. Damit eine Regel einen Match als gültig ansieht, muss dieser die Kontaktbedingung einhalten (siehe Abschnitt 2.1.3). Da die Transitionen isomorph abgebildet werden, hängt die Einhaltung der Kontaktbedingung nur von dem Matching der Stellen ab. Ein gefundener Match verletzt die Bedingung, wenn eine Stelle durch die Regel gelöscht wird, die außerhalb des Matches noch eine benachbarte Transition besitzt. Da im vornherein bekannt ist, welche Stellen gelöscht werden, kann diese Information genutzt werden, um die Stellenabbildbarkeitsmatrix entsprechend aufzubauen. Deshalb ist der zweite Parameter ein Set von Stellen. Für jede Stelle aus diesem Set erweitert sich Vergleich der semantischen Abbildbarkeit um die Nachbarschaftsbedingung. Diese besagt, dass die Quell- und Zielstellen exakt die gleiche Anzahl an Nachbarn besitzen müssen. Außerdem wird für die jeweiligen Quellstellen der entsprechende Wert in $arcRestrictedSourcePlaces$ auf *true* gesetzt. Mit Hilfe von $arcRestrictedSourcePlaces$ verfeinert $feasible_P$ die Suchraumbeschneidung, indem $rule_{in,P}$, $rule_{out,P}$ und $rule_{new,P}$ ebenfalls die gleiche Anzahl an Nachbarn fordern.

5.3.6. Implementierung von $feasible$

Bei der Implementierung von $feasible$ werden nicht alle Regeln als eigenständige Methoden umgesetzt. Von den zwölf unterschiedlichen Regeln (siehe Unterabschnitt 4.1.2) bleiben nur die vier für die Vorgänger- und Nachfolgeprüfung der Stellen und Transitionen erhalten. Die restlichen acht werden direkt in die beiden $feasible$-Methoden $isFeasiblePlace$ und $isFeasibleTransition$ integriert (siehe Abbildung 5.3).

Um die aufwändige Prüfung der Nachbarschaft zu vermeiden, fragen $isFeasiblePlace$ und $isFeasibleTransition$ zuerst die jeweilige Abbildbarkeitsmatrix ab. Falls ein gegebener Quellknoten semantisch auf den Zielknoten abbildbar ist, werden als nächstes die Vorgänger geprüft. Dies geschieht durch einem Aufruf von $isRulePredPlace$ oder $isRulePredTransition$. Danach betrachtet der Algorithmus in $isRuleSuccPlace$ oder $isRuleSuccTransition$ die Nachfolger. Die anschließende Suchraumbeschneidung wird direkt in den $feasible$-Methoden umgesetzt. Die dazu notwendigen Werte werden während der Prüfung der Vorgänger und Nachfolger gesammelt. Dazu besitzen die Methoden zwei dreielementige Arrays als zusätzliche Parameter. Die Elemente der Arrays stehen für die Anzahl der angrenzenden Knoten aus den out-, in- und Restknotenmengen. Falls eine Methode feststellen kann, dass ein Nachbarknoten noch nicht abgebildet wurde, wird der entsprechende Eintrag im dazugehörigen Array erhöht. Diese Vorgehensweise kann in $VF2SubState :: IsFeasiblePair$ aus [Cor+01b] nachgelesen werden. Der Vorteil besteht darin, dass die Prüfung der Nachbarn nur ein einziges mal erfolgen muss. Da in dieser Implementierung die Ordnungen der Knoten mittels Arrays und HashMaps hergestellt werden, kann somit das wiederholte Abfragen der Datenstrukturen vermieden werden.

Außerdem weist $isFeasiblePlace$ zur Erhaltung der Kontaktbedingung noch eine Besonderheit auf. Falls der Eintrag in $arcRestrictedSourcePlaces$ für die jeweilige Quellstelle $true$ lautet, wird bei der Suchraumbeschneidung anstatt der $\leq$-Beziehung die Gleichheit der Nachbarschaftszahlen gefordert.

