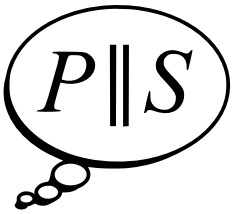
CARL VON OSSIETZKY universität OLDENBURG

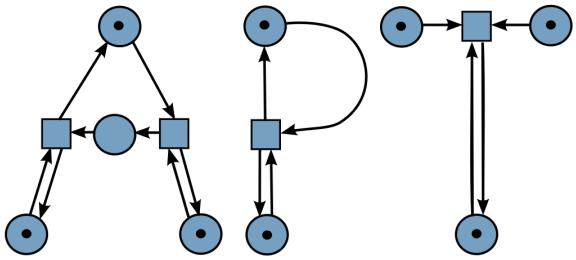
Fakultät II: Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik



Projektgruppe APT

Analyse von Petri-Netzen und Transitionssystemen

Endbericht vom 27. März 2013



Autoren: Björn von der Linde, Chris Schierholz, Daniel Lückehe, Dennis Borde, Maike Schwammberger, Manuel Gieseking, Raffaela Ferrari, Renke Grunwald, Sören Dierkes, Uli Schlachter, V. Spreckels, Vincent Göbel

Betreuer: Prof. Dr. Eike Best, PD Dr. Elke Wilkeit, Dr. Hans Fleischhack, Dipl.-Inform. Thomas Strathmann

Zusammenfassung

Für weiterführende Forschungen im Bereich der Transitionssysteme und Petri-Netze ist es hilfreich, diverse Eigenschaften beliebiger Beispiel-Netze schnell und einfach überprüfen zu können. Da viele vorhandene Analysewerkzeuge in diesem Bereich nicht über die erwünschte Bandbreite an Analyse-Algorithmen verfügen, wurde im Rahmen der Projektgruppe „Analyse von Petri-Netzen und Transitionssystemen“ ein Analysewerkzeug für Petri-Netze mit und ohne Beschriftungen und für beschriftete Transitionssysteme entwickelt.

In diesem Bericht werden Idee, Planung und Implementierung des entwickelten Analysewerkzeugs beschrieben. Zunächst werden der Projektablauf geschildert und einige grundlegende Definitionen aufgelistet. Danach werden die Anforderungen an das System genannt und einige Fremdsysteme auf ihre Nutzbarkeit für das Projekt untersucht, bevor direkt auf die Umsetzung des Systems eingegangen wird. Zwei zentrale Kapitel des Berichts sind die Beschreibung der konkreten Implementierung und der zugrunde liegenden Algorithmen. Zum Schluss wird beschrieben, wie die Benutzungsschnittstelle funktioniert, bevor dieser Bericht mit einer Reflexion über die Vor- und Nachteile des entwickelten Systems und einem Fazit, sowie einem Ausblick darüber, wie eine Weiterentwicklung des Werkzeugs aussehen könnte, endet.

Inhaltsverzeichnis

1 Einleitung

Transitionssysteme und Petri-Netze werden häufig benutzt, um verteilte oder zustandsbasierte Systeme zu modellieren. Um Aussagen über die Struktur jener Systeme zu machen oder sie zu optimieren, ist es hilfreich, ein einfaches, schnelles und umfangreiches Analysewerkzeug zur Hand zu haben. Aus diesem Grund formierte sich im April 2012 die Projektgruppe „Analyse von Petri-Netzen und Transitionssystemen“ (kurz: APT) der Abteilung „Parallele Systeme“ an der Carl von Ossietzky Universität Oldenburg.

Das Ziel der Projektgruppe war es, ein erweiterbares System zur Analyse von Petri-Netzen und Transitionssystemen zu planen und zu implementieren. Dieses Vorhaben war im Zeitraum vom Sommersemester 2012 und Wintersemester 2012/2013 zu bewältigen. Das erste Semester diente dazu, zunächst einige Fremdsysteme zu analysieren, Anforderungen auszuformulieren und grundlegende Projektstrukturen zu planen. Im zweiten Semester begann die Implementierungsphase, in der die gesetzten Anforderungen umgesetzt und durch ausgiebige Testphasen validiert wurden. Letzte Implementierungsarbeiten waren bis Mitte März 2013 zu erledigen. Das vorliegende Dokument ist der Endbericht der Projektgruppe und enthält detaillierte Informationen über Anforderungen, Planungsprozesse und Umsetzung des Projektes.

Als Einstieg in diesen Endbericht wird nach der Einleitung zunächst der Projektablauf vorgestellt, um eine Übersicht darüber zu geben, wie sich die Projektdurchführung gestaltet hat. Anschließend werden Grundlagen zu Petri-Netzen und beschrifteten Transitionssystemen und einige für das Projekt zentrale Eigenschaften definiert. Nach diesem Grundlagen-Kapitel sollen dem Leser im Kapitel „Anforderungen“ zunächst die Anforderungen an das System näher gebracht werden. Hierbei wird als Erstes ein Überblick über die funktionalen Anforderungen an das System inklusive

eines Ein- und Ausgabeformats sowie der optionalen Ausgabe gegeben. Danach werden nicht-funktionale Anforderungen an das System beschrieben, wie beispielsweise qualitätssichernde und prozessbezogene Anforderungen.

Bevor die konkrete Implementierung des Systems erläutert wird, werden einige Fremdsysteme beschrieben, die als erste Arbeitsphase im Projekt auf ihre Verwendbarkeit für das zu implementierende System untersucht wurden. Hierbei wird hervorgehoben, warum ein Fremdsystem in das Projekt eingebunden wurde oder nicht.

Im Kapitel „Implementierung“ wird zunächst die von der Projektgruppe entworfenen Datenstrukturen für Petri-Netze mit und ohne Beschriftungen und für Transitionssysteme beschrieben und es wird erläutert, warum keine in anderen gängigen Analysewerkzeugen benutzten Datenstrukturen gewählt wurden. Weiterhin wird auf ein zentrales Konzept der Implementierung dieses Projektes eingegangen: Das „Modulsystem“. Dieses Modulsystem ermöglicht eine saubere Abkapselung der Analyse-Algorithmen von der Benutzungsschnittstelle. Zudem wird die Struktur des entwickelten Dateiformates für die Ein- und Ausgabe von Petri-Netzen und Transitionssystemen beschrieben. Zum Schluss dieses Kapitels werden verschiedene Parser und Renderer beleuchtet.

Ein weiteres Kapitel dieses Berichts ist das Kapitel „Analyse-Algorithmen“. Hier werden die der Implementierung zugrunde liegenden Algorithmen detailliert beschrieben. Dabei werden folgende Gruppen von Algorithmen unterschieden:

- Graphentheoretische Algorithmen
- Algorithmen für Petri-Netze mit und ohne Beschriftungen
- Algorithmen für beschriftete Transitionssysteme

Nach den Analyse-Algorithmen werden im anschließenden Kapitel der Aufbau und die Anwendung der Benutzungsschnittstelle des Systems erläutert. Im letzten Kapitel wird neben der Einschätzung zu Vor- und Nachteilen des Werkzeugs auch ein Fazit zu den Arbeitsprozessen gegeben und es erfolgt abschließend ein Ausblick darüber, wie das entwickelte System eingesetzt und weiterentwickelt werden kann.

2 Projektablauf

Der Ablauf des Projekts der Gruppe APT unterteilte sich in zwei Phasen. Mit dem Start des Projekts am 01. 04. 2012 wurde die Analysephase eingeleitet. Zu Beginn der Analysephase fand ein Crash-Kurs statt, in welchem alle Projektgruppenteilnehmer ihr Wissen über Petri-Netze und beschriftete Transitionssysteme auffrischen konnten. Parallel zum Crash-Kurs wurden Kleingruppen gebildet, die sich mit Werkzeugen und Arbeiten zur Analyse von Petri-Netzen und beschrifteten Transitionssystemen beschäftigten. In der zweiten Arbeitsphase wurden die im Kapitel 4 genannten Anforderungen umgesetzt.

2.1 Erste Arbeitsphase: Die Analyse

In der Analysephase wurden folgende Analysewerkzeuge für Petri-Netze und Transitionssysteme betrachtet:

- LoLA (Low Level Petri Net Analyzer) und Sara (Structures for Automated Reachability Analysis)
- Synet und Petrify
- PEP (Programming Environment based on Petri Nets)
- Einige studentische Arbeiten zum Thema Petri-Netze und Transitionssysteme

Das Ziel der Analysephase war es, zu evaluieren, welche Arten von Analysewerkzeugen für Petri-Netze und Transitionssysteme bereits existieren, und herauszufinden, ob Fremdsysteme für Teilanforderungen unseres Systems eingebunden werden können.

Die Ergebnisse der Analyse der Fremdsysteme werden im Kapitel 5 „Fremdsysteme“ geschildert.

In dieser ersten Phase wurden weiterhin die Grundlagen für die spätere Umsetzungsphase gelegt und folgende frühe projektorganisatorische Aufgaben verteilt:

Webseite: Die Projektgruppe auf einer Webseite repräsentieren (beispielsweise durch ein Gruppenfoto und Ankündigung der Sitzungstermine).

Versionsverwaltung (git) und Projektmanagement-Tool (Redmine): Ein Projektmanagement-Tool (in diesem Fall: Redmine) aufsetzen und pflegen, in welchem Sitzungsprotokolle und Aufgabenverteilungen organisiert werden, und das git verwalten.

Moderation: Für die Projektgruppensitzungen die Tagesordnung vorschlagen, zu diesen Sitzungen einladen und diese moderieren.

Die Analysephase endete am 24. 07. 2012 damit, dass die Ergebnisse der Kleingruppen mittels Vorträgen vor der Projektgruppe präsentiert wurden. Am selben Tag wurde eine erste Rohfassung der Anforderungen an das Projekt beschlossen, womit die Umsetzungsphase initiiert wurde.

2.2 Zweite Arbeitsphase: Die Umsetzung

In der zweiten Phase des Projekts sollte das APT-Werkzeug geplant und implementiert werden. Hierzu wurden zu Beginn der Arbeitsphase zusätzlich zu den bestehenden projektorganisatorischen Aufgaben weitere vergeben:

Projektmanagement: Hat den Überblick über das Projekt, überprüft und setzt die Deadlines und die Aufgabenverteilung.

Dokumentenbeauftragter: Erstellt eine Vorlage in LaTeX für den Endbericht und ist Ansprechpartner für diese.

Qualitätsmanagement: Überwacht das Einhalten von Code-Konventionen und führt Code-Reviews durch.

Zudem wurden am Anfang der Umsetzungsphase einige projektbezogene Aufgaben vergeben:

Dynamische Module: Entwicklung eines dynamischen Modulsystems, um interne Logikklassen sauber von der Benutzungsschnittstelle abzukapseln

Dateiformat: Entwicklung eines eigenen Dateiformats .apt

Datenstrukturen: Entwicklung einer eigenen Bibliothek in Java für Petri-Netz- und Transitionssystem-Strukturen

Parser: Entwicklung eines Parsers zur Umwandlung diverser Dateiformate für Petri-Netze und Transitionssysteme in andere benötigte Formate

UI: Bereitstellung und Verwaltung einer Benutzungsschnittstelle

Details zu diesen einzelnen Aufgaben sind im Kapitel 6 „Implementierung“ und im Kapitel 8 „Benutzungsschnittstelle“ nachzulesen.

Die Umsetzungsphase gliedert sich in drei zentrale Teile, wobei sich die ersten beiden Teile auf die in Kapitel 4 genannten Anforderungen beziehen:

Teil 1: Umsetzung der funktionalen Anforderungen A bis C

Teil 2: Umsetzung der funktionalen Anforderungen D bis G

Teil 3: Endbericht schreiben

Hierbei ist anzumerken, dass Teil 2 auf Teil 1 aufbaut, so dass zunächst Teil 1 bearbeitet wurde und danach Teil 2. In allen drei Teilen wurden Anforderungen und Aufgaben zu kleinen Arbeitspaketen gebündelt, an denen jeweils zwei bis sechs Personen arbeiteten.

Die Umsetzungsphase endete am 27. 03. 2013 mit Abgabe des Analyse-Tools und des Endberichts.

3 Grundlagen

In diesem Kapitel werden Petri-Netze und Transitionssysteme vorgestellt und einige interessante Eigenschaften definiert. Diese werden für das Verständnis der nächsten Kapitel benötigt. Bis auf gekennzeichnete Ausnahmen sind alle Definitionen aus [Bes12] entnommen. Zur besseren deutschen Übersetzung wurde sich sprachlich an [BW12] orientiert, sofern sich die Definitionen in [Bes12] und [BW12] nicht unterschieden haben.

3.1 Allgemeine Definition eines beschrifteten Transitionssystems

Zunächst werden hier eine Definition eines beschrifteten Transitionssystems sowie die Definition von Feuerbarkeit bzw. Erreichbarkeit gegeben.

Definition 3.1.1 (beschriftetes Transitionssystem) *Ein **beschriftetes Transitionssystem** (engl. labelled transition system, kurz LTS) ist ein Tupel* $(S, \rightarrow, T, s_0)$ *wobei*

- S *die Menge aller Zustände,*
- T *die Menge aller Beschriftungen,*
- $\rightarrow \subseteq (S \times T \times S)$ *die Transitionsrelation (oder auch die Menge aller Kanten)*
- *und* s_0 *der Initialzustand*

sind.

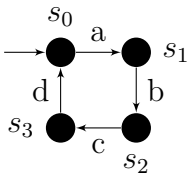


Abbildung 3.1: Beispiel für ein beschriftetes Transitionssystem

Ein beschriftetes Transitionssystem heißt endlich, wenn S und T und damit auch $\rightarrow$ endlich sind. In Abbildung 3.1 findet sich ein Beispiel für ein beschriftetes Transitionssystem.

Definition 3.1.2 *Eine Beschriftung $t \in T$ ist **feuerbar** (aktiviert) in einem Zustand $s \in S$, bezeichnet durch $s[t\rangle$, wenn ein Zustand $s' \in S$ existiert, sodass $(s, t, s') \in \rightarrow$. Außerdem bedeutet die Notation $s[t\rangle s'$, dass s' vom Zustand s durch die Ausführung von t **erreichbar** ist, welches eine alternative Schreibweise von $(s, t, s') \in \rightarrow$ ist.*
Die Definitionen von Feuerbarkeit und von Erreichbarkeit können für Beschriftungssequenzen $\sigma \in T^*$ induktiv erweitert werden:

- *$s[\varepsilon\rangle$ und $s[\varepsilon\rangle s$ gelten immer.*
- *$s[\sigma t\rangle$ ($s[\sigma t\rangle s'$) gelten, wenn es einen Zustand s'' gibt mit $s[\sigma\rangle s''$ und $s''[t\rangle$ ($s''[t\rangle s'$).*

*Ein Zustand s' ist **erreichbar** vom Zustand s, wenn es eine Beschriftungssequenz σ mit $s[\sigma\rangle s'$ gibt.*

Dabei wird $s[\sigma\rangle s'$ **Kreis** genannt, oder genauer ein Kreis an Zustand s, wenn $s = s'$. Außerdem wird mit $[s\rangle$ die Menge von Zuständen bezeichnet, die von Zustand s erreichbar ist.

3.2 Eigenschaften von beschrifteten Transitionssystemen

In diesem Abschnitt werden Eigenschaften beschrieben, die ein beschriftetes Transitionssystem haben kann und die im Analysewerkzeug geprüft werden können.

Definition 3.2.1 (zusammenhängend) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *heißt* ***stark zusammenhängend*** *(engl. strongly connected), wenn von jedem Zustand* $s \in S$ *aus zu jedem anderen Zustand ein gerichteter Weg gefunden wird, d. h. ein Weg, der den Kanten immer nur in Pfeilrichtung folgt.* N *heißt dagegen* ***schwach zusammenhängend*** *(engl. weakly connected), wenn zwischen je zwei Zuständen überhaupt ein Weg gefunden wird (egal ob dabei Kanten in Pfeilrichtung oder dagegen durchlaufen werden, oder sogar beides und abwechselnd).*

Definition 3.2.2 (Parikh-Vektor) *Sei* $N = (S, \rightarrow, T, s_0)$ *ein beschriftetes Transitionssystem. Für eine endliche Feuersequenz* $\sigma = t_1, ..., t_n \in T^*$ *von Beschriftungen ist der* ***Parikh-Vektor*** *(engl. Parikh vector) ein (Spalten-)Vektor* $\Psi(\sigma) : T \rightarrow \mathbb{N}$*, der angibt, wie oft* t *in* σ *vorkommt. Zwei Feuersequenzen* $\tau, \sigma \in T^*$ *heißen* ***Parikh-äquivalent*** *(engl. Parikh-equivalent), wenn* $\Psi(\tau) = \Psi(\sigma)$ *gilt. Ein Kreis ist ein kleinster Kreis, wenn es keinen Kreis mit einem kleineren Parikh-Vektor gibt.*

In Abbildung 3.2 ist ein Beispiel für kleinste Kreise angegeben. Darin ist zum Beispiel $s_0[ac\rangle s_0$ ein kleinster Kreis, während $s_0[abcd\rangle s_0$ kein kleinster Kreis ist.

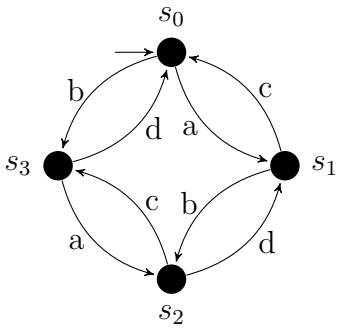


Abbildung 3.2: Beispiel für kleinste Kreise

Definition 3.2.3 (total erreichbar) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *heißt* ***total erreichbar*** *(engl. totally reachable), wenn gilt:* $[s_0\rangle = S$*.*

Definition 3.2.4 (deterministisch) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *heißt* ***deterministisch*** *(engl. deterministic), wenn für alle Zustände* $s \in [s_0\rangle$ *und für alle Transitionen* $t \in T$ *gilt, dass wenn* $s[t\rangle s'$ *und* $s[t\rangle s''$ *gilt, dann folgt daraus* $s' = s''$.

Definition 3.2.5 (reversibel) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *heißt* ***reversibel*** *(engl. reversible), wenn für alle Zustände* $s \in [s_0\rangle$ *gilt:* $s_0 \in [s\rangle$.

Definition 3.2.6 (persistent) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *heißt* ***persistent****, wenn für alle erreichbaren Zustände* s *und alle Beschriftungen* t, u *gilt: Wenn* $s[t\rangle$ *und* $s[u\rangle$ *und* $t \neq u$ *gelten, dann gibt es einen Zustand* $r \in S$*, so dass* $s[tu\rangle r$ *und* $s[ut\rangle r$ *gilt.*

Definition 3.2.7 (Deadlock) *Ein beschriftetes Transitionssystem* $N = (S, \rightarrow, T, s_0)$ *enthält einen* ***Deadlock****, wenn es einen toten Zustand enthält. Ein Zustand* s *ist* ***tot****, wenn in ihm keine Beschriftung* $t \in T$ *feuerbar ist.*

Definition 3.2.8 (Isomorphismus) *Seien* $N_1 = (S_1, \rightarrow_1, T, s_{01})$ *und* $N_2 = (S_2, \rightarrow_2, T, s_{02})$ *zwei beschriftete Transitionssysteme über der gleichen Beschriftungsmenge* T*. Die Transitionssysteme* N_1 *und* N_2 *sind* ***isomorph*** *(engl. isomorphic), wenn es eine Bijektion* β *von* S_1 *nach* S_2 *mit folgenden beiden Eigenschaften gibt:*

1. $\beta(s_{01}) = s_{02}$

2. $(s_1, t, s_2) \in \rightarrow_1$ *ist ein Pfeil in* N_1 *genau dann, wenn* $(\beta(s_1), t, \beta(s_2)) \in \rightarrow_2$ *ein Pfeil in* N_2 *ist für alle* $s_1, s_2 \in S$ *und* $t \in T$.

Die folgende Definition ist [FM90] entnommen, da später mit dieser Definition weitergearbeitet wird.

Definition 3.2.9 (Bisimulation) *Seien* $N_1 = (S_1, \rightarrow, T, s_{01})$ *und* $N_2 = (S_2, \rightarrow, T, s_{02})$ *zwei beschriftete Transitionssysteme über der gleichen Beschriftungsmenge* T*. Die Transitionssysteme* N_1 *und* N_2 *sind* ***bisimilar****, wenn es eine Relation* $\sim \subseteq (S_1 \times S_2)$ *mit folgenden Eigenschaften gibt:*
$\sim = \bigcap \sim^i$ *für* $i \in \mathbb{N}$*, wobei* $\sim^i$ *wie folgt definiert ist:*

- $\forall p \in S_1, \forall q \in S_2 : p \sim^0 q$,

- $p_1 \sim^{i+1} p_2$ *genau dann, wenn für alle* $t \in T$ *gilt:*
  $\forall r_1 \in N_1 : (p_1 \xrightarrow{t} r_1 \Rightarrow \exists r_2 \in N_2 : (p_2 \xrightarrow{t} r_2 \wedge r_1 \sim^i r_2)) \wedge$
  $\forall r_2 \in N_2 : (p_2 \xrightarrow{t} r_2 \Rightarrow \exists r_1 \in N_1 : (p_1 \xrightarrow{t} r_1 \wedge r_1 \sim^i r_2))$

3.3 Allgemeine Definition eines Petri-Netzes

In diesem Abschnitt wird eine Definition eines Petri-Netzes gegeben und es werden Begrifflichkeiten in Bezug zu Petri-Netzen vorgestellt und erläutert.

Definition 3.3.1 (Petri-Netz) *Ein* ***Petri-Netz*** *(engl. Petri net, kurz PN) ist ein Tripel* (S, T, F) *von*

- *einer abzählbaren Menge* S *an Stellen,*
- *einer abzählbaren Menge* T *an Transitionen mit* $S \cap T = \varnothing$
- *und einer Abbildung* $F : (S \times T) \cup (T \times S) \to \mathbb{N}$*, die Kanten zwischen Stellen und Transitionen definiert. Dabei gibt* $F(s,t)$ *an, wie viele Kanten von der Stelle* s *zur Transition* t *führen. Genauso gibt* $F(t,s)$ *an, wie viele Kanten von der Transition* t *zur Stelle* s *führen.*

Normalerweise sind Petri-Netze endlich, das heißt S und T sind endlich. Die Abbildung F kann durch zwei Matrizen $\mathbb{B}, \mathbb{F} \in \mathbb{N}^{|S|\times|T|}$ für das gesamte Petri-Netz dargestellt werden, indem $\mathbb{B}_{i,j}$ die Anzahl $F(s_i, t_j)$ der Kanten von der Stelle s_i zur Transition t_j angibt und $\mathbb{F}_{i,j}$ die Anzahl $F(t_j, s_i)$ der Kanten von der Transition t_j zur Stelle s_i angibt. Daher wird $\mathbb{B}$ auch **Rückwärtsmatrix** (engl. *backward* matrix) und $\mathbb{F}$ **Vorwärtsmatrix** (engl. *forward* matrix) genannt.

In der folgenden Definition wurde die Definition der Mehrfachkante aus [BW12] entnommen, da sie nicht in [Bes12] aufgeführt worden ist, aber im weiteren Verlaufs des Dokumentes benötigt wird.

Definition 3.3.2 *Es sei* $N = (S, T, F)$ *ein Petri-Netz. Für* $x \in S \cup T$ *wird* ${}^\bullet x := \{y \in S \cup T \mid F(y,x) \geq 1\}$ ***Vorbereich*** *von* x*,* $x^\bullet := \{y \in S \cup T \mid F(x,y) \geq 1\}$ ***Nachbereich*** *von* x *genannt. Verallgemeinernd ist für* $X \subseteq S \cup T$ *:* ${}^\bullet X := \bigcup_{x \in X} {}^\bullet x$, $X^\bullet := \bigcup_{x \in X} x^\bullet$ *. Existieren*

*zwischen einer Stelle s und einer Transition t Kanten in beiden Richtungen, d. h. gilt $F(s,t) \geq 1 \leq F(t,s)$, so wird von einer **Schleife** oder von einer **Nebenbedingung** (engl. side condition) gesprochen. Eine Schleife ist **einfach**, wenn gilt: $F(s,t) = 1 = F(t,s)$. Existieren zwischen einer Stelle s und einer Transition t mehr als eine Kante, d. h. gilt $F(s,t) > 1$ oder $F(t,s) > 1$, so wird von einer **Mehrfachkante** gesprochen.*

Definition 3.3.3 (Markierung) *Sei $N = (S,T,F)$ ein Petri-Netz. Die potenzielle Zustandsmenge dieses Petri-Netzes ist $\mathbb{N}^S$, also die Menge der Abbildungen von S nach $\mathbb{N}$. Eine Abbildung $M : S \to \mathbb{N}$ heißt **Markierung** oder Zustand des Petri-Netzes. Gilt $M(s_i) = k$, so bedeutet dies, dass die Stelle s_i genau k Token bzw. Marken trägt. Die Markierung wird nachfolgend als Spaltenvektor geschrieben.*

Die folgende Definition wurde aus [BW12] entnommen, da sie wichtig für das weitere Verständnis ist und nicht in [Bes12] aufgeführt ist.

Definition 3.3.4 (strikt größer) *Seien $k \in \mathbb{N}$ und $M, M' \in \mathbb{Z}^k$ Vektoren über $\mathbb{Z}$, dann ist M **strikt größer** als M' (in Zeichen: $M \gg M'$), falls $M(i) > M'(i)$ für alle $i \in \{1,...,k\}$ gilt. Analog wird **strikt kleiner** definiert.*
M heißt

$$\textbf{\textit{nicht negativ}}, \text{ wenn } M \geq 0 \text{ gilt};$$
$$\textbf{\textit{semipositiv}}, \text{ wenn } M > 0 \text{ gilt};$$
$$\textbf{\textit{positiv}}, \text{ wenn } M \gg 0 \text{ gilt};$$

*dabei ist **0** jeweils der Nullvektor.*

Definition 3.3.5 (Initiales Petri-Netz) *Ein **initiales Petri-Netz** N ist gegeben als $N = (S,T,F,M_0)$, mit*

- *einem Petri-Netz (S,T,F) und*
- *einem initialen Zustand $M_0 \in \mathbb{N}^S$.*

Mit (N, M_0) bezeichnen wir auch das Petri-Netz N mit M_0 als initialem Zustand. Der initiale Zustand wird auch als Anfangsmarkierung oder Startmarkierung bezeichnet.

Abbildung 3.3 zeigt ein Beispiel für ein initiales Petri-Netz.

Definition 3.3.6 (Feuern) *Sei $N = (S, T, F)$ ein Petri-Netz, $M \in \mathbb{N}^S$ ein Zustand von N und t eine Transition von N. Man nennt t* ***M-aktiviert*** *oder feuerbar im Zustand M, falls gilt:*

- $M \geq \mathbb{B}(t)$, *d. h.* $\forall s \in S : M(s) \geq \mathbb{B}_{s,t} = F(s,t)$

Weiter feuert t von Zustand M in den Zustand M', falls gilt:

- $M \geq \mathbb{B}(t)$ *und*
- $M' = M - \mathbb{B}(t) + \mathbb{F}(t)$

Die Kurzform für „t feuert von Zustand M nach Zustand M'“ ist $M[t\rangle M'$.

Definition 3.3.7 *Eine Transition $t \in T$ ist* ***feuerbar*** *von einem Zustand M aus, bezeichnet durch $M[t\rangle$, wenn ein Zustand $M' \in \mathbb{N}^S$ mit $M[t\rangle M'$ existiert. Die Notation $M[t\rangle M'$ bedeutet außerdem, dass M' von einem Zustand M durch das Feuern von t* ***erreichbar*** *ist. Die Definitionen von Feuerbarkeit und Erreichbarkeit können für $\sigma \in T^*$ induktiv erweitert werden:*

- $M[\varepsilon\rangle$ *und* $M[\varepsilon\rangle M$ *gelten immer.*
- $M[\sigma t\rangle$ *($M[\sigma t\rangle M'$) gelten, wenn es einen Zustand M'' gibt mit $M[\sigma\rangle M''$ und $M''[t\rangle$ ($M''[t\rangle M'$).*

σ heißt ***Feuersequenz*** *(engl. firing sequence) oder* ***Schaltfolge*** *(engl. execution sequence) von M aus (oder bei M aktiviert/schaltbar)* $:\Leftrightarrow M[\sigma\rangle$.
Weiter ist $\mathcal{E}(M) := \{M' \mid \exists \sigma \in T^* : M[\sigma\rangle M'\}$ die ***Erreichbarkeitsmenge*** *(engl. reachability set) von M, $\mathcal{E}(N) := \mathcal{E}(M_0)$ ist die Erreichbarkeitsmenge von N. Alternativ schreibt man auch $[M\rangle$ anstatt $\mathcal{E}(M)$.*

Definition 3.3.8 (Inzidenzmatrix) *Sei $N = (S, T, F)$ ein Petri-Netz. Die* ***Inzidenzmatrix*** *(engl. incidence matrix) von N ist die Funktion $C : S \times T \to \mathbb{Z}$ mit $C = \mathbb{F} - \mathbb{B}$.*

Ein Inzidenzmatrixeintrag $C(s,t)$ gibt also gerade an, wie sich die Markenzahl auf der Stelle s durch das Schalten der Transition t verändert (Schaltbilanz).

Definition 3.3.9 (Erreichbarkeitsgraph) *Sei $N = (S, T, F)$ ein Petri-Netz, $M \in \mathbb{N}^S$ ein Zustand von N und $\mathcal{E}(M)$ die Erreichbarkeitsmenge von M in N. Der* ***Erreichbarkeitsgraph*** *$EG(N, M)$ (engl. reachability graph) ist definiert als beschriftetes Transitionssystem $(\mathcal{E}(M), \rightarrow, T, M)$ mit der folgenden Menge an Kanten:*

$$\rightarrow = \{(M_1, t, M_2) \mid M_1 \in \mathcal{E}(M) \wedge M_1[t\rangle M_2\}$$

Wenn $N = (S, T, F, M_0)$ ein initiales Petri-Netz ist, so wird $EG(N) := EG(N, M_0)$ als der Erreichbarkeitsgraph von N bezeichnet.

Der Erreichbarkeitsgraph des in Abbildung 3.3 gezeigten initialen Petri-Netzes findet sich in Abbildung 3.1.

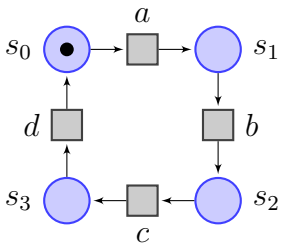


Abbildung 3.3: Ein Beispiel für ein initiales Petri-Netz mit vier Stellen und vier Transitionen. Im initialen Zustand liegt nur auf der Stelle s_0 ein Token.

Für nicht endliche Erreichbarkeitsgraphen gibt es eine Konstruktion, die endlich ist und einen ähnlichen Informationsgehalt hat. Um dies darzustellen, werden generalisierte Markierungen benötigt. Diese können für eine Stelle die „Zahl“ ω beinhalten, die aussagt, dass diese Stelle unbeschränkt viele Marken enthalten kann. Damit sind generalisierte Markierungen Vektoren über $\mathbb{N} \cup \{\omega\}$, welches kurz als $\mathbb{N}_\omega$ geschrieben wird. Für generalisierte Markierungen $M, M' : S \rightarrow \mathbb{N}_\omega$ werden die Definitionen von $\leq$ und $<$ wie folgt erweitert:

$$M \leq M' \Leftrightarrow (\forall s \in S : M(s) \leq M'(s)) \vee (M'(s) = \omega)$$
$$M < M' \Leftrightarrow (M \leq M') \wedge (M \neq M')$$

Der Feuerbegriff für eine Transition t in N ist durch

$$M[t\rangle \Leftrightarrow (\mathbb{B}(t) \leq M)$$
$$M[t\rangle M' \Leftrightarrow (\mathbb{B}(t) \leq M) \wedge (M' = \mathbb{F}(t) - \mathbb{B}(t) + M)$$

ebenfalls auf generalisierte Zustände aus $(\mathbb{N}_\omega)^S$ definiert, wobei

$$n + \omega = \omega = \omega + n, \forall n \in \mathbb{N}_\omega$$

gelte.

Definition 3.3.10 (Überdeckungsgraph) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Der **Überdeckungsgraph** (engl. coverability graph) $Cov(N)$ von N ist definiert als der kantengewichtete Graph $G = (V, E, T)$ (mit Kantenbeschriftungen aus T). Existiert in $Cov(N)$ ein Knoten M und eine Stelle s mit $M(s) = \omega$, so heißt s auch ω-Stelle oder ω-Koordinate von $Cov(N)$.*

Die von der Projektgruppe genutzte Konstruktion des Überdeckungsgraphen wird in Abschnitt 7.3.2 vorgestellt.

Definition 3.3.11 (beschriftetes Petri-Netz) *Ein **beschriftetes Petri-Netz** N (engl. labelled Petri net, kurz LPN) ist ein Tupel $N = (S, T, F, M_0, \Sigma, h_l)$, wobei*

- *(S, T, F, M_0) ein Petri-Netz mit einer initialen Markierung,*
- *Σ ein Alphabet und*
- *$h_l : T \to \Sigma$ eine Beschriftungsabbildung sind.*

Dabei kann h_l zu einem Homomorphismus $h_l : T^* \to \Sigma^*$ erweitert werden:

$$h_l(\varepsilon) := \varepsilon$$
$$h_l(\sigma t) := h_l(\sigma) h_l(t)$$

und weiter für Sprachen $L \subseteq T^*$:

$$h_l(L) = \bigcup_{v \in L} h_l(v)$$

Der **Erreichbarkeitsgraph** eines beschrifteten Petri-Netzes ist genauso definiert wie bei einem Petri-Netz, außer dass jeder seiner Pfeile von der Form (M_1, a, M_2) mit $M_1[t\rangle M_2$ und $h_l(t) = a$ ist.

Definition 3.3.12 (Sprache von Petri-Netzen) *Sei $N = (S, T, F, M_0, \Sigma, h_l)$ ein beschriftetes Petri-Netz.*

$$L(N) := \{w \in \Sigma^* \mid \exists \sigma \in T^* : M_0[\sigma\rangle \wedge w = h_l(\sigma)\}$$

*wird die **Präfix-Sprache** oder nur **Sprache** von N genannt.*

3.3.1 Arten von Petri-Netzen

In diesem Abschnitt werden zwei Arten von Petri-Netzen vorgestellt, die wichtig für die kommenden Anforderungen sind. Sie sind jeweils Petri-Netze mit besonderen Strukturen.

Definition 3.3.13 (n-Bit-Netz) *Sei $n \in \mathbb{N}$ die Anzahl an Bits. Ein initiales Petri-Netz $N = (S, T, F, M_0)$ ist ein **n-Bit-Netz**, wenn es aus n Teilnetzen besteht, wobei das k-te Teilnetz ($1 \leq k \leq n$) folgende Struktur hat:*

- *Es gibt zwei Stellen s_{0k} und s_{1k}.*
- *Es gibt zwei Transitionen set_k und $unset_k$.*
- *Es liegt initial ein Token auf s_{0k}*
- *$F(s_{0k}, set_k) = 1 = F(set_k, s_{1k})$ und $F(s_{1k}, unset_k) = 1 = F(unset_k, s_{0k})$*

Abbildung 3.4 zeigt solch ein n-Bit-Netz mit $n = 4$.

Edsger W. Dijkstra erfand das Beispiel der fünf Philosophen [Dij71], um Fragen der Deadlockfreiheit und der Fairness in verteilten Systemen zu studieren. Dabei sitzen

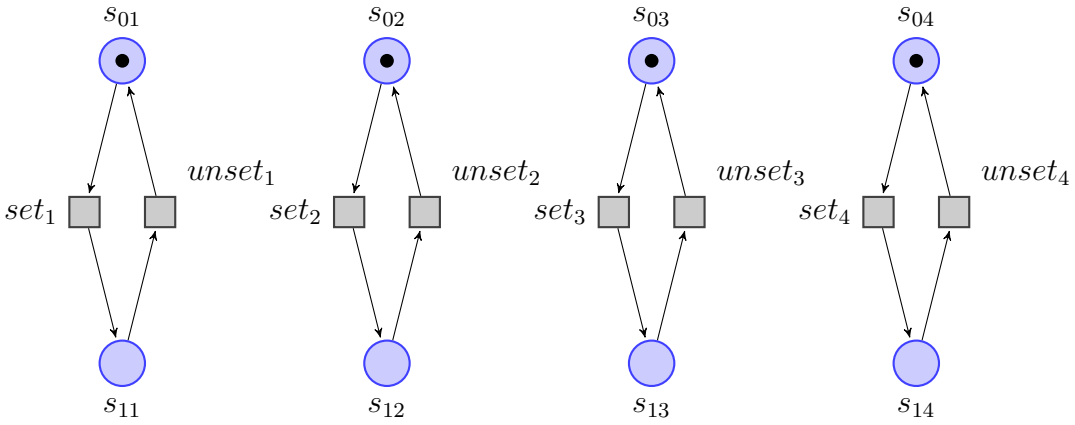


Abbildung 3.4: n-Bit-Netz mit $n = 4$

fünf Philosophen an einem Tisch mit jeweils einer Portion Spaghetti vor sich. Zwischen jeweils zwei nebeneinander sitzenden Philosophen liegt genau eine Gabel, also gibt es insgesamt fünf Gabeln. Die Philosophen benötigen zum Essen ihrer Spaghetti jeweils die zwei Gabeln, die rechts und links neben ihnen liegen, wodurch nicht alle Philosophen gleichzeitig essen können. Diese Situation kann als Petri-Netz abgebildet und wegen der Regelmäßigkeit für beliebig viele Philosophen mit der entsprechenden Anzahl an Gabeln verallgemeinert werden. Daraus ergibt sich folgende Definition eines Philosophen-Petri-Netzes:

Definition 3.3.14 (Philosophen-Petri-Netz) *Sei k die Anzahl der Philosophen und damit auch die der Gabeln. Ein Petri-Netz $N = (S, T, F, M_0)$ bildet das Philosophen-Problem ab, wenn folgende Eigenschaften gegeben sind:*

- *Die Gabeln werden durch k Stellen g_i mit $0 \leq i < k$ abgebildet.*
- *Initial liegen auf den Stellen g_i mit $0 \leq i < k$ jeweils genau ein Token, die eine auf dem Tisch liegende Gabel repräsentieren.*
- *Die k Philosophen werden jeweils durch drei Stellen e_i, t_i und w_i mit $0 \leq i < k$ modelliert, die die Zustände denkend (t_i, thinking), wartend (w_i, waiting) oder essend (e_i, eating) repräsentieren. In welchem Zustand sich die einzelnen Philosophen befinden, wird durch ein Token auf der entsprechenden Stelle angezeigt.*
- *Initial befinden sich die Philosophen im Zustand denkend, weshalb auf den Stellen t_i mit $0 \leq i < k$ jeweils ein Token liegt.*
- *Die Transitionen $nimm_i$, iss_i und $satt_i$ mit $0 \leq i < k$ können den Zustand des*

Philosophens i durch Schalten verändern.