5.3.7. Zustandsaktualisierung

Die Zustandsaktualisierung wird beim Hinzufügen eines Paares in den Methoden $addPlacePair$ und $addTransitionPair$ durchgeführt. Falls das untersuchte Paar in einer höheren Rekursionsebene fehlschlägt, wird der entsprechende Backtrackingschritt in $backtrackPlacePair$ und $backtrackTransitionPair$ vollzogen (siehe Abbildung 5.3). Zur Aktualisierung des Zustands besitzt jede der vier Methoden zwei Parameter. Der

Erste ist der Index des Quellknotens und der Zweite der des Zielknotens. Der Index eines Knotens entspricht seiner Position in der dazugehörigen Ordnung. Bei der abstrakten Definition der Zustandsaktualisierung werden nur die jeweiligen core-, in- und out-Arrays aktualisiert. Da in dieser Implementierung zusätzlich der Status der SSR gepflegt wird, muss der PNVF2 die entsprechenden statistischen Werte anpassen. Das soll am Beispiel des Hinzufügens eines Stellenpaares verdeutlicht werden.

Als Erstes zählt der Algorithmus die beiden Werte für die bisher abgebildeten Stellen $coreMatchedPlacesCount$ und Knoten $coreMatchedNodesCount$ um eins nach oben. Die Zahl der bisher abgebildeten Knoten $coreMatchedNodesCount$ entspricht der Tiefe der Rekursion und wird zur Aktualisierung der Terminalarrays verwendet. Danach behandelt der Algorithmus den Sonderfall, bei dem die Stellen nicht aus den Terminalmengen stammen. Dazu werden gegebenenfalls die entsprechenden Werte in den Quellterminalarrays $outSourcePlaces$ und $inSourcePlaces$ sowie den Zielterminalarrays $outTargetPlaces$ und $inTargetPlaces$ gesetzt. Bei der Aktualisierung der Terminalarrays erhöht der PNVF2 außerdem die entsprechenden statistischen Variablen ($outSourcePlacesCount$, $inSourcePlacesCount$, ...) ebenfalls um eins. Der zu einem Terminalarray dazugehörige statistische Wert entspricht der Anzahl der Felder des Arrays, die ungleich null sind. Nachdem zuerst die Anzahl der bisher abgebildeten Knoten erhöht und danach der Sonderfall behandelt wurde, werden die Einträge der core-Arrays $coreSourcePlaces$ und $coreTargetPlaces$ auf die entsprechenden Knotenindizes gesetzt. Abschließend aktualisiert der PNVF2 die Terminalarrays und die dazugehörigen Werte für die Nachbarn der beiden übergebenen Knoten. Damit ist ein Stellenpaar zum Zustand hinzugefügt worden. Beim Backtracking erfolgen die Schritte in umgekehrter Reihenfolge. Diese Vorgehensweise kann in $VF2SubState :: AddPair$ und $VF2SubState :: BackTrack$ aus [Cor+01b] nachgelesen werden.

5.4. MatchVisitor und die negativen Anwendungsbedingungen

Bei einer Regelanwendung sind neben der Einhaltung der Klebebedingungen auch die Erfüllung etwaiger negativer Anwendungsbedingungen (NACs) zu beachten (siehe Unterabschnitt 3.1.2). Das ungezielte Erzeugen von Matches und ihre nachträgliche Prüfung ist allerdings äußerst ineffizient. Außerdem kann ein Aufrufer durch den Nichtdeterminismus nicht erkennen, ob er nach einigen Versuchen schon alle möglichen Matches getroffen hat. Für solche Einsatzzwecke bietet der PNVF2 die Möglichkeit einen eigenen $MatchVisitor$ zu definieren. In der Abbildung 5.4 wird ein Beispielaufbau für einen NAC-Test dargestellt.

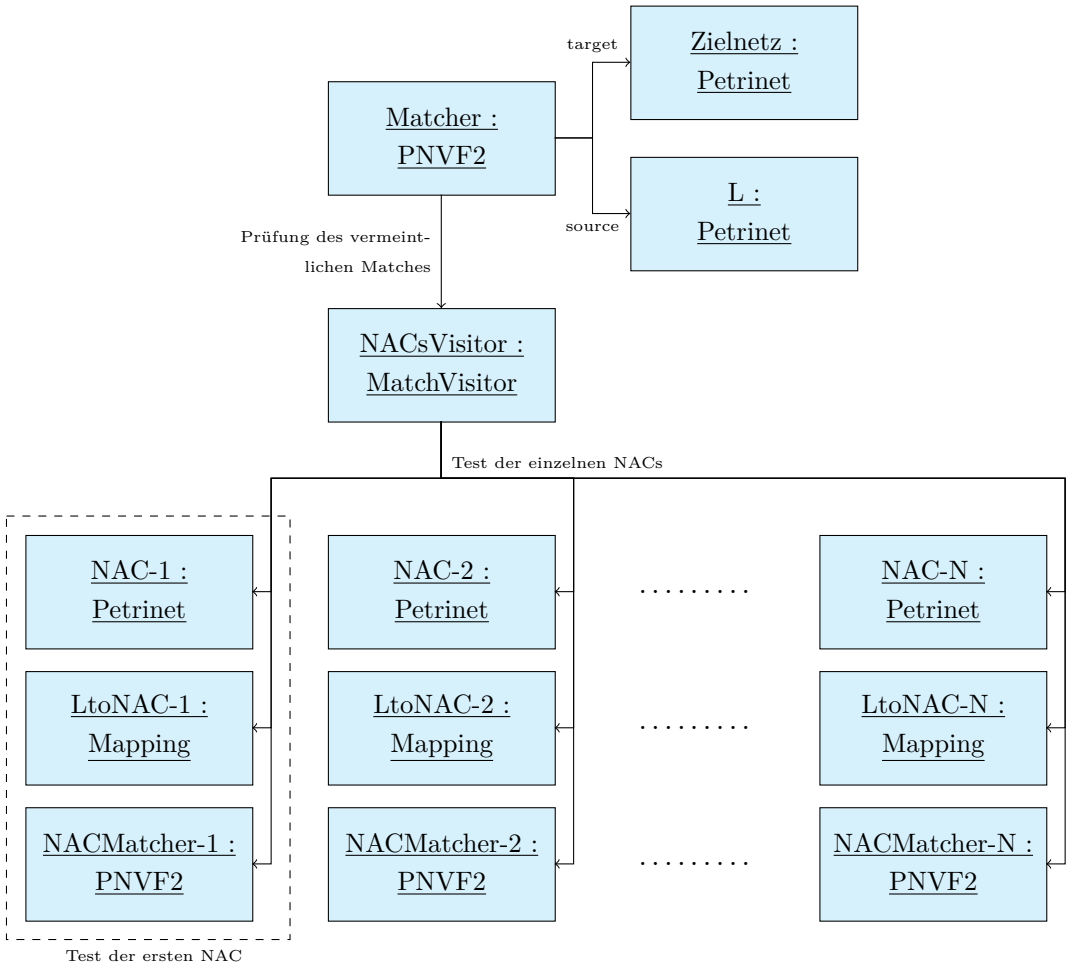


Abbildung 5.4.: Prüfung der negativen Anwendungsbedingungen

Zur Prüfung der NACs muss ein spezieller $MatchVisitor$ erzeugt werden, der zum einen die Menge der NACs und zum anderen die entsprechenden $Mapping$s von L auf die NACs enthält. Die PNVF2-Instanz, die einen Match von L in das Zielnetz sucht, bekommt nun diesen $NACsVisitor$ als $MatchVisitor$ übergeben. Danach wird wie gewohnt der Matchingprozess gestartet. In dem Moment, in dem der $Matcher$ (siehe Abbildung 5.4) einen vermeintlich gültigen Match findet, wird der $NACsVisitor$ zur Prüfung aufgerufen. Dieser nimmt den $Match$ entgegen und testet ihn gegen jede einzelne negative Anwendungsbedingung. Da aus dem $Mapping$ von L auf die jeweilige NAC und dem zu prüfenden Match schon einige Knotenzuordnungen bekannt sind, berechnet der $NACsVisitor$ zuerst den entsprechenden Teilmatch zwischen der NAC und dem Zielnetz. Danach startet er eine neue PNVF2-Instanz und übergibt ihr diesen

Teilmatch. Die neue PNVF2-Instanz versucht auf der Basis des Teilmatches, einen gültigen Match zu berechnen. Falls das für keine NAC gelingt, erfüllt der vermeintliche Match alle negativen Anwendungsbedingungen und ist ein gültiger Match zwischen L und dem Zielnetz. Der Test von anderen höheren Anforderungen kann auf die gleiche Weise umgesetzt werden.