- *Die Transition $nimm_i$ kann schalten, wenn sich jeweils Token auf t_i und g_i befinden. Es wird dann ein Token auf Stelle w_i gelegt. Ein denkender Philosoph kann also seine linke Gabel nehmen, um den Zustand wartend zu erreichen.*
- *Die Transition iss_i kann schalten, wenn sich jeweils Token auf w_i und $g_{(i+k-1) \bmod k}$ befinden. Es wird dann ein Token auf Stelle e_i gelegt. Wenn also die rechte Gabel verfügbar ist, kann ein wartender Philosoph diese nehmen und den Zustand essend erreichen.*
- *Die Transition $satt_i$ kann schalten, wenn sich ein Token auf e_i befindet. Es wird dann jeweils ein Token auf die Stellen t_i, g_i und $g_{(i+k-1) \bmod k}$ gelegt. Um also wieder zum Zustand denkend zurückzukehren, legt ein essender Philosoph beide Gabeln zurück.*

Abbildung 3.5 zeigt solch ein Petri-Netz für drei Philosophen.

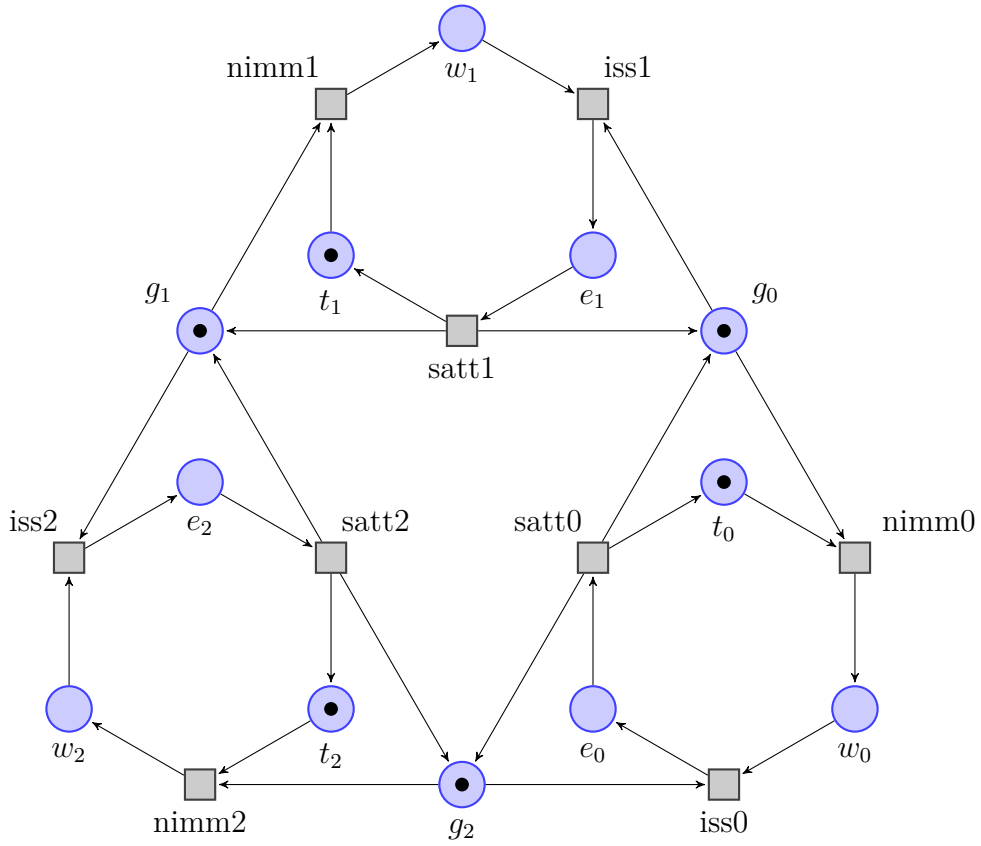


Abbildung 3.5: Philosophen-Petri-Netz mit drei Philosophen

3.4 Eigenschaften von Petri-Netzen

In diesem Abschnitt werden Eigenschaften beschrieben, die ein Petri-Netz haben kann und die im Analysewerkzeug analysiert werden können.

Definition 3.4.1 (schlicht) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***schlicht*** *(engl. plain), wenn F den Wertebereich $\{0, 1\}$ hat, also Mehrfachkanten verboten sind.*

Definition 3.4.2 (pur) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***pur*** *(engl. pure), wenn N keine Schleifen besitzt.*

Definition 3.4.3 (isoliert) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt ein Element $x \in S \cup T$* ***isoliert*** *(engl. isolated), wenn ${}^\bullet x \cup x^\bullet = \varnothing$ gilt.*

Die folgende Definition wurde schon für beschriftete Transitionssysteme eingeführt. Sie ist vergleichbar mit dieser, aber hier auf Petri-Netze zugeschnitten.

Definition 3.4.4 (reversibel) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Eine Transition $t \in T$ heißt* ***reversibel*** *(engl. reversible), wenn für jede erreichbare Markierung $M \in \mathcal{E}(M_0)$ gilt: Wenn $M[t\rangle M'$ dann gilt $M \in \mathcal{E}(M')$. N heißt reversibel, wenn jede Transition reversibel ist.*

Definition 3.4.5 (beschränkt) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Dann heißt eine Stelle $s \in S$* ***m-beschränkt*** *(engl. m-bounded) für ein $m \in \mathbb{N}$, falls für alle Feuersequenzen $\sigma \in T^*$ und Zustände M aus $M_0[\sigma\rangle M$ stets $M(s) \leq m$ folgt. Die Stelle s heißt* ***beschränkt*** *(engl. bounded), wenn sie m-beschränkt für irgendein $m \in \mathbb{N}$ ist. N heißt beschränkt, wenn alle Stellen $s \in S$ beschränkt sind. N heißt m-beschränkt für ein $m \in \mathbb{N}$, wenn jede Stelle $s \in S$ m-beschränkt ist.*

Definition 3.4.6 (sicher) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Dann heißt eine Stelle $s \in S$* ***sicher*** *(engl. safe), wenn sie 1-beschränkt ist. N heißt sicher, wenn alle Stellen sicher sind.*

Definition 3.4.7 (lebendig) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz.*

- *Eine Transition $t \in T$ heißt* ***einfach lebendig*** *(engl. simply live), wenn es eine Feuersequenz $\sigma \in T^*$ gibt mit $M_0[\sigma t\rangle$.*

- *Eine Transition $t \in T$ heißt **tot** (engl. dead), wenn sie nicht einfach lebendig ist.*
- *Die Transition t wird **schwach lebendig** (engl. weakly live) genannt, wenn es ein unendliches Wort w gibt, in dem t unendlich oft vorkommt und für jeden (endlichen) Präfix σ von w gilt: $M_0[\sigma\rangle$.*
- *Die Transition t wird **lebendig** (engl. live) oder **stark lebendig** (engl. strongly live) genannt, wenn für jeden erreichbaren Zustand $M \in \mathcal{E}(M_0)$ gilt : Es gibt eine Feuersequenz σ mit $M[\sigma t\rangle$.*

N heißt (einfach/schwach/stark) lebendig, wenn jede Transition (einfach/schwach/-stark) lebendig ist. N heißt tot, wenn N keine einfach lebendige Transition besitzt.

Definition 3.4.8 (sprachäquivalent) *Seien $N = (S, T, F, M_0, \Sigma, h_l)$ und $N' = (S', T', F', M_0', \Sigma, h_l)$ zwei beschriftete Petri-Netze. N und N' heißen **sprachäquivalent** (engl. language-equivalent), wenn $L(N) = L(N')$ gilt.*

Definition 3.4.9 (S-Invariante) *Ein Vektor $x \in \mathbb{Z}^{|S|}$ heißt **S-Invariante** (engl. S-invariant), wenn $C^T \cdot x = 0$ gilt.*
*Eine semipositive S-Invariante x ist **minimal**, wenn keine S-Invariante x' mit $0 < x' < x$ existiert.*

Definition 3.4.10 (T-Invariante) *Ein Vektor $y \in \mathbb{Z}^{|T|}$ heißt **T-Invariante** (engl. T-invariant), wenn $C \cdot y = 0$ gilt.*
*Eine semipositive T-Invariante y ist **minimal**, wenn keine T-Invariante y' mit $0 < y' < y$ existiert.*

Die folgende Definition wurde schon für beschriftete Transitionssysteme eingeführt. Sie ist vergleichbar mit dieser, aber hier auf Petri-Netze zugeschnitten.

Definition 3.4.11 (zusammenhängend) *Sei $N = (S, T, F)$ ein Petri-Netz. N heißt **stark zusammenhängend** (engl. strongly connected), wenn von jedem Knoten (Stelle/Transition) aus zu jedem anderen Knoten ein gerichteter Weg gefunden werden kann, d. h. ein Weg, der den Kanten immer nur in Pfeilrichtung folgt. N heißt dagegen **schwach zusammenhängend** (engl. weakly connected), wenn man zwischen je zwei Knoten überhaupt einen Weg findet (egal ob dabei Kanten in Pfeilrichtung oder dagegen durchlaufen werden, oder sogar beides und abwechselnd).*

Definition 3.4.12 (S-Netz) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***S-Netz*** *(engl. Snet), wenn für alle Transitionen $t \in T$ gilt:$|\, {}^\bullet t \,| \leq 1 \geq |\, t^\bullet \,|$*

Definition 3.4.13 (T-Netz) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***T-Netz*** *(engl. T-net), wenn für alle Stellen $s \in S$ gilt: $|\, {}^\bullet s \,| \leq 1 \geq |\, s^\bullet \,|$*

Definition 3.4.14 (free-choice) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***FC-Netz*** *(engl. FC-net), falls N*

$$\forall t_1, t_2 \in T : {}^\bullet t_1 \cap {}^\bullet t_2 \neq \varnothing \implies {}^\bullet t_1 = {}^\bullet t_2$$

erfüllt. Ein Petri-Netz $N' = (S, T, F, M_0)$ mit Anfangsmarkierung M_0 heißt ***FC-System****, falls (S, T, F) ein FC-Netz ist.*

Die folgende Definition wurde aus [B+12a] entnommen. Dort ist sie unter dem Name *fc-net* in Anforderung B8d zu finden, weshalb sie zur besseren Abgrenzung von *FC-Netz* nach Absprache in *restricted free choice* umbenannt wurde.

Definition 3.4.15 (restricted free-choice) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***restricted free-choice****, falls N*

$$\forall t_1, t_2 \in T : {}^\bullet t_1 \cap {}^\bullet t_2 \neq \varnothing \implies |\, {}^\bullet t_1 \,| = |\, {}^\bullet t_2 \,| = 1$$

erfüllt.

Definition 3.4.16 (Fallen and Siphons) *Sei $N = (S, T, F)$ ein Petri-Netz.*
Eine Menge $D \subseteq S$ heißt ***Siphon****, wenn ${}^\bullet D \subseteq D^\bullet$.*
Eine Menge $Q \subseteq S$ heißt ***Falle*** *(engl. trap), wenn $Q^\bullet \subseteq {}^\bullet Q$.*

Definition 3.4.17 (output-nonbranching) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***output-nonbranching****, falls N*

$$\forall s \in S : |s^\bullet| \leq 1$$

erfüllt.

Definition 3.4.18 (konfliktfrei) *Sei $N = (S, T, F)$ ein Petri-Netz, dann heißt N* ***konfliktfrei*** *(engl. conflict-free), falls N*

$$\forall s \in S : |s^\bullet| > 1 \implies s^\bullet \subseteq {}^\bullet s$$

erfüllt.

Definition 3.4.19 (BCF) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. N heißt **behaviourally conflict-free** (kurz BCF), wenn für je zwei Transitionen $t, t' \in T$ mit $t \neq t'$ und für jede Markierung $M \in \mathcal{E}(M_0)$ gilt: Wenn $M[t\rangle$ und $M[t'\rangle$ gilt, dann ist ${}^\bullet t \cap {}^\bullet t' = \varnothing$.*

Definition 3.4.20 (BiCF) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. N heißt **binary conflict-free** (kurz BiCF), wenn für je zwei Transitionen $t, t' \in T$ mit $t \neq t'$ und für jede Markierung $M \in \mathcal{E}(M_0)$ gilt: Wenn $M[t\rangle$ und $M[t'\rangle$ gilt, dann ist $\forall s \in S : M(s) \geq F(s,t) + F(s,t')$.*

Die folgende Definition wurde schon für beschriftete Transitionssysteme eingeführt. Sie ist vergleichbar mit dieser, aber hier auf Petri-Netze zugeschnitten.

Definition 3.4.21 (persistent) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Eine Transition $t \in T$ heißt **persistent**, wenn für jede erreichbare Markierung $M \in \mathcal{E}(M_0)$ und für jede Transition $t' \in T$ mit $t \neq t'$ gilt: Wenn $M[t\rangle$ und $M[t'\rangle$ gilt, dann gilt auch $M[tt'\rangle$ und $M[t't\rangle$. N heißt persistent, wenn jede Transition persistent ist.*

Definition 3.4.22 (k-Markierung) *Sei $N = (S, T, F, M_0)$ ein Petri-Netz. Eine Markierung M ist eine **k-Markierung** (engl. k-marking), wenn es eine Markierung M' gibt, so dass $M(s) = k \cdot M'(s)$ für jede Stelle $s \in S$ gilt. N ist ein **k-Netz** (engl. k-net), wenn M_0 eine k-Markierung ist.*

In der folgenden Definition wurde die Definition des Shuffle-Operators aus [PW98] entnommen.

Definition 3.4.23 (Starke *k*-Separierbarkeit) *Sei $N = (S, T, F, k \cdot M_0)$ – nicht $N = (S, T, F, M_0)$ – ein Petri-Netz mit initialer k-Markierung, dann heißt N **stark k-separierbar** (engl. strongly k-separable), wenn für jede feuerbare Sequenz $(k \cdot M_0)[\sigma\rangle$ es $\sigma_1, ..., \sigma_k$ gibt, so dass gilt:*

$$\forall j, 1 \leq j \leq k : M_0[\sigma_j\rangle \text{ und } \sigma \in \sqcup\!\sqcup_{j=1}^{k} \sigma_j$$

*Der **Shuffle Operator** $\sqcup\!\sqcup$ bildet zwei Wörter (hier: Sequenzen) in eine Liste von Wörtern ab, die alle möglichen Verschachtelungen der beiden Wörter beinhaltet.*

Definition 3.4.24 (Schwache *k*-Separierbarkeit) *Sei* $N = (S, T, F, k{\cdot}M_0)$ *– nicht* $N = (S, T, F, M_0)$ *– ein Petri-Netz mit initialer k-Markierung, dann heißt N* ***schwach k-separierbar*** *(engl. weakly k-separable), wenn für jede feuerbare Sequenz* $(k{\cdot}M_0)[\sigma\rangle$ *es* $\sigma_1, ..., \sigma_k$ *gibt, sodass gilt:*

$$\forall j, 1 \leq j \leq k : M_0[\sigma_j\rangle \text{ und } \Psi(\sigma) = \sum_{j=1}^{k} \Psi(\sigma_j)$$

In Abbildung 3.6 ist ein größeres Petri-Netz dargestellt, welches einige aufgeführte Eigenschaften enthält. Es wird außerdem als Testnetz für das später in Abschnitt 7.3.11 erläuterte Modul genutzt.

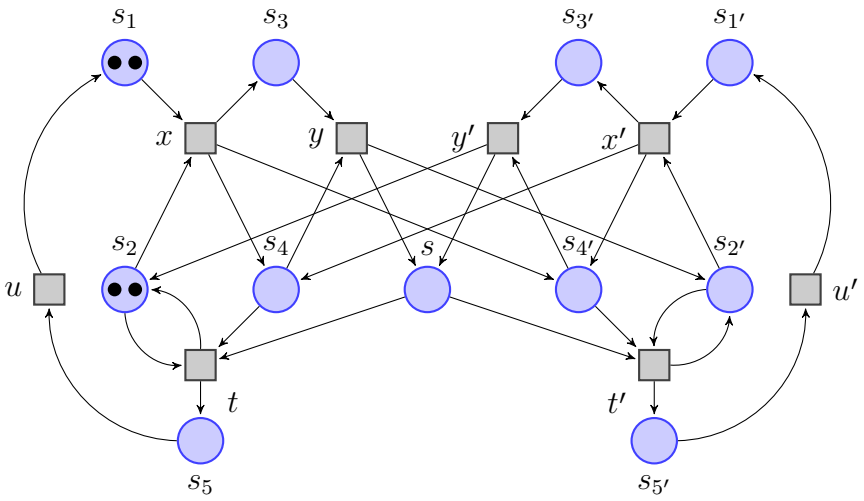


Abbildung 3.6: Ein beschränktes, schlichtes, reversibles und persistentes 2-Netz [BD11]

4 Anforderungen

In diesem Abschnitt werden die funktionalen und nichtfunktionalen Anforderungen aufgeführt, die richtungsweisend für die gesamte Arbeit der Projektgruppe waren.

4.1 Funktionale Anforderungen

Die folgenden funktionalen Anforderungen sind maßgeblich für den Funktionsumfang des entwickelten Analysewerkzeugs. Sie stammen aus [B+12a] und [B+12b] und stellen die Spezifikation des Analysewerkzeugs dar. Sie wurden nur dahingehend abgewandelt, dass die eigentliche funktionale Anforderung ins Deutsche übersetzt worden ist. Abweichungen von diesen beiden Dokumenten sind als solche gekennzeichnet.

Die funktionalen Anforderungen geben an, auf welche Eigenschaften beschriftete Transitionssysteme, Petri-Netze und beschriftete Petri-Netze getestet werden können sollen und was bei Erfüllung bzw. Nichterfüllung einer solchen Eigenschaft ausgegeben werden soll. Außerdem sollen noch bestimmte Elemente in Transitionssystemen und in Petri-Netzen mit und ohne Beschriftungen berechnet werden können. Sowohl für die Eigenschaften als auch für die Berechnungen wird außerdem angegeben, welche Vorbedingungen überhaupt gelten müssen, damit eine Prüfung der Eigenschaft oder eine Berechnung durchgeführt werden kann.