5.5. Fairness der Matchverteilung

Die Simulation benötigt eine Gleichverteilung der Matches zwischen zwei Netzen (siehe Unterabschnitt 1.1.2 und Unterabschnitt 3.1.3). Dazu wurden zwei Strategien vorgestellt. Die Erzeugung aller möglichen Matches ist über einen $MatchVisitor$ zu erreichen. Dazu sammelt dieser sämtliche ihm übergebenen Matches ein, um sie anschließend abzulehnen. Nachdem der PNVF2 mit einer $MatchException$ abbricht, sind alle Matches als Zustand des Visitors erhalten geblieben. Die andere Strategie betrifft die Simulation der Fairness durch den nichtdeterministischen Ablauf des Algorithmus. Dieser wird durch die Permutation der Knoten und die zufällige Wahl der Knotenart auf jeder Rekursionsebene hergestellt (siehe Unterabschnitt 4.1.1). Auf diese Weise werden nicht alle Matches gesucht, sondern nur ein einzelner Match nichtdeterministisch ermittelt, was deutlich effizienter ist. Diese Strategie besitzt allerdings auch Nachteile. Als Beispiel sollen die Netze in Abbildung 5.5 dienen.

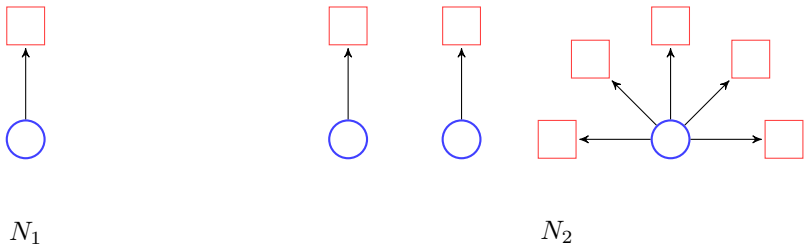


Abbildung 5.5.: Fairness des nichtdeterministischen Ablaufs

In dem Beispiel wird ein Match von N_1 nach N_2 gesucht. Zwischen den beiden Netzen existieren insgesamt sieben verschiedene Matches. Die Simulation erfordert, dass jeder dieser Matches mit einer Wahrscheinlichkeit von $\frac{1}{7}$ getroffen wird. Allerdings wirkt sich die Struktur der Netze stark auf die Wahrscheinlichkeit der einzelnen Matches aus. Falls auf der ersten Rekursionsebene Transitionen abgebildet werden, hat jeder Match eine

Wahrscheinlichkeit von $\frac{1}{7}$. Das ist jedoch nicht mehr der Fall, wenn Stellen untersucht werden. Von den sieben Matches entfallen fünf auf die große Komponente. Da bei diesen fünf Matches die Stellen zusammenfallen, steigt die Wahrscheinlichkeit für einen Match der beiden kleinen Komponenten. Dazu muss man beachten, dass der Algorithmus zuerst jede der Stellen mit einer Wahrscheinlichkeit von $\frac{1}{3}$ trifft. Welche der drei Stellen bei einem bestimmten Matchingschritt verwendet wird, hängt von der Stellenordnung ab. Nachdem die Stellen abgebildet worden sind, müssen die Transitionen zugeordnet werden. Allerdings existieren bei der großen Komponente fünf benachbarte Transitionen. Deshalb werden die einzelnen Matches der großen Komponente weniger oft getroffen als die der kleinen Komponenten.

Für einfache Tests ist der PNVF2 mit seinem Nichtdeterminismus somit problemlos einsetzbar. Allerdings hängt die Verwertbarkeit der Ergebnisse bei einer Simulation von der Struktur der Netze ab.

6. Zusammenfassung, Fazit und Ausblick

Ein Problem beim praktischen Einsatz rekonfigurierbarer Petrinetze ist die effiziente Regelanwendung. Damit eine Regel angewendet werden kann, muss zuerst ein Match zwischen der Regel und dem aktuellen Petrinetz gefunden werden, der sowohl die Klebebedingungen als auch die negativen Anwendungsbedingungen erfüllt. Darüber hinaus werden im Allgemeinen nicht nur einzelne Regeln, sondern eine Regeldatenbank untersucht. Auch die Gleichverteilung der einzelnen Matches muss beachtet werden. Das Ziel dieser Arbeit war es, jede dieser Anforderungen bei der Entwicklung eines speziellen Matchingalgorithmus für rekonfigurierbare Petrinetze zu erfüllen.

6.1. Zusammenfassung

Dazu wurde in Kapitel 3 zuerst ein geeigneter Ausgangsalgorithmus gewählt. Da der Bereich des Matchings von Graphen stetiger Forschung unterliegt, wurden beispielhaft verschiedene Ansätze vorgestellt. Die Wahl fiel schließlich auf den VF2. Der VF2 ist ein rekursiver Baumsuchalgorithmus, wodurch er alle Matches zwischen zwei Graphen aufzählen kann. Beim Matching versucht er die Komponenten des Quellgraphen als Ganzes in den Zielgraphen abzubilden und nutzt zur Beschneidung des Suchraumes die Information über die Nachbarschaft des aktuellen Teilmatches. Außerdem zeichnet er sich durch einen geringen Speicherverbrauch aus.

Anschließend wurde der VF2 in Kapitel 4 zum PNVF2 weiterentwickelt. Da die Petrinetze andere Anforderungen an den Matchingalgorithmus stellen als allgemeine Graphen, wurden die Methoden des VF2 entsprechend angepasst. Eine grundlegende Anforderung an den PNVF2 ist, dass die berechneten Matches korrekt sind. Die Prüfung der Kompatibilität der Knoten wurde in zwei speziellen $feasible$-Methoden umgesetzt (jeweils eine für die Stellen und für die Transitionen). Im Abschnitt 4.3 wurde gezeigt, dass die rekursive Erweiterung eines Teilmatches durch die Hinzunahme eines $feasible$ Paares immer einen korrekten Match zur Folge hat. Eine weitere Anforderung an den Algorithmus ist, dass er nichtdeterministisch abläuft. Zur Erzeugung des Nichtdeterminismus werden die Knoten bei jedem neuen Matchingvorgang permutiert. Außerdem

wird auf jeder Rekursionsebene die abzubildende Knotenart zufällig gewählt. Um die Effizienz der Suche sowohl im erfolglosen Fall als auch bei der Durchsuchung des gesamten Suchraumes zu verbessern, wurden Abbildbarkeitsmatrizen eingeführt. Sie betrachten die semantische Abbildbarkeit der einzelnen Knoten. Mit ihrer Hilfe kann der PNVF2 erkennen, ob zwischen zwei Petrinetzen ein Match möglich ist. Falls das nicht der Fall ist, kann er auf den Matchingversuch verzichten. Die Möglichkeit nicht anwendbare Regeln zu erkennen, ist insbesondere für die effiziente Suche in einer Regeldatenbank notwendig.