Gegeben: Ein endliches beschriftetes Transitionssystem

A1: Es soll die syntaktische Korrektheit überprüft werden können.

Eingabe: LTS oder LTS mit Anfangszustand s_0
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall soll angezeigt werden, was inkorrekt ist.

In [B+12a] wurde angemerkt, dass die vorige Zeile fast universell auch für alle anderen Anforderungen gilt, dabei in der Regel aber nicht alle Gegenbeispiele angezeigt werden müssen.

A2: Es soll die *deterministisch*-Eigenschaft überprüft werden können.
Eingabe: LTS, Anfangszustand s_0
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall soll ein Folgezustand und ein Beschriftungen angegeben werden, die den Determinismus verletzen.

A3: Es soll die totale Erreichbarkeit überprüft werden können.
Eingabe: LTS, Anfangszustand s_0
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall soll ein Zustand angegeben werden, der von s_0 aus nicht erreichbar ist.

A4: Es soll die *persistent*-Eigenschaft überprüft werden können.
Eingabe: LTS, Anfangszustand s_0
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall soll ein Folgezustand und zwei Beschriftungen angegeben werden, die die Persistenz verletzen.

A5: Es soll die *reversibel*-Eigenschaft überprüft werden können.
Eingabe: LTS, Anfangszustand s_0
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall soll ein Folgezustand angegeben werden, von dem aus s_0 nicht erreichbar ist.

A6: Es soll überprüft werden können, ob ein LTS auch von einem Petri-Netz generiert werden kann.
Eingabe: LTS, Anfangszustand s_0

Ausgabe: Im „Ja“-Fall soll ein (unbeschriftetes) Petri-Netz ausgegeben werden, dessen Erreichbarkeitsgraph isomorph zum LTS ist.
Im „Nein“-Fall soll eine Fehlermeldung und 2 Zustände bzw. Zustand/Beschriftung ausgegeben werden, die die Separation verletzen.

A7: Es soll überprüft werden können, ob ein LTS auch von einem verteilten Petri-Netz generiert werden kann.
Eingabe: LTS, Anfangszustand s_0, eine Zuordnung von Transitionen t auf Locations $l(t)$.
Ausgabe: Im „Nein“-Fall soll eine Ausgabe eines unbeschrifteten Netzes erfolgen, dessen Erreichbarkeitsgraph isomorph zum LTS ist und der die Eigenschaft hat: Falls $l(t) = l(u)$ und $t \neq u$, dann haben t und u keine gemeinsame Vorstelle, d. h. ${}^\bullet t \cap {}^\bullet u = \varnothing$.
Im „Nein“-Fall sollen eine Fehlermeldung und 2 Zustände bzw. Zustand/Beschriftung ausgegeben werden, die die Separation verletzen.

A8: Es soll überprüft werden können, ob alle kleinsten Kreise den gleichen Parikh-Vektor haben.
Eingabe: LTS
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall sollen zwei Gegenbeispiel-Kreise ausgegeben werden.

A9: Es soll überprüft werden können, ob alle kleinsten Kreise den gleichen oder wechselseitig disjunkten Parikh-Vektor haben.
Eingabe: LTS
Ausgabe: Ja/Nein
Optional: Im „Nein“-Fall sollen zwei Gegenbeispiel-Kreise ausgegeben werden.

A10: Es sollen die (schwach) zusammenhängenden Komponenten berechnet werden können.
Eingabe: LTS
Ausgabe: Liste der Zusammenhangskomponenten

A11: Es sollen die stark zusammenhängenden Komponenten berechnet werden können.
Eingabe: LTS

Ausgabe: Liste der starken Zusammenhangskomponenten

A12: Es sollen die Parikh-Vektoren der kleinsten Kreise berechnet werden können.
Eingabe: LTS
Ausgabe: Liste der Parikh-Vektoren kleinster Kreise und für jeden Vektor einen Beispielkreis

Gegeben: Ein Petri-Netz (möglicherweise mit Kantengewichten und Nebenbedingungen)

B1: Es soll die syntaktische Korrektheit überprüft werden können.
Eingabe: PN oder PN mit Anfangsmarkierung M_0
Ausgabe: Ja/Nein

B2: Es soll *schlicht*-Eigenschaft überprüft werden können.
Eingabe: PN
Ausgabe: Ja/Nein

B3: Es soll *pur*-Eigenschaft überprüft werden können.
Eingabe: PN
Ausgabe: Ja/Nein

B4: Es soll die Eigenschaft *nicht pur aber nur mit einfachen Nebenbedingungen* überprüft werden können.
Eingabe: PN
Ausgabe: Ja/Nein

B5: Es soll überprüft werden können, ob es isolierte Elemente gibt.
Eingabe: PN
Ausgabe: Ja/Nein

B6: Es soll überprüft werden können, ob das Petri-Netz (schwach) zusammenhängend ist.
Eingabe: PN

Ausgabe: Ja/Nein

B7: Es soll überprüft werden können, ob das Petri-Netz stark zusammenhängend ist.

Eingabe: PN
Ausgabe: Ja/Nein

B8: Es soll überprüft werden können, ob das Petri-Netz mit semipositiven S-Invarianten überdeckt ist.
Eingabe: PN
Ausgabe: Ja/Nein

B9: Es soll überprüft werden können, ob das Petri-Netz mit semipositiven T-Invarianten überdeckt ist.
Eingabe: PN
Ausgabe: Ja/Nein

B10: Es soll überprüft werden können, ob das Petri-Netz ein T-Netz ist.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein
Optional: Es soll ein Fehler ausgegeben werden, wenn die Eingabe nicht schlicht ist.

B11: Es soll überprüft werden können, ob das Petri-Netz ein S-Netz ist.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein
Optional: Es soll ein Fehler ausgegeben werden, wenn die Eingabe nicht schlicht ist.

B12: Es soll überprüft werden können, ob das Petri-Netz ein FC-Netz ist.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein
Optional: Es soll ein Fehler ausgegeben werden, wenn die Eingabe nicht schlicht ist.

B13: Es soll überprüft werden können, ob das Petri-Netz restricted free choice ist.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein
Optional: Es soll ein Fehler ausgegeben werden, wenn die Eingabe nicht schlicht ist.

B14: Es soll die *ON*-Eigenschaft überprüft werden können.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein

B15: Es soll die *CF*-Eigenschaft überprüft werden können.
Eingabe: Schlichtes PN
Ausgabe: Ja/Nein

B16: Es soll die *beschränkt*-Eigenschaft überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0
Ausgabe: Ja/Nein
Im „Nein“-Fall soll eine Stelle angegeben werden, die unbeschränkt ist.

B17: Es soll die *sicher*-Eigenschaft überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0
Ausgabe: Ja/Nein
Im „Nein“-Fall soll eine Stelle angeben werden, die nicht sicher ist.

B18: Es soll BCF überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt
Ausgabe: Ja/Nein
Im „Nein“-Fall soll eine Folgemarkierung und 2 Transitionen angegeben werden, die BCF verletzen.

B19: Es soll BiCF überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt
Ausgabe: Ja/Nein
Im „Nein“-Fall soll eine Folgemarkierung und 2 Transitionen angegeben werden, die BiCF verletzen.

B20: Es soll die *persistent*-Eigenschaft überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt
Ausgabe: Ja/Nein
Im „Nein“-Fall soll eine Folgemarkierung und 2 Transitionen angegeben werden, die Persistenz verletzen.

B21: Es soll die *reversibel*-Eigenschaft überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt
Ausgabe: Ja/Nein
Im „Nein"-Fall soll eine Folgemarkierung angegeben werden, von der aus M_0 nicht erreichbar ist.

B22: Es sollen alle Kreis-Eigenschaften wie beim LTS überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt
Ausgabe: Siehe A8, A9 und A12

B23: Es soll die *Lebendig*-Eigenschaft (und andere Varianten von Lebendigkeit) überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt, evtl. einzelne Transition t
Ausgabe: Ja/Nein
Im „Nein"-Fall soll ein Gegenbeispiel angegeben werden.

B24: Es soll die *schwach/stark separierbar*-Eigenschaft bezogen auf k überprüft werden können.
Eingabe: PN, Anfangsmarkierung M_0, beschränkt, natürliche Zahl $k \geq 1$
Ausgabe: Ja/Nein
Im „Nein"-Fall soll ein Gegenbeispiel angegeben werden.

B25: Es soll überprüft werden können, ob es isolierte Elemente gibt.
Eingabe: PN
Ausgabe: Liste der isolierten Elemente

B26: Es sollen die (schwach) zusammenhängenden Komponenten berechnet werden können.
Eingabe: PN
Ausgabe: Liste der Zusammenhangskomponenten

B27: Es sollen die stark zusammenhängenden Komponenten berechnet werden können.
Eingabe: PN
Ausgabe: Liste der starken Zusammenhangskomponenten

B28: Es sollen die Rückwärts-, Vorwärts-, und die Inzidenzmatrix berechnet werden können.
Eingabe: PN
Ausgabe: $\mathbb{B}$ bzw. $\mathbb{F}$ bzw. C

B29: Es sollen alle Nebenbedingungen berechnet werden können.
Eingabe: PN
Ausgabe: Liste aller Nebenbedingungen

B30: Es sollen alle (minimalen, semipositiven) S-Invarianten und T-Invarianten berechnet werden können.
Eingabe: PN
Ausgabe: Liste aller (minimalen, semipositiven) S-Invarianten bzw. T-Invarianten

B31: Es sollen alle minimalen Siphons berechnet werden können.
Eingabe: PN
Ausgabe: Liste aller minimalen Siphons D mit $D \neq \varnothing$

B32: Es sollen alle minimalen Fallen berechnet werden können.
Eingabe: PN
Ausgabe: Liste aller minimalen Fallen Q mit $Q \neq \varnothing$

B33: Es soll das größte k berechnet werden können, für das M_0 eine k-Markierung ist.
Eingabe: PN, Anfangsmarkierung M_0
Ausgabe: Die größte natürliche Zahl $k \geq 1$, so dass M_0 eine k-Markierung ist.

B34: Es soll, falls beschränkt, der Erreichbarkeitsgraph beziehungsweise, falls unbeschränkt, der Überdeckungsgraph berechnet werden können.
Eingabe: PN, Anfangsmarkierung M_0
Ausgabe: Der Erreichbarkeitsgraph von (PN,M_0), falls (PN,M_0) beschränkt ist, ein Überdeckungsgraph von (PN,M_0), falls (PN,M_0) unbeschränkt ist.

Gegeben: Ein beschriftetes Petri-Netz mit Beschriftungsmenge L

C1: Es soll die syntaktische Korrektheit überprüft werden können.
Eingabe: LPN oder LPN mit Anfangsmarkierung M_0
Ausgabe: Ja/Nein

C2: Es soll überprüft werden können, ob ein Wort in der Sprache des LPN liegt.
Eingabe: LPN und Wort w über L
Ausgabe: Eine Feuersequenz, die das gegebene Wort erzeugt, falls $w \in L(LPN)$, Nein falls $w \notin L(LPN)$

C3: Es soll der Erreichbarkeitsgraph berechnet werden können, wenn das LPN beschränkt ist.
Eingabe: LPN, Anfangsmarkierung M_0
Ausgabe: Der Erreichbarkeitsgraph von (PN,M_0), falls (PN,M_0) beschränkt ist, als LTS mit Beschriftungen aus L (nicht notwendigerweise aus der Menge der Transitionen).
Sonst soll es eine Fehlermeldung „(LPN,M_0) unbeschränkt“ geben oder ein Überdeckungsgraph von (PN,M_0) soll ausgegeben werden.

C4: Es soll die Sprachäquivalenz überprüft werden können.
Eingabe: LPN N_1 und LPN N_2 mit Anfangsmarkierungen M_{01} bzw. M_{02}, beide über Beschriftungsmenge L und beide beschränkt
Ausgabe: Ja, falls L(N_1,M_{01})=L(N_2,M_{02}), sonst nein.
Im „Nein“-Fall soll ein Wort angegeben werden, das in der einen Sprache liegt, aber nicht in der anderen.

C5: Es soll die Isomorphie der Erreichbarkeitsgraphen überprüft werden können.
Eingabe: LPN N_1 und LPN N_2 mit Anfangsmarkierungen M_{01} bzw. M_{02}, beide über Beschriftungsmenge L und beide beschränkt
Ausgabe: Ja, falls die Erreichbarkeitsgraphen von (N_1,M_{01}) und von (N_2,M_{02}) isomorph sind. Sonst nein (und Angabe des Grundes).

C6: Es soll die Bisimulation der Erreichbarkeitsgraphen überprüft werden können.

Eingabe: LPN N_1 und LPN N_2 mit Anfangsmarkierungen M_{01} bzw. M_{02}, beide über Beschriftungsmenge L und beide beschränkt.
Ausgabe: Ja, falls die Erreichbarkeitsgraphen von (N_1,M_{01}) und von (N_2,M_{02}) bisimilar sind. Sonst nein (und Angabe des Grundes).

Generierung regelmäßiger Netze

D1: Generierung von n-Bit-Netzen
Eingabe: Eine Zahl n
Ausgabe: Das n-Bit-Netz

D2: Generierung aller n-Philosophen-Netze
Eingabe: Eine Zahl n
Ausgabe: Das n-Philosophen-Netz

D3: Generierung aller schlichten T-Systeme
Eingabe: Eine Zahl ns, eine Zahl nt und eine Zahl m mit ns, nt, $m \in \mathbb{N}$
Ausgabe: Alle schlichten T-Systeme mit höchstens ns Stellen, höchstens nt Transitionen und höchstens m Marken.

D4: Generierung aller schlichten T-Systeme
Eingabe: Eine Zahl nst und eine Zahl m mit nst, $m \in \mathbb{N}$
Ausgabe: Alle schlichten T-Systeme mit höchstens nst Stellen und Transitionen und höchstens m Marken.

Generierung von isomorphen T-Systemen zu speziellen Petri-Netzen

E1: Prüfe folgende Eigenschaften:

- N ist schlicht
- M_0 ist eine k-Markierung mit $k >= 2$
- (N, M_0) ist beschränkt

- (N, M_0) ist reversibel
- (N, M_0) ist persistent
- alle kleinsten Kreise von (N, M_0) haben den gleichen Parikh-Vektor.

Wenn alle Eigenschaften erfüllt sind, prüfe für alle erzeugbaren schlichten T-Systeme (N', M_0') der maximalen Größe g, ob der Erreichbarkeitsgraph von (N', M_0') isomorph mit dem von $(N, M0)$ ist.

Eingabe: Ein Petri-Netz N mit Anfangsmarkierung M_0 und eine Größe g.

Ausgabe: Wenn ein isomorphes T-System gefunden wurde, soll (N', M_0') ausgegeben werden. Wenn kein T-System gefunden wurde, soll diese Tatsache bekannt gegeben werden.

E2: Überprüfung wie in E1, aber es sollen nicht alle (N', M_0') überprüft werden, sondern nur ein zufällig gewähltes.

Eingabe: Ein Petri-Netz N mit Anfangsmarkierung M_0 und eine Größe g.

Ausgabe: Wenn ein isomorphes T-System gefunden wurde, soll (N', M_0') ausgegeben werden. Wenn kein T-System gefunden wurde, soll diese Tatsache bekannt gegeben werden.

Prüfung von Erweiterungen eines gegebenen LTS auf verschiedene Eigenschaften

F1: Prüfe für alle Erweiterungen (Z', s_0) der maximalen Größe g von Z folgende Eigenschaften:

- (Z', s_0) ist reversibel
- (Z', s_0) ist persistent
- alle kleinsten Kreise von Z' haben genau den gleichen Parikh-Vektor.

Eingabe: Ein endliches LTS Z mit Anfangszustand s_0 und eine Größe g

Ausgabe: Wenn eine Erweiterung gefunden wird, soll (Z', s_0) ausgegeben werden (Wenn minimal, Ausgabe „minimal“). Danach Berechnung eines Netzes mit Erreichbarkeitsgraph (Z', s_0), wenn möglich. Wenn keine Erweiterung der Größe

$\leq g$ gefunden wird, Bekanntgabe dieser Tatsache. Dabei ist g die Größe der Differenz zwischen Z und Z'; wenn $g = 0$, dann wird nur $Z' = Z$ untersucht.

F2: Wie F1 nur mit explizit vorgegebenem Parikh-Vektor

Eingabe: Ein endliches LTS Z mit Anfangszustand s_0, eine Größe g und ein Parikh-Vektor

Ausgabe: Wenn eine Erweiterung gefunden wird, soll (Z', s_0) ausgegeben werden. (Wenn minimal, Ausgabe „minimal“) Danach Berechnung eines Netzes mit Erreichbarkeitsgraph (Z', s_0), wenn möglich. Wenn keine Erweiterung der Größe $\leq g$ gefunden wird, Bekanntgabe dieser Tatsache. Dabei ist g die Größe der Differenz zwischen Z und Z'; wenn $g = 0$, dann wird nur $Z' = Z$ untersucht.

F3: Wie F1, außer dass ein ON-Netz mit Erreichbarkeitsgraph (Z', s_0) gesucht wird

Eingabe: Ein endliches LTS Z mit Anfangszustand s_0 und eine Größe g

Ausgabe: Wenn eine Erweiterung gefunden wird, soll (Z', s_0) ausgegeben werden. (Wenn minimal, Ausgabe „minimal“) Danach Berechnung eines ON-Netzes mit Erreichbarkeitsgraph (Z', s_0), wenn möglich. Wenn keine Erweiterung der Größe $\leq g$ gefunden wird, Bekanntgabe dieser Tatsache. Dabei ist g die Größe der Differenz zwischen Z und Z'; wenn $g = 0$, dann wird nur $Z' = Z$ untersucht.

Ermittlung von Petri-Netzen mit bestimmten Eigenschaften

G1: Prüfe, ob es ein Petri-Netz (N, M_0) mit folgenden Eigenschaften gibt:

- N ist schlicht
- (N, M_0) ist beschränkt
- (N, M_0) ist stark lebendig
- (N, M_0) ist nicht reversibel
- (N, M_0) ist persistent
- M_0 ist eine k-Markierung
- (N, M_0) ist nicht stark separierbar

Eingabe: Es wird keine Eingabe benötigt.

Ausgabe: Ja/Nein
Im „Ja“-Fall soll das gefundene Petri-Netz ausgegeben werden.

G2: Prüfe, ob es ein Petri-Netz (N, M_0) mit folgenden Eigenschaften gibt:

- N ist schlicht
- (N, M_0) ist beschränkt
- (N, M_0) ist stark lebendig
- (N, M_0) ist nicht reversibel
- (N, M_0) ist persistent
- M_0 ist eine k-Markierung
- (N, M_0) ist nicht schwach separierbar

Eingabe: Es wird keine Eingabe benötigt.
Ausgabe: Ja/Nein
Im „Ja“-Fall soll das gefundene Petri-Netz ausgegeben werden.