Die Beschreibung der Implementierung des PNVF2 erfolgte in Kapitel 5. Dort wurden auch die verschiedenen Anpassungen zur Einhaltung der Klebebedingungen, der negativen Anwendungsbedingungen und die erreichte Fairness besprochen. Zur Einhaltung der Klebebedingungen unterscheidet der PNVF2 zwischen zwei Arten von Stellen. Der Unterschied besteht darin, ob eine Stelle im Zielnetz zusätzliche Nachbarn aufweisen darf oder nicht. Beim Start des Matchingvorganges können die für die Klebebedingung relevanten Stellen übergeben werden. Diese werden danach zur Einschränkung der Abbildbarkeitsmatrizen und damit auch zur Erkennung der nicht anwendbaren Regeln verwendet (siehe Unterabschnitt 5.3.5). Die negativen Anwendungsbedingungen können durch einen speziellen $MatchVisitor$ geprüft werden. Dieser wird während des Matchings aufgerufen und bekommt den aktuellen Match zur Prüfung übergeben. Der PNVF2 verbessert die Effizienz der Prüfung der negativen Anwendungsbedingungen dadurch, dass beim Testen ein Teilmatch vorgegeben werden kann (siehe Abschnitt 5.4). Für die Umsetzung des Nichtdeterminismus wurden außerdem verschiedene Pseudozufallszahlengeneratoren betrachtet und ein geeigneter Generator gewählt. Bei der Auswahl der Generatoren wurden insbesondere ihre statistischen Eigenschaften und ihre Periodenlängen betrachtet (siehe Unterabschnitt 5.3.2). Die Untersuchung der Fairness des PNVF2 erfolgte im Abschnitt 5.5. Es hat sich gezeigt, dass die Fairness des Matchings unter der aktuellen Nichtdeterminismusstrategie von der Struktur der Netze abhängt. Um sicher für eine Gleichverteilung der Matches zu sorgen, muss somit ein $MatchVisitor$ verwendet werden.

6.2. Fazit

Der PNVF2 erfüllt die an ihn gestellten Anforderungen und ist damit für das Matching in rekonfigurierbaren Petrinetzen geeignet. Regeln, die nicht angewendet werden können, kann der PNVF2 erkennen und so das teure Matching vermeiden. Außerdem ist er in

der Lage, bei jedem Matchingversuch einen anderen korrekten Match zu liefern. Bei Einsatzszenarien, die auf eine Gleichverteilung der Matches angewiesen sind, können alle Matches über einen $MatchVisitor$ erzeugt werden. Die Wahl des zu verwendenden Matches kann anschließend nichtdeterministisch stattfinden. Der PNVF2 bildet somit eine solide Basis für ReConNet und für weitere Forschungsansätze.

6.3. Ausblick

Aspekte des Algorithmus, die weitere Aufmerksamkeit bedürfen, sind seine Fairness und sein Nichtdeterminismus. Wie Abschnitt 5.5 zeigt, hängt die Wahrscheinlichkeit der einzelnen Matches von der Struktur der Netze ab. Hier kann gegebenenfalls eine andere Nichtdeterminismusstrategie gesucht werden, die diese Abhängigkeit minimiert. Außerdem muss noch die Vollständigkeit des PNVF2 bewiesen werden.

Durch die Aufstellung der Abbildbarkeitsmatrizen kann der PNVF2 flexibel auf die semantischen Unterschiede der einzelnen Knoten reagieren. Außerdem wird mit ihnen der Suchraum effektiv beschnitten. Bei den Matrizen besteht Entwicklungspotential hinsichtlich ihrer Aufstellung und der Einbeziehung der Nachbarschaft der Knoten.

Ein weiteres Thema bildet das Matching in einer Regeldatenbank. Da der PNVF2 einzelne Petrinetze untersucht, wird seine Laufzeit immer von der Größe der Datenbank abhängen. Hier besteht die Möglichkeit einen effizienten Vorfilteralgorithmus zu entwickeln. Zu guter Letzt können auf der Basis des PNVF2 in ReConNet die negativen Anwendungsbedingungen umgesetzt werden.

A. Anlagen

A.1. VF2-Notation

Zur Vorstellung des VF2 wird in dieser Arbeit eine Untergraphisomorphie von $G_1 = (V_1, E_1)$ nach $G_2 = (V_2, E_2)$ gesucht. In den Papern [Cor+01a; Cor+99; Cor+04] erfolgt die Suche allerdings in umgekehrter Richtung. Das bedeutet, dass die Indizies der Graphen und aller darauf aufbauenden Mengen (z.B. T_1^{out}, M_1 usw.) vertauscht sind. Außerdem wird dort ein Graph als Paar $G_1 = (N_1, B_1)$ definiert, wobei $V_1 = N_1$ und $E_1 = B_1$ ist.

Des Weiteren haben die Regeln zur Strukturerhaltung $feasible(s_n, v, w)$, $rule_{pred}$, $rule_{succ}$, $rule_{out}$, $rule_{in}$ und $rule_{new}$ andere Namen und Definitionen. Die Namen lauten $F(s, n, m)$, R_{pred}, R_{succ}, R_{out}, R_{in} und R_{new}. Damit die Regeln besser auf den PNVF2 erweitert werden können, werden sie in dieser Arbeit in einer abgewandelten Form aufgeführt.

Literatur

[Apa] *Commons Math*. Version: 3. The Apache Software Foundation. URL: http://commons.apache.org/proper/commons-math/.

[Blu13] M. Blumreiter. *Implementierung des PNVF2 in ReConNet*. Revision: 634. Hochschule für Angewandte Wissenschaften Hamburg, 2013. URL: http://jenkins.timadorus.org/job/wppetrinetze/.

[Con+04] D. Conte u. a. "Thirty Years Of Graph Matching In Pattern Recognition". In: *International Journal of Pattern Recognition and Artificial Intelligence* 18.3 (2004), S. 265–298. DOI: 10.1142/S0218001404003228.

[Cor+01a] L. P. Cordella u. a. "An improved algorithm for matching large graphs". In: *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*. 2001, S. 149–159. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.5342.

[Cor+01b] L. P. Cordella u. a. *VFLib Graph Matching Library, version 2.0 (C++)*. Letzter Zugriff: 30. März 2013. Intelligent Systems und Artificial Vision Lab (SIVALab) of the University of Naples "Federico II", 2001. URL: http://mivia.unisa.it/datasets/graph-database/vflib-graph-matching-library-version-2-0/.

[Cor+04] L. P. Cordella u. a. "A (sub)graph isomorphism algorithm for matching large graphs". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26.10 (2004), S. 1367 –1372. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.75.

[Cor+10] Th. H. Cormen u. a. *Algorithmen - eine Einführung*. 3., überarb. und erw. Aufl. München: Oldenbourg-Verl., 2010. ISBN: 978-3-486-59002-9.

[Cor+99] L. P. Cordella u. a. "Performance Evaluation of the VF Graph Matching Algorithm". In: *Image Analysis and Processing, 1999. Proceedings. International Conference on*. 1999, S. 1172 –1177. DOI: 10.1109/ICIAP.1999.797762.

[DZ09] Yin-Tang Dai und Shi-Yong Zhang. “A fast labeled graph matching algorithm based on edge matching and guided by search route”. In: *Wavelet Analysis and Pattern Recognition, 2009. ICWAPR 2009. International Conference on*. 07/2009, S. 1–7. DOI: 10.1109/ICWAPR.2009.5207466.

[EHP09] H. Ehrig, F. Hermann und U. Prange. “Cospan DPO Approach: An Alternative for DPO Graph Transformations”. In: *Bulletin of the EATCS* 98 (2009), S. 139–149.

[EP03] H. Ehrig und J. Padberg. “Graph Grammars and Petri Net Transformations”. In: *Lectures on Concurrency and Petri Nets*. Hrsg. von J. Desel, W. Reisig und G. Rozenberg. Bd. 3098. Lecture Notes in Computer Science. Springer-Verlag, 2003, S. 496–536. ISBN: 3-540-22261-8.

[Ehr+07] H. Ehrig u. a. “Independence of net transformations and token firing in reconfigurable place/transition systems”. In: *Proceedings of the 28th international conference on Applications and theory of Petri nets and other models of concurrency*. ICATPN’07. Siedlce, Poland: Springer-Verlag, 2007, S. 104–123. ISBN: 978-3-540-73093-4.

[Has] *HashMap Dokumentation aus dem Java 7 Collections Framework*. Letzter Zugriff: 30. März 2013. Oracle. URL: http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html.

[IB03] C. Irniger und H. Bunke. “Theoretical analysis and experimental comparison of graph matching algorithms for database filtering”. In: *Proceedings of the 4th IAPR international conference on Graph based representations in pattern recognition*. GbRPR’03. York, UK: Springer-Verlag, 2003, S. 118–129. ISBN: 3-540-40452-X. URL: http://dl.acm.org/citation.cfm?id=1757868.1757883.

[LO04] M. Llorens und J. Oliver. “Structural and dynamic changes in concurrent systems: reconfigurable Petri nets”. In: *Computers, IEEE Transactions on* 53.9 (2004), S. 1147–1158. ISSN: 0018-9340. DOI: 10.1109/TC.2004.66.