4.2 Nicht-funktionale Anforderungen

In diesem Abschnitt werden die nicht-funktionalen Anforderungen vorgestellt, die an das zu entwickelnde System, als an das APT-Tool, gestellt worden sind. Diese waren nicht von Anfang an vorgegeben, sondern haben sich durch Entscheidungen der Projektgruppe ergeben. Aus diesem Grund ist jede nicht-funktionale Anforderungen durch den Tag [**pg**] oder den Tag [**bet**] gekennzeichnet, wodurch angegeben wird, ob sie von der Projektgruppe oder von den Betreuern stammt.

Technische Anforderungen

NFA1 Das System muss in Java 7 geschrieben sein. [**pg**]

NFA2 Für die Repräsentation von Petri-Netzen und Transitionssystemen müssen günstige Datenstrukturen gewählt werden. [**pg**]

NFA3 Es muss ein bestehendes Dateiformat erweitert oder ein eigenes Dateiformat erstellt werden. [**pg**]

NFA4 Die Petrify-Schnittstelle muss installiertes Petrify nutzen können. [**pg**]

NFA5 Die Synet-Schnittstelle muss installiertes Synet nutzen können. [**pg**]

NFA6 Das System soll über die Kommandozeile gesteuert werden können. [**bet**]

Qualitätsanforderungen

NFA7 Der Quellcode des Systems muss festgelegten Code-Koventionen entsprechen. [**pg**]

NFA8 Der Quellcode des Systems muss über JavaDoc dokumentiert werden. [**pg**]

NFA9 Die Bedienung des APT-Tool soll schnell und intuitiv erlernbar sein. [**bet**]

NFA10 Ein Handbuch/Readme muss dem Anwender zur Verfügung gestellt werden. [**bet**]

NFA11 Das System soll verständliche Fehlermeldungen anzeigen. [**bet**]

NFA12 Das System soll in Abhängigkeit der Komplexität des berechneten Problems in angemessener Zeit eine Antwort liefern. [**pg**]

NFA13 Das System soll durch Verwendung von Modulen erweiterbar gemacht werden. [**pg**]

Benutzungsschnittstellen-Anforderungen

NFA14 Module müssen über die Kommandozeile aufrufbar und ausführbar sein. [**bet**]

Prozessanforderungen

NFA15 Die Projektgruppe muss mindestens alle zwei Wochen tagen. [**pg**]

5 Fremdsysteme

In diesem Kapitel werden die Ergebnisse der Evaluation der Fremdsysteme aufgeführt, die in der ersten Arbeitsphase betrachtet wurden. Dabei handelt es sich um fünf Programme sowie drei studentische Arbeiten. Zu jedem Fremdsystem wird kurz beschrieben, welchem Zweck es dient und inwieweit es in dem zu erstellenden Analysewerkzeug verwendet werden kann.

5.1 LoLA/Sara

LoLA und Sara sind zwei Werkzeuge zur Analyse von Petri-Netzen. Diese werden in den folgenden Abschnitten näher beschrieben.

5.1.1 LoLA

Der Low Level Petri Net Analyzer (LoLA) wurde von Karsten Wolf und anderen entwickelt und wurde am 24. 07. 2008 zum ersten Mal in der Version 1.00 veröffentlicht[1]. In der Analysephase war die Version 1.16 vom 15. 06. 2011 aktuell. Das Werkzeug ermöglicht die Analyse von Petri-Netzen durch die Überprüfung auf verschiedene Eigenschaften. Darüber hinaus bietet LoLA Optimierungstechniken an, um die Analyse zu beschleunigen und ggf. den Speicherverbrauch zu reduzieren. Sowohl Aufgabenstellung als auch zu nutzende Optimierungstechniken müssen ausgewählt werden, bevor

[1] `http://service-technology.org/tools/lola`, zuletzt aufgerufen am 26. 02. 2013

LoLA übersetzt wird. Es gibt also nicht ein allgemeines Programm, das diverse Aufgaben erledigen kann, sondern es werden mehrere spezialisierte Programme erzeugt.

LoLA kann Petri-Netze auf Netzeigenschaften (Reversibilität, Heimatmarkierungen, Deadlockfreiheit, Beschränktheit) und auf Eigenschaften von Markierungen, Stellen und Transitionen überprüfen. Auch CTL-Formeln werden von LoLA unterstützt, um beispielsweise die Erreichbarkeit einer Belegung oder die Lebendigkeit von Transitionen zu testen.

Beispielsweise wird durch die Formel *p1 = 42 AND p2 >= 32* eine Markierung beschrieben, die 42 Token auf die Stelle *p1* und mindestens 32 Token auf die Stelle *p2* legt.

Um zu prüfen, ob eine Transition stark lebendig ist, kann *ALLPATH ALWAYS EXPATH EVENTUALLY (p2 >= 1)* verwendet werden. Hierbei wird angenommen, dass nur *p2* im Vorbereich der Transition ist. Dies gibt bereits einen kleinen Ausblick auf die Möglichkeiten von CTL-Formeln, die für das Analysewerkzeug aber nicht relevant waren.

Eine weitere Besonderheit ist, dass LoLA ein eigenes Dateiformat verwendet, um Petri-Netze zu beschreiben. Dieses Dateiformat kann höhere Netze darstellen. Höhere Netze sind insbesondere hilfreich, um Netze mit wiederkehrenden Strukturen zu beschreiben. Hierdurch können beispielsweise fast beliebig große Philosophen-Netze kompakt beschrieben werden.

Es wurde festgestellt, dass LoLA viele Analyse-Funktionen bereitstellt, diese jedoch fast alle nicht mit unseren Anforderungen übereinstimmen. Außerdem lassen sich externe Programme nicht leicht in Java einbinden, da diese für diverse Plattformen mitgeliefert werden müssen. Dies würde es dem Benutzer erschweren APT zu verwenden. Daher war es nicht sinnvoll, LoLA zu verwenden. Jedoch wurden einige Leistungstest durchgeführt, auf die später eingegangen wird.

5.1.2 Sara

Das Tool „Structures For Automated Reachability Analysis“ (Sara) wurde von Harro Wimmel entwickelt und wurde am 12. 07. 2010 veröffentlicht[2]. Sara kann eingesetzt werden, um Petri-Netze auf die Erreichbarkeit von Markierungen zu überprüfen. Dazu kann pro Stelle im Petri-Netz eine Tokenzahl angegeben werden, die entweder genau erreicht werden muss oder eine Ober- /Untergrenze für die Tokenzahl in der gesuchten Markierung darstellt. Ebenfalls kann durch Sara überprüft werden, ob tote Transitionen, also Transitionen, die niemals schalten können, vorliegen oder ob Transitionsmultimengen realisierbar sind, also eine Schaltfolge existiert, in der jede Transition so oft schaltet, wie in der Eingabe angegeben. Im Gegensatz zu LoLA können Netzeigenschaften nicht direkt getestet werden, sondern müssen, soweit möglich, in eine entsprechende Formel umgewandelt werden.

Das Ergebnis zu Sara fiel ähnlich wie die Beobachtungen zu LoLA aus. Es ist erkennbar, dass Saras Ansatz wenig Gemeinsamkeiten mit den Anforderungen an unser Tool hat. Außerdem würde unser Werkzeug unzuverlässiger werden, wenn ein externes Programm eingebunden wird. Daher wurde Sara nicht weiter verwendet.

Es folgen einige Tests, die die Leistungsfähigkeit von Sara untersuchen.

5.1.3 Leistungstests

Es wurden eine Reihe von Leistungstests mit LoLA und mit Sara durchgeführt. Als Testgrundlage wurde dabei das Philosophenproblem genommen, dass bereits in Definition 3.3.14 vorgestellt wurde. Die Messungen wurden auf drei Systemen mit Intel Core i7-2600 3,4 GHz CPU und 8 GiB RAM unter FreeBSD 9.0 durchgeführt. Dazu wurde folgender Ablauf gewählt:

- Auffalten von höheren Netzen durch LoLA
- Suche nach Deadlocks durch LoLA
- Suche nach Home Markings durch LoLA

[2] `http://download.gna.org/service-tech/sara/`, zuletzt aufgerufen am 26. 03. 2012

- Prüfung von einfacher Lebendigkeit durch Sara

Die Resultate sind in den Abbildungen 5.1 bis 5.4 abgebildet.

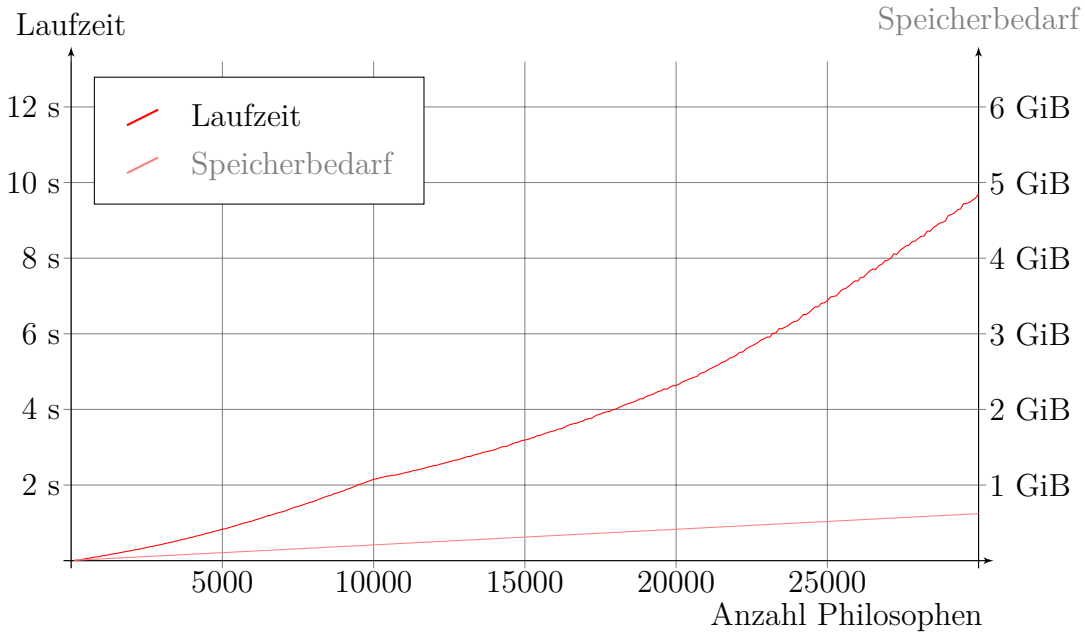


Abbildung 5.1: Auffalten von höheren Netzen durch LoLA

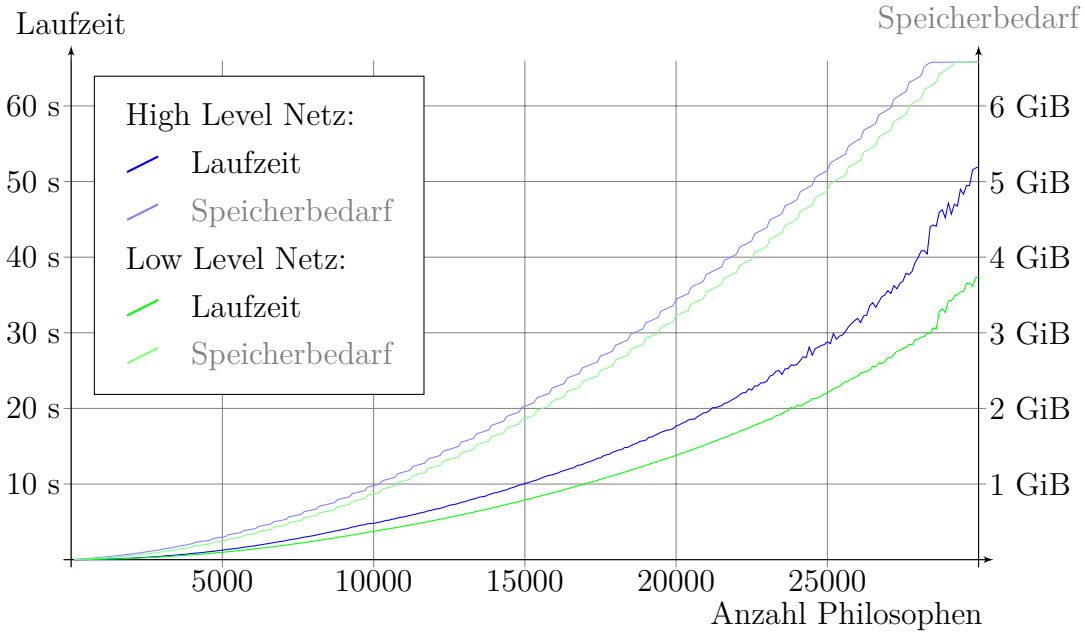


Abbildung 5.2: Suche nach Deadlocks durch LoLA

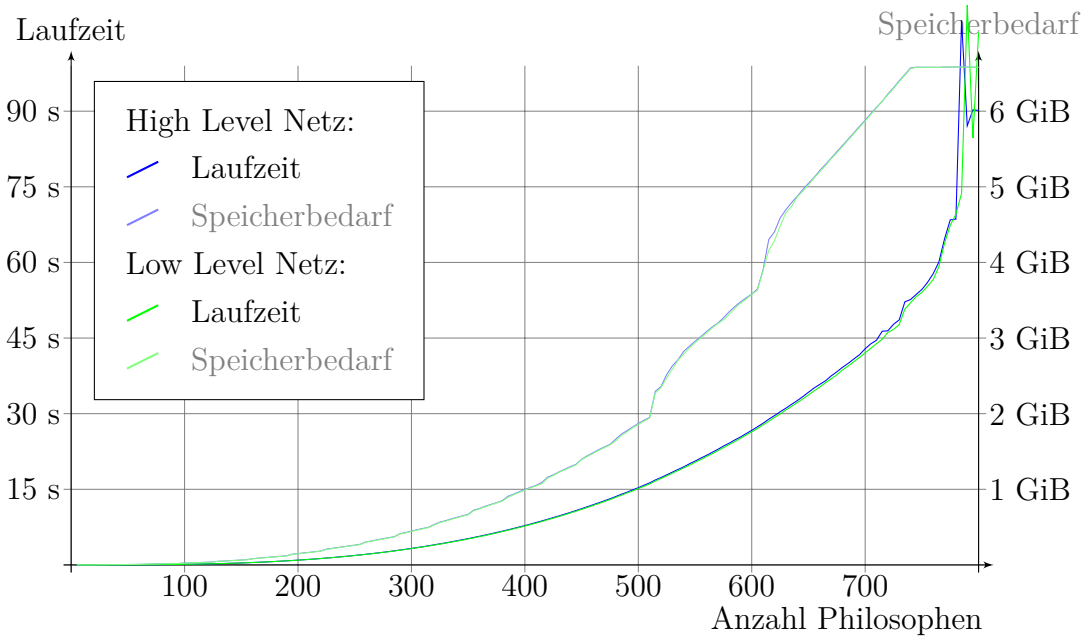


Abbildung 5.3: Suche nach Home Markings durch LoLA

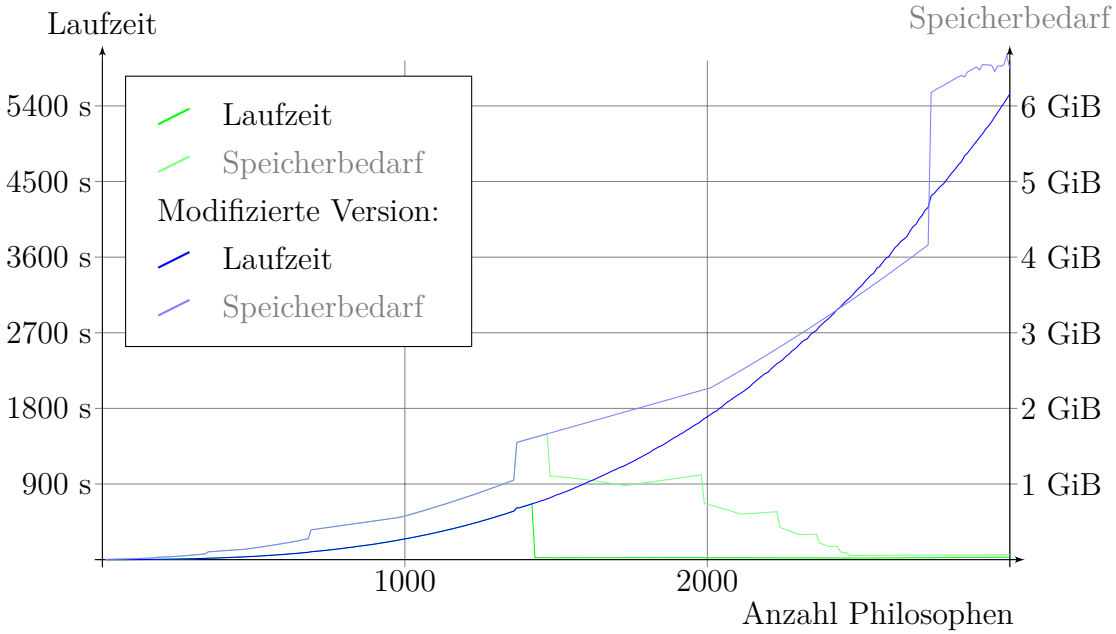


Abbildung 5.4: Prüfung von einfacher Lebendigkeit durch Sara

Beim Auffalten von höheren Netzen konnte festgestellt werden, dass der Speicherbedarf linear ansteigt, während die Laufzeit exponentiell wächst. Bei der Suche nach Deadlocks und Home Markings steigen sowohl Laufzeit als auch Speicherbedarf expo-

nentiell an. Analog ist dies der Fall für Sara bei der Prüfung auf einfache Lebendigkeit. Durch die Tests wurde ersichtlich, dass bei der Entwicklung eines Analysewerkzeugs für Petri-Netze eine gute Speicherausnutzung wichtig ist, da die Auslastung exponentiell anwächst.

5.2 Synet/Petrify

Synet und Petrify sind Tools, mit denen Transitionssysteme analysiert und erzeugt sowie Petri-Netze generiert werden können. In den nächsten beiden Abschnitten werden sie kurz vorgestellt.

5.2.1 Synet

Synet ist das Ergebnis einer französischen Forschungsgruppe, die unter der Leitung von Benoît Caillaud steht. Bei Synet handelt es sich um ein Kommandozeilenprogramm, mit dem es möglich ist, Petri-Netze zu synthetisieren, also aus einem LTS ein PN zu erzeugen, welches das ursprüngliche LTS als Erreichbarkeitsgraph besitzt. Entwickelt wird es seit 1996, wobei das letzte Update mit der Versionsnummer 2.0b von Benoît Caillaud aus dem Jahr 2002 stammt[3].

Synet hat einen elementaren Vorteil im Gegensatz zu den anderen Programmen aus diesem Kapitel: Transitionssysteme *können* Locations enthalten und diese können von Synet synthetisiert werden. Locations sind als disjunkte Gruppierung von Elementen in dem System zu verstehen. Den Übergängen werden Locations zugewiesen, welche sich dann nach dem Synthetisieren auch wieder im Petri-Netz wiederfinden. Dabei ist zu beachten, dass diese Locations nicht zwingend angegeben werden müssen, um das Transitionssystem zu synthetisieren.
Nachteil von Synet ist, dass es nur unter 64-Bit-Unix-Systemen läuft (Getestet unter der 64-Bit Version von Ubuntu 12.04).