[LS07] P. L’Ecuyer und R. Simard. “TestU01: A C library for empirical testing of random number generators”. In: *ACM Trans. Math. Softw.* 33.4 (08/2007). ISSN: 0098-3500. DOI: 10.1145/1268776.1268777. URL: http://doi.acm.org/10.1145/1268776.1268777.

[Luc10] Prof. Dr. K. von Luck. *Living Place Hamburg - A place for concepts of IT based modern living*. Letzter Zugriff: 30. März 2013. Hochschule für Angewandte Wissenschaften Hamburg, 2010. URL: http://livingplace.informatik.haw-hamburg.de/content/LivingPlaceHamburg_en.pdf.

[MB00] B. T. Messmer und H. Bunke. "Efficient subgraph isomorphism detection: a decomposition approach". In: *Knowledge and Data Engineering, IEEE Transactions on* 12.2 (2000), S. 307–323. ISSN: 1041-4347. DOI: 10.1109/69.842269.

[MN98] M. Matsumoto und T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (01/1998), S. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: http://doi.acm.org/10.1145/272991.272995.

[PD08] G. Pomberger und H. Dobler. *Algorithmen und Datenstrukturen*. it informatik. München: Pearson Studium, 2008. ISBN: 978-3-8273-7268-0.

[PLM06] F. Panneton, P. L'Ecuyer und M. Matsumoto. "Improved long-period generators based on linear recurrences modulo 2". In: *ACM Trans. Math. Softw.* 32.1 (03/2006), S. 1–16. ISSN: 0098-3500. DOI: 10.1145/1132973.1132974. URL: http://doi.acm.org/10.1145/1132973.1132974.

[PLM08] F. Panneton, P. L'Ecuyer und M. Matsumoto. *ERRATA for the WELLRNG paper*. Letzter Zugriff: 30. März 2013. 09/2008. URL: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/wellrng-errata.txt.

[Pad+12] J. Padberg u. a. "ReConNet: A Tool for Modeling and Simulating with Reconfigurable Place/Transition Nets." In: *ECEASST* 54 (2012). URL: http://dblp.uni-trier.de/db/journals/eceasst/eceasst54.html#PadbergEOH12.

[Pad12] J. Padberg. "Abstract Interleaving Semantics for Reconfigurable Petri Nets." In: *ECEASST* 51 (2012). URL: http://dblp.uni-trier.de/db/journals/eceasst/eceasst51.html#Padberg12.

[Pet62] C. A. Petri. "Kommunikation mit Automaten". ger. Diss. Universität Hamburg, 1962.

[Pnm] *Petri Net Markup Language*. Letzter Zugriff: 30. März 2013. www.pnml.org. URL: http://www.pnml.org.

[Pra+08] U. Prange u. a. “Transformations in Reconfigurable Place/Transition Systems”. In: *Concurrency, Graphs and Models*. Hrsg. von P. Degano, R. De Nicola und J. Meseguer. Bd. 5065. LNCS. Springer-Verlag, 2008, S. 96–113. ISBN: 978-3-540-68676-7.

[Rei12] F. Reiter. “Modellierung und Analyse von Szenarien des Living Place mit rekonfigurierbaren Petrinetzen”. Bachelor Thesis. HAW Hamburg, 2012. URL: http://opus.haw-hamburg.de/volltexte/2012/1855/pdf/BA_Reiter.pdf.

[SS10] G. Saake und K. Sattler. *Algorithmen und Datenstrukturen: Eine Einführung mit Java*. Hrsg. von G. Saake und K. Sattler. Bd. 4. dpunkt.verlag Heidelberg, 2010.

[Ull76] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *J. ACM* 23.1 (01/1976), S. 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925. URL: http://doi.acm.org/10.1145/321921.321925.

[Wag03] C. Wagenknecht. *Algorithmen und Komplexität*. Informatik interaktiv. München: Fachbuchverlag Leipzig im Carl Hanser Verlag, 2003. ISBN: 3-446-22314-2.

Versicherung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17. April 2013 Mathias Blumreiter