[3] `http://www.irisa.fr/s4/tools/synet/`, zuletzt aufgerufen am 26. 03. 2013

Synet nutzt unterschiedliche Formate für seine Dateien, welche im Folgenden kurz vorgestellt werden.
Synet speichert die Locations des Transitionssystem in Dateien mit der Endung .dis. Die Grammatik dieser Dateien ist wie folgt aufgebaut:

```
<.dis> ::= <mapping-list>
<mapping-list> ::= | <mapping> <mapping-list>
<mapping> ::= ( <event-id> , <location-id> )
```

Listing 5.1: .dis-Grammatik

Ein kurzes Beispiel in dem jede Transition einer neuen Location zugeordnet ist:

```
(a,A)
(b,B)
(c,C)
```

Listing 5.2: .dis-Grammatik-Beispiel

Transitionssysteme werden in Synet in .aut Dateien gespeichert, ihre Grammatik lautet:

```
<.aut> ::= des(<nat>,<nat>,<nat>) <trans-list>
<trans-list> ::= | <trans> <trans-list>
<trans> ::= ( <state-id> , <event-id> , <state-id> )
```

Listing 5.3: .aut-Grammatik

Ein Transitionssystem könnte daher wie folgt aussehen:

```
des(0,100,100)
(0,a,1)
(1,c,2)
(2,b,3)
(3,c,0)
```

Listing 5.4: .aut-Grammatik-Beispiel

Die Ausgabe, also das Petri-Netz, wird von Synet in .net-Dateien gespeichert:

```
<.net> ::= <statement-list>
<statement-list> ::= | <statement> <statement-list>
<statement> ::= <location> | <transition> | <place> | <flow>
<location> ::= location <location-id>
<transition> ::= transition <event-id> {:: <location-id>}
<place> ::= place <place-id> {:= <nat>} {:: <location-id>}
<flow> ::= flow <place-id> -- { <nat> | # } -> <event-id>
| flow <place-id> <- { <nat> } -- <event-id>
```

Listing 5.5: .net-Grammatik

Synet benötigt die beiden obigen Dateien als Eingabe um folgendes Netz zu erstellen:

```
location B
location C
location A
transition b :: B
transition c :: C
transition a :: A
place x_2 :: C
place x_1 := 1 :: B
place x_0 := 2 :: A
flow x_2 ---> c
flow x_1 --2-> b
flow x_0 --2-> a
flow x_2 <--- b
flow x_2 <--- a
flow x_1 <--- c
flow x_0 <--- c
```

Listing 5.6: .net-Grammatik-Beispiel

Synet kann dabei zwei Ausgaben generieren, entweder wird der Output auf der Konsole oder in einer Datei ausgegeben. Dabei wird dann das .net-Format genutzt.
Weil Synet das uns einzig bekannte Tool ist, welches Locations verarbeiten kann,

haben wir uns dazu entschieden Synet mit in unser Werkzeug zu integrieren.

5.2.2 Petrify

Petrify[4] wurde von Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor und Alexandre Yakovlev 1994 erstmals entwickelt. Petrify ist genau wie Synet ein Tool, welches zum Synthetisieren von Petri-Netzen genutzt wird. Im Gegensatz zu Synet nutzt Petrify das ASTG-Format (Aynchronus Signal Transition Graph), um Graphen darzustellen. Auch das ASTG-Format stellt mehr Möglichkeiten zur Verfügung als wir nutzen werden. Ein kurzes Beispiel sieht wie folgt aus:

```
.model name
.inputs a b c d
.graph
a c
c a
b d
d b
.marking { <c,a> <d,b> }
.end
```

Listing 5.7: Ein Beispielnetz im Petrify-Format

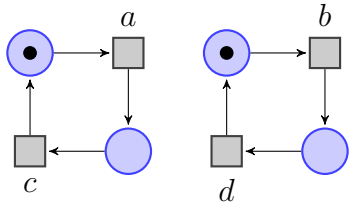


Abbildung 5.5: Beispiel aus Listing 5.7

An diesem Beispiel fällt sofort eine elementare Eigenschaft des ASTG-Formats auf: Es werden keine Stellen definiert, sondern nur Transitionen. Es nimmt daher die Stellen dazwischen einfach implizit an. Somit kann die Markierung schließlich nicht direkt auf

[4] `http://www.lsi.upc.edu/~jordicf/petrify/`, zuletzt aufgerufen am 26. 03. 2013

Stellen deklariert werden, sondern es müssen für einen Token auf einer Stelle die zwei Transitionen angegeben werden, zwischen denen diese Stelle implizit angenommen wird. Daher müssen auch nur die Transitionen vorher deklariert werden. Dabei gibt es Ausnahmen; wenn beispielsweise eine Stelle, in die kein Token zurückführt, markiert ist, dann wird diese mit *pZahl* angegeben und können dann auch explizit markiert werden. Weiterhin zu beachten ist, dass die Netze 1-beschränkt sind. Nachteil von Petrify ist, dass es nur unter 32-Bit-Systemen läuft.

Der Unterschied zwischen Synet und Petrify ist, dass Petrify keine Locations verwalten kann und 1-beschränkte Netze als Eingabe erwartet. Der Vorteil war jedoch bisher, dass Petrify in Performance Tests besser abgeschnitten hat als Synet und daher ebenfalls implementiert wurde.

5.3 PEP

PEP[5] ist ein in der Programmiersprache C und TCL geschriebenes Modellierungs- und Verifikations-Framework für parallele Systeme und bietet eine große Anzahl an verschiedenen Modellierungssprachen und Verifikationstechniken. Entwickelt wurde dieses Werkzeug von einer Forschungsgruppe um Eike Best, Hans Fleischhack, Bernd Grahlmann und Christian Stehno aus Oldenburg. Die aktuelle Version ist 2.0 Beta 4 vom September 2004.

Das PEP-Werkzeug beinhaltet eine graphische Oberfläche, die die einzelnen Module für Modellierungs-, Verifikations- und Analyseaufgaben verwaltet. Für unsere Problemstellung waren lediglich letztere relevant, zu deren Funktionsumfang zum Beispiel Tests auf Beschränktheit, Lebendigkeit von FC-Systemen und T-Systemen, Reversibilität und strukturelle Eigenschaften, sowie Algorithmen zur Berechnung von Schaltfolgen, Siphons und Fallen, Erreichbarkeit von Markierungen wie auch den Überdeckungsgraphen gehören. Eine weitere Besonderheit des Programms stellen die Generatoren für spezielle Netzklassen dar.

Ein großes Problem bei der Nutzbarkeit des Programms stellt die Nichterstellbarkeit

[5] `http://sourceforge.net/projects/peptool/`, zuletzt aufgerufen am 26. 03. 2013

Modul	Erstellbar?	Funktionstüchtig?
Bounded	Ja	Ja
Bufgen	Ja	Ja
Elementary	Ja	Ja
Erreichbarkeit	Ja	Ja, beachte Vorbedingungen
Fc_system_liveness	Ja	testet Lebendigkeit von T-Systemen
Freechoice	Ja	Ja
Gencomp	Nein	–
Maxconf	Nein	–
Mm	Nein	–
Pargen	Ja	Ja
pep2lp	Nein	–
Philgen	Ja	Ja
Phillgen	Ja	Ja
Reversibel	Ja	Ja, beachte Vorbedingungen
Schaltfolgen	Ja	Ja
Simplify	Ja	Ja
Slist	Nein	–
Slotgen	Ja	Ja
Trap	Ja	Ja
t_system	Ja	Ja
t_system_deadlock	Ja	Nein
t_system_liveness	Ja	Ja
Unfold	Ja	Ja

Tabelle 5.1: PEP-Module

des gesamten Systems dar. Diese liegt zum Beispiel an fehlenden Dateien, unterschiedlichen, gleichzeitig existierenden Versionen von ein und derselben Datei sowie an einer Auslegung auf alte Systeme. Vorteilhaft ist jedoch das modulare Konzept, sodass sich einzelne Algorithmen mit etwas Aufwand und Anpassungen separat erstellen lassen.

Tabelle 5.1 bietet eine Übersicht über die Module mit der Information, ob diese erstellbar waren und richtig funktionieren.

Zum Testen der Algorithmen wurden die Programme unter verschiedenen Betriebssystemen erstellt. Verwendet wurden Ubuntu 11.4 und 12.4, Mac OS X 10.6 und 10.7

mit gcc 4.2, Windows Vista SP2 mit mingw32 4.4. Erstellt wurde PEP immer als 32-Bit-Version, da es bei der 64-Bit-Version zum Beispiel Fehler bei der Adressberechnung gab.

Neben der Nichterstellbarkeit einiger Module ist die Verwendung der Programme mit weiteren Schwierigkeiten behaftet, da es in großen Zügen an Dokumentationen fehlt. Beispielsweise wird bei den Algorithmen für Erreichbarkeit wie auch Reversibilität ein lebendiges und beschränktes FC-System erwartet, was jedoch weder geprüft wird, noch an geeigneter Stelle dokumentiert ist. Außerdem prüft das Modul fc_system_liveness, welches die Lebendigkeit von FC-System testen soll, die Lebendigkeit von T-Systemen (prüft, ob alle kleinsten Kreise mindestens ein Token unter der Startmarkierung haben). Diese Probleme treten wahrscheinlich nicht auf, wenn das ganze Projekt erstellt und über die graphische Oberfläche genutzt wird, erschwert jedoch das separate Verwenden der einzelnen Module. Zusätzlich scheint das Modul t_system_deadlock noch nicht vollständig implementiert zu sein, da die relevante Methode in jedem Fall wahr zurück liefert.

5.3.1 Nutzbarkeit für das Projekt

Aus der Sicht der Projektgruppe ist PEP ein mächtiges Instrument. Es bietet viele Möglichkeiten von der Erzeugung, über die Transformation bis zur Analyse von Petri-Netzen. Jedoch gibt es entscheidende Einschränkungen durch die erwähnten Schwächen. Ein Beispiel für die Nutzung von gemeinsamen Dateien in verschiedenen, inkompatiblen Versionen ist ein Modul zur Verwaltung von Listen, welches in neun verschiedenen Versionen vorliegt. Auch die knappe Kommentierung der Quelltexte und Variablennamen in nicht aussagekräftiger Form erschweren die direkte Übernahme von PEP.

Aus diesen Gründen wurde PEP für das Projekt nicht direkt verwendet, sondern lediglich als Inspirationsquelle genutzt.

5.4 Studentische Arbeiten

Neben den in vorigen Abschnitten evaluierten Programmen wurden auch ausgewählte studentische Arbeiten, die einen Bezug zu Petri-Netzen und Transitionssystemen sowie deren Eigenschaften hatten, betrachtet und in Bezug auf Verwendbarkeit evaluiert. Die nachfolgenden Abschnitte stellen die Ergebnisse dar.

5.4.1 Diplomarbeit Robert Bleiker

Robert Bleiker entwickelte in seiner Diplomarbeit *Das Wortproblem für Petri-Netze* [Ble09] ein Programm inklusive grafischem Editor[6], mit dem es ermöglicht wird, ein beschriftetes Petri-Netz auf gesuchte Wörter zu untersuchen bzw. alle enthaltenen Wörter einer gewissen Länge auszugeben.

Die Idee hinter dieser Arbeit basiert auf der Automatentheorie, in der es möglich ist, Sprachen durch Automaten zu beschreiben. Durch die Ähnlichkeit zwischen Automaten und Petri-Netzen soll das Wortproblem auf Petri-Netze übertragen werden. Zu diesem Zweck muss ein Petri-Netz von drei- ($N = (S, T, F)$) auf sieben Tupel ($N = (S, T, F, M_0, M_f, \sum, h)$) erweitert werden, um Start- und Endzustände anzugeben. Das Tupel setzt sich wie folgt zusammen:

- $(S, T, F, M_0, \sum, h_l)$ ist ein beschriftetes Petri-Netz.
- $\mathrm{M}_f \subseteq \mathrm{N}^s$ ist die endliche Menge von Endzuständen.

Ziel dieser Arbeit war es, durch ein Programm herauszufinden, ob ein gegebenes Wort in der Sprache liegt, sowie alle Wörter auszugeben, die in der Sprache des Petri-Netzes liegen können.

Zur Analyse erstellte Robert Bleiker eine eigene Datenstruktur, die von jeder Stelle den Vor- sowie Nachbereich enthält. Durch diese hohe Redundanz wird der Rechenaufwand minimiert. Bei Robert Bleikers Algorithmus wird ausgehend von der

[6] Unter `http://www.uni-oldenburg.de/department-fuer-informatik/parsys/arbeiten/abgeschlossen/2009/da-bleiker/` (Letzter Aufruf: 26. 03. 2013) ist das Programm abrufbar.

Startmarkierung die Folgemarkierung bestimmt und die Präfixe der Folgemarkierung zwischengespeichert. Dabei werden nur Transitionen berücksichtigt, die den Folgebuchstaben des gesuchten Wortes enthalten. Dies geschieht für jeden zwischengespeicherte Markierung. Wird ein Endzustand erreicht, werden die Präfixe als Wort der Sprache ausgegeben bzw. überprüft, ob das gesuchte Wort erzeugt wurde. Wird jedoch die vorgegebene Wortlänge erreicht, ohne dass ein Endzustand erreicht wurde, bricht die Suche ab. Das Suchverfahren ist per Breitensuche implementiert.

Ein durch Robert Bleiker inspiriertes Verfahren zur Worterkennung wird im APT-Tool verwendet.

5.4.2 Diplomarbeit Florian Hinz

In seiner Diplomarbeit *Untersuchung und Implementierung von Algorithmen zur Isomorphie bei Petri-Netzen* [Hin12] behandelte Florian Hinz verschiedene Algorithmen zur Isomorphieprüfung bei Petri-Netzen und implementierte schließlich einen dieser Algorithmen.

Um die Isomorphie bei zwei Petri-Netzen zu prüfen, müssen zunächst die Überdeckungsgraphen zu den Netzen erstellt werden, um schließlich auf die Erreichbarkeitsgraphen direkt den Isomorphie-Test anzuwenden. Daher wendet Florian Hinz zunächst den Algorithmus von Starke (siehe [Sta90]) zur Generierung der Überdeckungsgraphen an und prüft die Isomorphie mit dem später ausführlicher beschriebenen VF2-Algorithmus.

Das System von Florian Hinz wurde in C++ geschrieben und ist kommandozeilenorientiert. Als Eingabeformat für Petri-Netze wird das low- und high-level-PEP-Format für Petri-Netze verwendet (siehe Abschnitt 5.3). Zusätzlich zur Isomorphieprüfung existiert ein Petri-Netz-Generator für n-Bit-Netze. Um effizient auf Graphen arbeiten zu können, verwendet Florian Hinz die in C++ geschriebene Open-Source-Bibliothek LEMON [7].

[7] `http://lemon.cs.elte.hu` , zuletzt aufgerufen am 26. 03. 2013

Da der VF2-Algorithmus zur Isomorphie-Prüfung im Vergleich mit anderen Isomorphie-Algorithmen eine gute Performance hat, wurde entschieden, diesen Algorithmus zur Erfüllung der Anforderung C5 als Isomorphie-Test für beschriftete Petri-Netze auch in dem APT-System zu nutzen. Details zum VF2-Algorithmus und seiner Komplexität können in Abschnitt 7.4.3 nachgelesen werden. Dem Starke-Algorithmus zur Erzeugung des Überdeckungsgraphen wurde ein anderer Algorithmus vorgezogen. Details zu diesem gewählten Algorithmus können in Abschnitt 7.3.2 nachgeschlagen werden.

Um den VF2-Algorithmus zu benutzen, wurde nicht das ganze Programm von Florian Hinz eingebunden, sondern es wurde lediglich das Code-Fragment, welches den VF2-Algorithmus enthält, in Java übersetzt und in das Modul zur Isomorphie-Prüfung eingebunden. Die Entscheidung gegen das Einbinden des ganzen Fremdsystems basiert auf den folgenden Argumenten:

Programmiersprachen-Konflikt: Das APT-Werkzeug ist in Java geschrieben, die Arbeit von Florian Hinz in C++, daher wäre das Einbinden des Programms inklusive der genutzten Bibliotheken umständlich.

Redundanz-Vermeidung: Nicht das ganze Programm, sondern lediglich der Part mit dem VF2-Algorithmus wird benötigt. Viele unnötige Klassen und Bibliotheken würden in das APT-System eingebunden.

Dateiformat-Konflikt: Das Fremdsystem unterstützt ein anderes Dateiformat, es müssten also Dateiformate geparst werden.

Wartbarkeit: In das System integrierter Java-Code ist wesentlich einfacher zu Warten und Weiterzuentwickeln als Code in einem Fremdsystem.

5.4.3 Diplomarbeit Yangzi Zhang

Yangzi Zhang setzte sich in ihrer Diplomarbeit mit dem Thema *Algorithmische Überprüfung struktureller Eigenschaften von Petri-Netzen und Transitionssystemen*[Zha10]

auseinander. Die Ausführung des Systems dieser Diplomarbeit war leider nicht möglich, weswegen die Implementierung dieser Arbeit nicht für das APT-Werkzeug benutzt wurde.

6 Implementierung

In diesem Kapitel werden vier zentrale Implementierungs-Konzepte des APT-Werkzeugs beschrieben:

Datenstrukturen: Es wurden eigene Datenstrukturen entworfen. In diesem Abschnitt soll zunächst begründet werden, warum nicht auf eine vorhandene Datenstruktur-Bibliothek zurückgegriffen wurde, bevor ein Klassendiagramm einen Überblick über die implementierten Strukturen gibt und Details zur technischen Umsetzung jeweils anhand eines Beispiels zu Petri-Netze mit und ohne Beschriftungen und zu beschrifteten Transitionssystemen erläutert werden.

Modulsystem: Das implementierte Modulsystem unterstützt eine saubere Abkapselung interner Analyse-Algorithmen von der Benutzungsschnittstelle. In diesem Abschnitt wird anhand des `BoundedModule` erläutert, wie das Modulsystem funktioniert.

Dateiformat: Das Dateiformat `.apt` wurde eigens für das APT-Werkzeug entwickelt. In diesem Abschnitt wird jeweils anhand eines Beispiels erläutert, wie das Dateiformat für Petri-Netze mit und ohne Beschriftungen und für Transitionssysteme aufgebaut ist. Weiterhin soll motiviert werden, warum ein eigenes Dateiformat entwickelt wurde und kein bestehendes Dateiformat gewählt wurde.

Parser und Renderer Die Parser und Renderer bilden die Schnittstelle zwischen den Datenstrukturen und den Dateiformaten. In diesem Abschnitt wird erklärt welche Parser und Renderer in dem Werkzeug implementiert und welche Konzepte dazu verwandt wurden. Des Weiteren wird anhand des *APTPNParser* erklärt

wie ein eigener Parser in das Werkzeug integriert werden kann.

6.1 Datenstrukturen

Dieser Abschnitt stellt die den Algorithmen zugrunde liegenden Datenstrukturen für Petri-Netze und Transitionssysteme vor. Die Entwicklung der selbigen orientiert sich stark an den in Abschnitt 4 beschriebenen Anforderungen; sie ist also auf die dort vorgestellten Probleme und gewünschten Algorithmen zugeschnitten. Dies stellt auch den hauptsächlichen Grund für die Entwicklung eigener Datenstrukturen und der Entscheidung, keine bestehenden Bibliotheken zu verwenden beziehungsweise zu erweitern, dar. Für diese Entscheidung wurden folgende Graphbibliotheken betrachtet:

C++: IGraph[1], OGDF (Open Graph Drawing Framework)[2], BGL (Boost Graph Library)[3], LEDA (Library of Efficient Data Types and Algorithm)[4], LEMON (Library for Efficient Modeling and Optimization in Networks)[5], PNAPI (Petri Net API)[6]

C#: .Net Graph Library[7], QuickGraph[8]

Python: SNAKES (The Net Algebra Kit For Editors And Simulators)[9]

Java: JBPT (Business Process Technologies 4 Java)[10], JUNG (Java Universal Network/Graph Framework)[11], JFern(Java-based Petri Net framework)[12], PNK (Petri Net Kernel)[13]

[1] `http://igraph.sourceforge.net/introduction.html` , zuletzt aufgerufen am 26. 03. 2013
[2] `http://www.ogdf.net` , zuletzt aufgerufen am 26. 03. 2013
[3] `http://www.boost.org/doc/libs/1_50_0/libs/graph/doc/table_of_contents.html` , zuletzt aufgerufen am 26. 03. 2013
[4] `http://www.algorithmic-solutions.com/leda/index.htm` , zuletzt aufgerufen am 26. 03. 2013
[5] `http://lemon.cs.elte.hu` , zuletzt aufgerufen am 26. 03. 2013
[6] `http://www.informatik.uni-rostock.de/~nl/tools.html` , zuletzt aufgerufen am 26. 03. 2013
[7] `http://graphlib.codeplex.com` , zuletzt aufgerufen am 26. 03. 2013
[8] `http://quickgraph.codeplex.com` , zuletzt aufgerufen am 26. 03. 2013
[9] `http://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/index.html` , zuletzt aufgerufen am 26. 03. 2013
[10] `http://code.google.com/p/jbpt/` , zuletzt aufgerufen am 26. 03. 2013
[11] `http://jung.sourceforge.net/` , zuletzt aufgerufen am 26. 03. 2013
[12] `http://jfern.sourceforge.net/` , zuletzt aufgerufen am 26. 03. 2013
[13] `http://www2.informatik.hu-berlin.de/top/pnk/` , zuletzt aufgerufen am 26. 03. 2013

Die betrachteten Bibliotheken waren teilweise spezifisch für den benötigten Anwendungsfall entwickelt (zum Beispiel: OGDF, PNAPI, JBPT, JFern, JUNG), oder hatten eine hohe Einstiegskomplexität (zum Beispiel: BGL). Auch waren sie häufig nur rudimentär dokumentiert und eine spezielle Implementierung für Transitionssysteme fehlte (nur Implementierungen für allgemeine Graphen, oder zusätzlich für Petri-Netze), so dass diese hätte integriert werden müssen. Aufgrund dessen fiel die Entscheidung eine eigene Bibliothek für unsere Problematik – losgelöst von anderen Bibliotheken, aber unter deren Verwendung als Inspirationsquelle – zu entwickeln. Den solidesten Eindruck für unsere Bedürfnisse machten jedoch der Petri Net Kernel (Java), LEMON (C++), QuickGraph (C#) und SNAKES (Python).

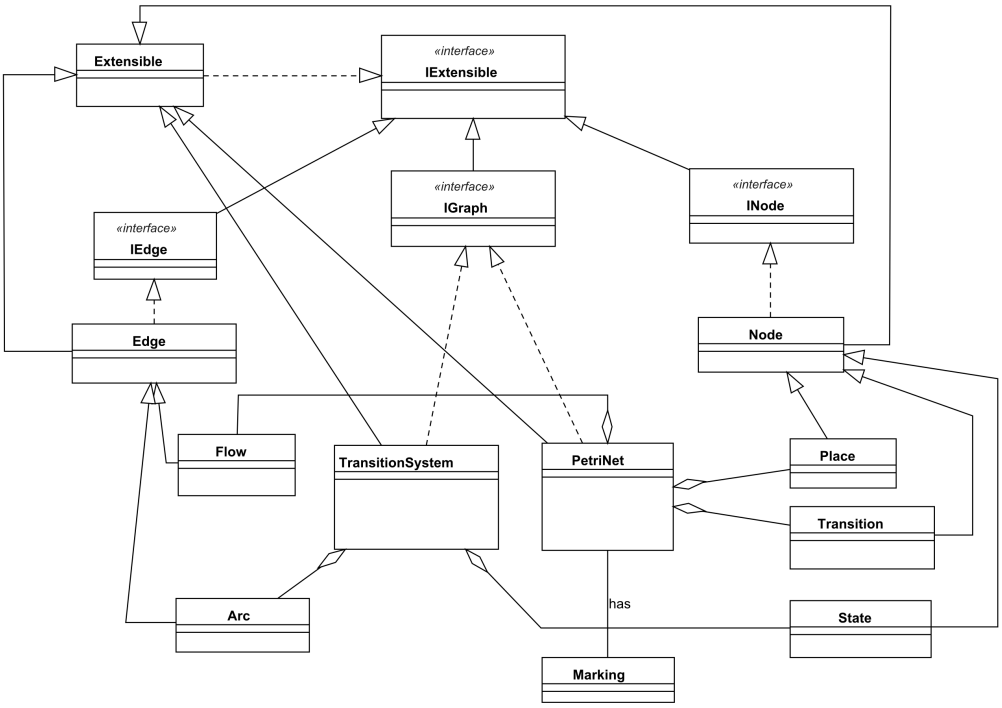


Abbildung 6.1: Klassendiagramm zu den Datenstrukturen

In Abbildung 6.1 ist eine Übersicht über die relevanten Klassen des Datenstrukturenpaketes und deren Abhängigkeiten gegeben. Dabei ist die grundlegende strukturelle Idee für den Aufbau der Datenstrukturen die zentrale Verwaltung von Daten in der `PetriNet`- beziehungsweise `TransitionSystem`-Klasse. Objekte, wie zum Beispiel Stellen, Transitionen und Kanten, können nicht direkt erstellt werden, sondern müssen über die Graph-Klassen erzeugt werden.

Im Folgenden werden zuerst die Funktionalitäten für Petri-Netze und anschließend die für Transitionssysteme vorgestellt, wobei diese beiden Systeme sehr analog zueinander implementiert wurden.

6.1.1 Petri-Netze mit und ohne Beschriftungen

Dieser Abschnitt stellt zuerst einige Grundfunktionen der Datenstrukturen für Petri-Netze vor und demonstriert diese anschließend ein einem etwas größeres Beispiel.

```
PetriNet pn = new PetriNet("Testnet");
// Erzeuge Stelle mit der ID p1
Place p1 = pn.createPlace("p1");
// Erzeuge Stelle mit einer autogenerierten ID (p0)
Place p0 = pn.createPlace();
// Erzeuge drei Stellen mit den angegebenen IDs
Place[] places = pn.createPlaces("p2", "p3", "p4");
// Erzeuge Transition mit ID t1 und Beschriftung label
Transition t1 = pn.createTransition("t1", "label");
// Erzeuge Transition mit ID t3 und Beschriftung t3
Transition t3 = pn.createTransition("t3");
// Erzeuge Transition mit autogenerierter ID (t0)
Transition t0 = pn.createTransition();
// Erzeuge fünf Transitionen mit autogenerierten IDs
Transition[] transitions = pn.createTransitions(5);
```

Listing 6.1: Erstellen von Stellen und Transitionen

Wie in Listing 6.1 beschrieben, kann ein Petri-Netz optional mit einem Namen versehen werden. Zum Erzeugen von Stellen und Transitionen gibt es die beiden Ansätze, die Objekte mit einem eindeutigen Bezeichner (ID) zu erstellen oder automatisch eine ID erstellen zu lassen. Dabei muss die ID – in Anlehnung an Definition 3.3.1 – nicht nur für die Stellen und Transitionen jeweils eindeutig sein, sondern auch eindeutig in der Vereinigung der beiden Mengen. Es darf also keine Transition dieselbe ID haben wie eine Stelle. Die Transitionen und Stellen werden im Petri-Netz lexikalisch sortiert, gespeichert und als eine nicht veränderbare Menge nach außen zur Verfügung gestellt, wodurch die Vorgehensweise, dass nur das Petri-Netz Veränderungen vornehmen kann, sichergestellt wird. Dies liegt vorallem darin begründet, dass die

Methoden, die auf den Objekten selber aufgerufen werden können, lediglich Weiterleitungen (Delegate) an die zugehörige Graph-Klasse darstellen. Dies bedeutet, dass die Objekte selber nahezu keine Funktionalitäten implementiert haben, sondern lediglich die zugehörigen Funktionen der Graph-Klasse aufrufen. Eine Ausnahme von dieser Vorgehensweise stellt lediglich die Klasse für Markierungen eines Petri-Netzes dar, auf die später noch genauer eingegangen wird.

Die `create`-Methoden liefern immer das erzeugte Objekt beziehungsweise eine Liste der erzeugten Objekte zurück, damit diese weiterverwendet werden können. Zusätzlich existieren auch noch `create`-Methoden, die Stellen, Transitionen oder Kanten aus anderen Netzen kopieren können. Wird keine Beschriftung für eine Transition angegeben, wird automatisch die ID als Beschriftung gewählt.

Wie in Listing 6.2 zu sehen ist, können Kanten über die IDs der Knoten oder die Referenzen auf die Objekte selber erstellt werden. Dabei ist der erste Parameter immer der Startknoten und der zweite der Zielknoten.

```
// Kante p1 -> t2 mit Gewicht 1
Flow f = pn.createFlow(p1, t2);
f.setWeight(2); // Gewicht auf 2 geändert
// Kante p2 -> t1 mit Gewicht 3
Flow flow = pn.createFlow("p2", "t1", 3);
```

Listing 6.2: Erstellen von Kanten

Die Initialmarkierung eines Petri-Netzes kann auf zwei Wegen erstellt werden. Zum einen kann sie insgesamt als Tokenanzahlen zu den zugehörigen Stellen im Netz gesetzt werden oder es kann einzelnd den Stellen eines Petri-Netzes eine Tokenanzahl zugewiesen werden (vgl. Listing 6.3).

```
// Setzt die Initialmarkierung in lexikalischer Ordnung
pn.setInitialMarking(new Marking(pn, 1, 2, 3, 4, 5));
// Ändert die Tokenanzahl der Initialmarkierung von Stelle p1
p1.setInitialToken(5);
```

Listing 6.3: Setzen einer initialen Markierung

Markierungen haben im Allgemeinen die Möglichkeit, sich aufgrund des Feuerns ei-

ner Transition auf zwei unterschiedliche Arten zu verhalten. Zum einen kann von der Transition aus geschaltet werden, dann wird die übergebene Markierung nicht verändert, sondern eine neue angelegt. Soll jedoch die Markierung selbst verändert werden, dann kann von der Markierung das Feuern durchgeführt werden. Diese beiden Vorgehensweisen sind in Listing 6.4 dargestellt und haben den Vorteil, dass zum einen eine schnellere Version ohne kopieren der Markierung existiert und zum anderen sich nicht um das Kopieren von Markierungen gekümmert werden muss. Damit sind die beiden hauptsächlich benötigten Anwendungsfälle effizient abgedeckt.

```
// Erzeuge eine Markierung für ein Netz mit 4 Stellen
Marking m = new Marking(pn, 1, 2, 3, 4);
// z. B. [p3:4] [p2:3] [p1:2] [p0:1]
// Erzeuge eine echte Kopie dieser Markierung. Lediglich die Referenz auf
// das zugehörige Petri-Netz ist eine Referenzkopie.
Marking mcopy = new Marking(m);
// feure t3 (p2-t3->p0), ändere dabei aber m nicht
Marking changed = pn.getTransition("t3").fire(m);
// Also m = [p3:4] [p2:3] [p1:2] [p0:1] und
// changed = [p3:4] [p2:2] [p1:2] [p0:2]
// feure nun t3 und ändere dabei mcopy
Marking marking = mcopy.fire(pn.getTransition("t3"));
// Also mcopy = [p3:4] [p2:2] [p1:2] [p0:2]
// Referenzgleichheit: marking = mcopy
```

Listing 6.4: Feuern von Transitionen

Analog zum Erstellen von Objekten können diese auch wieder aus dem Netz entfernt werden. Bei Stellen und Transitionen kann dies einfach über die ID oder das Referenzobjekt selbst gelöst werden. Dabei werden die ein- und ausgehenden Kanten des gelöschten Knoten ebenso entfernt. Kanten können eindeutig über ihren Start- und Zielknoten identifiziert und damit gelöscht werden, wie in Listing 6.5 zu sehen ist.

```
pn.removeNode(t2);
pn.removePlace("p4");
pn.removeNode("p3");
pn.removeFlow("p2", "t1");
```

Listing 6.5: Löschen von Objekten

Zusätzlich bieten das Petri-Netz wie auch die Knoten im Petri-Netz die Möglichkeit, den Vor- und Nachbereich eines Knoten zu erhalten. Dabei handelt es sich wieder

lediglich um eine unveränderliche Sicht auf die in dem Netz vorgehaltenen Daten.

Die Speicherung der Vor- wie auch Nachbereiche wird, um ein gutes Mittel zwischen Speicher- und Laufzeitkomplexität zu erlangen, in den Datenstrukturen mithilfe von den von Java zur Verfügung gestellten *SoftReferences* gelöst. Das bedeutet, dass die Vor- und Nachbereiche (Knoten wie auch Kanten) zwar für jeden Knoten gespeichert werden, der GarbageCollector jedoch die Möglichkeit hat, wenn der Speicherbedarf der Anwendung zu groß wird, diese zu löschen. Damit muss also zum Beispiel beim Erstellen und Löschen von Kanten wie auch Knoten darauf geachtet werden, dass eventuell die relevanten Mengen neu berechnet werden müssen, damit diese immer konsistent sind. Dieses Verfahren ist jedoch für den Nutzer der Datenstrukturen nach außen nicht sichtbar. Der Anwender bekommt zu jeder Zeit die Vor- und Nachbereiche, ohne dass es für ihn relevant ist, ob sie gerade neu berechnet oder die gespeicherten geliefert werden.

Um möglichst frei die Objekte mit zusätzlichen Informationen verknüpfen zu können, implementieren alle Objekte (`PetriNet`, `Place`, `Transition`, `Flow`) das Interface `IExtensible`, welches eine `HashMap` von `String` und `Object` enthält. Das heißt diesen Instanzen können beliebige Objekte mit einer Zeichenkette als Key hinzugefügt werden. Da bei einer Kopie der einzelnen Klassen auch diese `HashMap`s kopiert werden, kann zusätzlich angegeben werden, welche der hinzugefügten Objekte mit kopiert werden sollen. Hierbei ist zu beachten, da sich beliebige Objekte (`Object`) in der `HashMap` speichern lassen, dass es sich bei der Kopie der einzelnen Values aus der `HashMap` lediglich um eine Referenzkopie handeln kann. Beispielhaft ist das Hinzufügen einer Extension, deren Referenz bei der Kopie des Transitionssystems kopiert werden würde, in Listing 6.7 zu sehen.

Abschließend werden beispielhaft einige Möglichkeiten der Petri-Netz-Strukturen in Listing 6.6 dargestellt und die daraus resultierende graphische Darstellung in Abbildung 6.2 visualisiert.

```java
PetriNet pn = new PetriNet("Testnet");
// Erzeuge einige Stellen
Place[] places = pn.createPlaces("p1", "p2", "p4");
// Erzeugt eine mit ID p0 autogenerierte Stelle
Place p3 = pn.createPlace();
// Erzeuge einige Transitionen
Transition[] transitions = pn.createTransitions("t1", "t2", "t3");
Transition t4 = pn.createTransition();
// Erzeuge einige Kanten
pn.createFlow("t1", "p1", 2);
Flow f = pn.createFlow("p1", "t2");
f.setWeight(3);
pn.createFlow("t2", "p2");
pn.createFlow("p2", "t3");
pn.createFlow("t3", p3.getId());
pn.createFlow(p3, t4);
pn.createFlow(t4.getId(), "p4");
pn.createFlow("p4", "t1");
// Lösche einen Knoten einschließlich der Kanten
pn.removeTransition(t4);
// Lösche eine Kante
pn.removeFlow("t1", "p1");
// Auch dies löscht eine Kante
pn.getFlow("p1", "t2").setWeight(0);
// Setzt die initiale Marking in lexikalischer Sortierung
pn.setInitialMarking(new Marking(pn, 1, 2, 3, 4));
// Ändert die initiale Markierung
pn.getPlace("p1").setInitialToken(5);
```

Listing 6.6: Beispiel für einige Funktionalitäten der Petri-Netz-Klassen

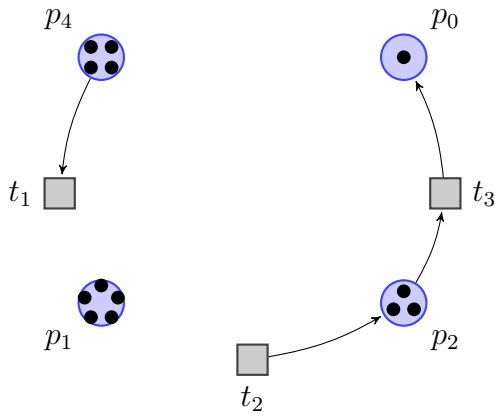


Abbildung 6.2: Das durch Listing 6.6 erzeugte Petri-Netz

6.1.2 Beschriftete Transitionssysteme

Die Struktur und die Funktionalität eines Transitionssystems sind analog zu der eines Petri-Netzes implementiert. Dabei unterscheiden sie sich von denen eines Petri-Netzes nur dadurch, dass die Kanten zusätzlich eine Beschriftung besitzen und damit eindeutig durch Start-, Zielknoten und Beschriftung identifiziert werden können und dass aufgrund dieser Beschriftung ein Alphabet (die Menge der Beschriftungen) berechnet und bei jeder Veränderung der Beschriftungen im System aktualisiert wird. Außerdem gibt es keine Markierungen mehr, aber einen initialen Zustand. Ein etwas größeres Beispiel befindet sich in Listing 6.7 und der zugehörige Graph wird in Abbildung 6.3 gezeigt.

Die Vorteile der Vielzahl an automatisierten Tests, wie in Abschnitt 9.1 beschrieben, wurden bei der Entwicklung der Datenstrukturen gut deutlich, da sie mitten in der Entwicklungsphase der Algorithmen neu strukturiert und teilweise komplett neu geschrieben wurden. Es stellte sich als eine große Erleichterung heraus, dass durch einfaches Ausführen der über 1600 Tests schnell viele Fehler, Randfälle und unpraktische Implementierungen in den Datenstrukturen aufgedeckt wurden und nach erfolgreichem Durchlauf der Tests ein gutes Maß für die Sinnhaftigkeit der Implementierung gegeben war.

```
TransitionSystem ts = new TransitionSystem("testSystem");
// Erstelle drei Zustände
State[] states = ts.createStates("s1", "s2", "s3");
// Erstelle einen Zustand (s0) und setze ihn als Startzustand
ts.setInitialState(ts.createState());
// Kante erstellen
Arc a = ts.createArc("s1", "s2", "a");
// und nachträglich das Label ändern
a.setLabel("b");
// Ein paar weitere Kanten
ts.createArc(states[0], states[1], "a"); /*Achtung: states[0] hat ID s1*/
ts.createArc(states[1], states[2], "b");
ts.createArc("s2", "s3", "c");
State s0 = ts.getInitialState();
ts.createArc(states[2], s0, "d");
ts.createArc(s0, states[1], "a");
ts.createArc(s0, states[1], "b");
// Merke das bis hier her erstellte Alphabet in einer Extension
ts.putExtension("initialAlphabet", new HashSet<>(ts.getAlphabet()));
// Lösche einen Knoten einschließlich der Kanten
ts.removeState("s1");
// Durch Setzen dieses Labels würden 2 Kanten s0 -e-> s2
// existieren
try {
    ts.getArc(s0, states[1], "b").setLabel("a");
} catch (ArcExistsException e) {
}
// Setze die Labels von den Kanten aus dem Vorbereich von s2 um
Set<Arc> ed = ts.getPresetEdges("s2");
int counter = 0;
for (Arc arc : ed) {
    arc.setLabel("l" + (++counter));
}
// Verändere noch ein Label
ts.getArc(states[2].getPresetNodes().iterator().next(), states[2], "b").
    setLabel("a");
```

Listing 6.7: Beispiel für einige Funktionalitäten der Transitionssystem-Klassen

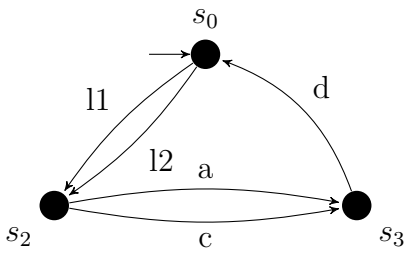


Abbildung 6.3: Das durch Listing 6.7 erzeugte Transitionssystem

6.2 Modulsystem

Dieser Abschnitt beschreibt das verwendete Modulsystem. Die in Kapitel 4 aufgeführten Anforderungen beschreiben jeweils eine für sich stehende Aufgabe. Um diese Aufgaben und die Algorithmen, die diese lösen, sichtbar für den Benutzer von APT zu kennzeichnen, wurde das Konzept des Moduls eingeführt.

Ein Modul nimmt eine Eingabe entgegen und erzeugt aus dieser eine Ausgabe. Die Eingabe und Ausgabe kann dabei beliebig komplex sein und ist abhängig von der Aufgabe, die das Modul löst. Üblicherweise ist die Eingabe ein Petri-Netz oder ein beschriftetes Transitionssystem, während die Ausgabe beschreibt, ob das übergebene Petri-Netz bzw. beschriftete Transitionssystem eine bestimmte Eigenschaft erfüllt oder nicht; komplexere Ausgaben von Petri-Netzen und Transitionssystemen, sowie beliebige Kombinationen sind ebenfalls vorgesehen.

Zur Umsetzung von Modulen wurde im Java-Programm das Interface `Module` eingeführt. Ein Modul implementiert dieses Interface, um zu beschreiben, welche Eingabe verarbeitet und welche Ausgabe daraus erzeugt werden kann. Zudem kann das Modul durch die Implementierung seinen Namen sowie eine Beschreibung zur Verfügung stellen; diese Informationen werden dem Benutzer über die Benutzungsschnittstelle, die im Kapitel 8 beschrieben wird, angezeigt.

Mithilfe der Methoden `require()` und `provide()` des Interface `Module` spezifiziert ein Modul seine Eingabe und Ausgabe, die vom Modulsystem genutzt werden, um Module

in APT aufzurufen. Diese Spezifikationen liegen dem Modulsystem als Objekte der Klassen `ModuleInputSpec` bzw. `ModuleOutputSpec` vor.

Durch Methoden wie `getName()` und `getLongDescription()` werden der Benutzungsschnittstelle Informationen bereitgestellt, die dem Benutzer angezeigt werden, um das Modul zu beschreiben. Die Benutzungsschnittstelle nutzt die Informationen aus dem Objekt der Klasse `ModuleInputSpec` eines Moduls, um dem Benutzer mitzuteilen, welche Eingaben das Modul versteht; das Objekt dient somit zusätzlich dem Zweck, dem Benutzer bei der Nutzung eines Moduls behilflich zu sein.

Innerhalb der Methode `require()` bestimmt ein Modul die erwartete Eingabe. Zur Spezifikation der Eingabeparameter dient die Methode `addParameter()`. Die vereinfachte Signatur dieser Methode zeigt, welche Daten zur Spezifikation der Eingabe benötigt werden.

```
addParameter(name, class, documentation, properties)
```

Der Aufruf der Methode wird anhand des in APT genutzten Moduls `BoundedModule` deutlich.

```
addParameter("pn", PetriNet.class,
            "The Petri net that should be examined");
```

Das Modul `BoundedModule` spezifiziert einen Parameter mit dem Namen `pn` vom Typ Petri-Netz (in Java die Klasse `PetriNet`); zudem wird die Bedeutung des Parameters durch den Text „The Petri net that should be examined“ beschrieben.

Das Modul `BoundedModule` nutzt das `properties`-Argument der genutzten Methode `addParameter()` nicht. Allgemein können Module jedoch dem Modulsystem mithilfe der `properties` mitteilen, dass ein Parameter bestimmte Vorbedingungen erfüllen muss (z. B. dass das eingegebene Petri-Netz schlicht sein muss). Werden diese Vorbedindungen nicht erfüllt, kann das Modul nicht aufgerufen werden und somit nicht seine Aufgabe erfüllen; der Benutzer wird durch die Benutzungsschnittstelle über diesen Umstand informiert.

Neben den benötigten Parametern, die durch die Methode `addParameter()` spezifiziert werden, können auch optionale Parameter spezifiziert werden. Dazu dient die Methode `addOptionalParameter()`, die eine ähnliche Signatur besitzt. Da ein optionaler Parameter jedoch entfallen kann, muss ein Wert bestimmt werden, der automatisch an das Modul übergeben wird, wenn der Parameter nicht angegeben wurde.

Im Modul `BoundedModule` wird diese Methode genutzt, um in der Eingabe den optionalen Parameter `k` als natürliche Zahl zu erlauben; falls `k` nicht übergeben wird, wird stattdessen der Wert `null` an das Modul übergeben.

```
addOptionalParameter("k", Integer.class, null,
                     "If given, k-boundedness is checked");
```

Innerhalb der Methode `provide()` spezifiziert ein Modul seine Ausgabe. Dazu dient die Methode `addReturnValue()`. Die vereinfachte Signatur der Methode zeigt, welche Daten zur Spezifikation der Ausgabe benötigt werden.

```
addReturnValue(name, klass, properties)
```

Der Aufruf dieser Methode wird erneut im Modul `BoundedModule` verdeutlicht:

```
addReturnValue("bounded", Boolean.class,
               ModuleOutputSpec.PROPERTY_SUCCESS);
```

Das Modul `BoundedModule` spezifiziert als Ausgabe den Rückgabewert mit dem Namen `bounded`; der Rückgabewert ist ein Wahrheitswert (in Java die Klasse `Boolean`).

Der `properties`-Parameter kann von Modulen genutzt werden, um festzulegen, dass ein Rückgabewert bestimmte Eigenschaften besitzt. Als `properties` wurde im Modul `BoundedModule` der Methode die Eigenschaft `PROPERTY_SUCCESS` übergeben; diese besagt, dass der Wahrheitswert des Rückgabewerts `bounded` darüber entscheidet, ob das Modul erfolgreich war oder nicht (Erfolg steht hierbei für die Erfüllung der Beschränktheit).

Die Methoden `addParameter()`, `addOptionalParameter()` und `addReturnValue()`

lassen sich in der Methode `require()` bzw. `provide()` beliebig oft aufrufen, so dass ein Modul beliebig viele Parameter und Rückgabewerte spezifizieren kann.

Mit der spezifizierten Eingabe und Ausgabe eines Moduls kann dieses durch die Methode `run()` aufgerufen werden. Ein Modul implementiert diese Methode, um den tatsächlichen Algorithmus auszuführen und somit die gewünschte Aufgabe zu lösen.

Dazu erhält das Modul Zugriff auf die vom Benutzer übergebenen Argumente entsprechend der Spezifikation der Eingabe durch die Methode `require()` in Form eines Objektes der Klasse `ModuleInput`. Das Modul kann dann die Ausgabe entsprechend der Spezifikation in der Methode `provide()` in Form eines Objektes der Klasse `ModuleOutput` erzeugen und somit dem Benutzer zur Verfügung stellen.

Ein vereinfachter Auszug der Implementierung im Modul `BoundedModule` zeigt die Aufgabe der Methode `run()`.

```
run(ModuleInput input, ModuleOutput output) {
  PetriNet pn = input.getParameter("pn", PetriNet.class);
  bounded_result = ... // Aufruf des eigentlichen Algorithmus...
  output.setReturnValue("bounded", Boolean.class, bounded_result);
}
```

Das Modul `BoundedModule` greift auf das Petri-Netz als Argument `pn` der Eingabe zu und führt den Algorithmus, der das Petri-Netz auf Beschränktheit prüft, aus und erzeugt so den Rückgabewert `bounded` bestehend aus dem Ergebnis des Algorithmus als Ausgabe.

Die Abbildung 6.4 veranschaulicht die in diesem Abschnitt vorgestellten Konzepte und stellt diese in Beziehung zueinander.

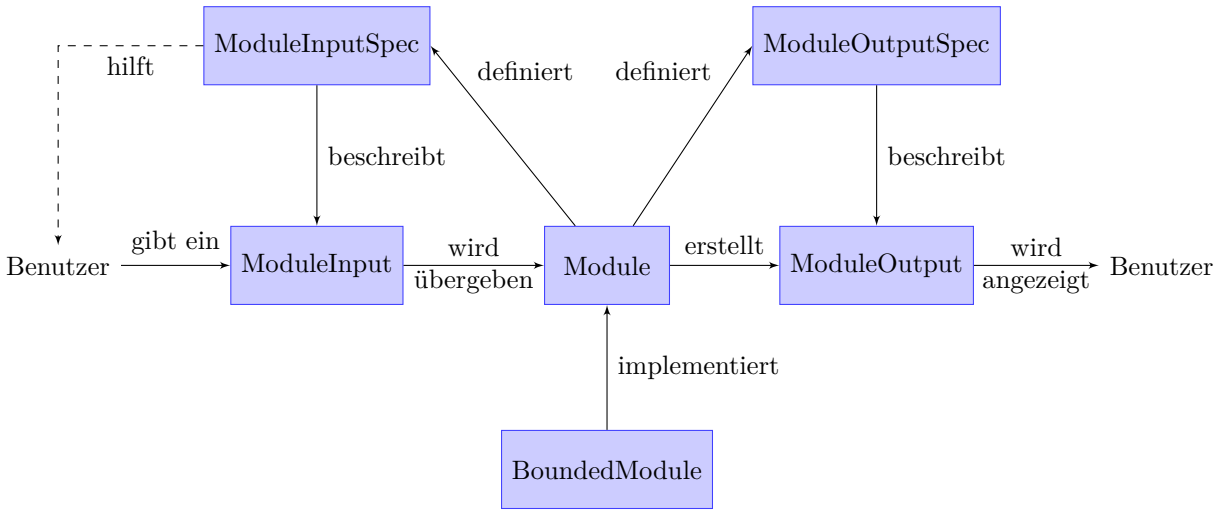


Abbildung 6.4: Ein grober Überblick des Modulsystems anhand des Moduls `BoundedModule`.

6.3 Dateiformate

Dieser Abschnitt stellt das in unserem Werkzeug verwendete Dateiformat für Petri-Netze mit und ohne Beschriftungen wie auch für beschriftete Transitionssysteme vor. Im Folgenden wird es kurz als das APT-Format – in Anlehnung an unseren Projektgruppen-Namen – bezeichnet. Dieses Format ist so strukturiert, dass versucht wurde, eine hohe Lesbarkeit für den Menschen wie auch für die Maschinen zu gewährleisten, wie auch möglichst einfach und schnell schreibbar zu sein. Die Entwicklung eines eigenen und dem Nichtbenutzen eines existierenden Dateiformates liegt eben in diesen erbetenen Anforderungen an das Dateiformat begründet, welche von keinem der betrachteten Formate (LoLA, PEP, APNN, PNML, Synet, Petrify) in gewünschtem Maße gegeben war.

Allgemein ist das APT-Format in Sektionen eingeteilt, die mit der Beschriftung *.sektion* beginnen und unterhalb beziehungsweise neben dieser Beschriftung den jeweiligen Inhalt enthalten. Dabei werden als Kompromiss zwischen der Lesbar- und der Schreibbarkeit Whitespaces ignoriert. Dies verringert zwar die Lesbarkeit, da zum Beispiel der gesamte Graph in einer Zeile notiert werden kann, jedoch vereinfacht es die Schreibbarkeit, da sich nicht gemerkt werden muss, an welchen Stellen Zeilenumbrüche von Nöten sind, um Parserfehler zu vermeiden. Es liegt also in der Verantwortung

des Anwenders diesen Liberalismus nicht auszunutzen und an dieser Stelle auf eine Lesbarkeit seiner Dateien zu achten.

Im Folgenden wird zunächst das Format für Petri-Netze mit und ohne Beschriftungen und anschließend das für beschriftete Transitionssysteme vorgestellt.

6.3.1 Petri-Netze mit und ohne Beschriftungen

Dieser Abschnitt liefert eine Beschreibung jeder Sektion von Petri-Netzen mit und ohne Beschriftungen mit anschließendem Beispiel, in dem die Variationen der Schreibweise dieser Sektion vorkommen.

Der Name einer Datei oder des Graphen wird durch

```
.name "name"
```

angegeben. Dabei sind in der Zeichenkette alle Zeichen abgesehen von den doppelten Anführungszeichen, Zeilenumbrüchen und Tabulatoren erlaubt.

Der Typ des Graphen wird mit

```
.type LPN
```

definiert; dabei wird festgelegt, ob es sich bei dem vorliegenden Graphen um ein Transitionssystem (LTS) oder um ein Petri-Netz mit (LPN) oder ohne Beschriftungen (PN) handelt.

Optional kann dem Petri-Netz eine Beschreibung angefügt werden.

```
.description "Ein Beschreibungstext für das Petri-Netz"
```

Die dort verwendete Zeichenkette ist die Einzige, deren möglichen Zeichen unterschiedlich zu den oben beschriebenen ist. Bei dieser können zusätzlich – für längere

Beschreibungen – Zeilenumbrüche eingefügt werden.

Die Stellen, die im Petri-Netz verwendet werden sollen, werden mit

```
.places
s1[key="value", key2="value"]
s2
```

angegeben, wobei mit den Key-Value-Paaren optional beliebige Werte an Stellen gebunden werden können. Ein Beispiel dafür sind die in Abschnitt 5.2.1 eingeführten Locations. Als Keys können Bezeichner, die mit einem kleinen oder großen Buchstaben (a-z, A-Z) oder dem Unterstrich beginnen und weiter nur Buchstaben (a-z, A-Z), Zahlen (0-9) oder Unterstriche nutzen, verwendet werden. Falls keine zusätzlichen Werte hinzugefügt werden sollen, können die Klammern und Werte weggelassen werden. Bei den Bezeichnern der Stellen können dieselben Zeichen gewählt werden wie für die Keys, bis auf das zusätzlich – angelehnt an das Format von Synet – auch mehrziffrige Zahlen (0-9) verwandt werden können.

Analog zu den Stellen können auch die Transitionen definiert werden.

```
.transitions
t1[key="value", key2="value", label="label"]
t2
```

Eine Besonderheit nimmt dabei der Key *label* ein, welcher für beschriftete Petri-Netze notwendig ist. Mit unserem Parser (vgl. 6.4.1) werden diese dann direkt als Beschriftung in dem Petri-Netz gesetzt, wobei die anderen Key-Value-Paare lediglich als Extensions (vgl. 6.1.1) angefügt werden. Falls keine Beschriftung gesetzt ist, wird der Bezeichner als Beschriftung verwandt. Weiteres zur konkreten Implementierung des Parser für dieses Dateiformat findet sich im Abschnitt 6.4.1.

In der Sektion *.flows* werden die Verbindungen zwischen Transitionen und Stellen aufgeführt, indem jeweils für die Transitionen deren Vor- wie auch Nachbereich angegeben wird.

Im folgenden Beispiel wird also die Stelle `s1` über die Transition `t1` mit sich selbst verbunden. Dabei ist der Bereich vor dem Pfeil der Vorbereich der Transition, ihr Nachbereich befindet sich hinter dem Pfeil.

```
.flows
t1: {s1} -> {s1}
```

Mehrfachgewichtete Kanten werden mit einem Multiplikator aus den natürlichen Zahlen versehen, welcher angibt, wie viele Token mit dieser Kante übertragen werden sollen.

```
.flows
t1: {} -> {s2, 2*s1, s2, 0*s3, 3*s2}
```

Dabei muss das Kantengewicht 1 nicht zusätzlich als Multiplikator angegeben werden und der Multiplikator 0, also eine Kante mit dem Gewicht 0, ist gleichbedeutend mit dem Nichtauflisten dieser Stelle im Nach- beziehungsweise Vorbereich. Damit kann also der leere Nach- beziehungsweise Vorbereich mit der leeren Menge {} erhalten werden. Außerdem ergibt sich das dreifache Auflisten von `s2` mit einmaligem Multiplikator 3 zu einer Kante mit dem Gewicht 5. Des Weiteren ist zu beachten, dass die Angabe des Multiplikators nicht kommutativ ist. Dies liegt daran, dass – wie oben erwähnt – auch Stellennamen aus einfachen Zahlen existieren können und damit die Semantik nicht mehr eindeutig wäre.

Analog zu der Angabe der Vor- beziehungsweise Nachbereiche in Multi-Mengenschreibweise kann die Anfangsmarkierung des Petri-Netzes definiert werden.

```
.initial_marking {2*s1, s2}
```

Es befinden sich also initial zwei Token auf der Stelle `s1` und ein Token auf der Stelle `s2`. Wird diese Sektion komplett weggelassen, dann hat das Netz auf keiner Stelle ein Token.

Eine weitere wichtige Funktion innerhalb des Formats sind die Kommentare. Diese werden mit // initialisiert und gelten für die gesamte Zeile oder werden mit /* */

für Kommentarbereiche, die auch mehrzeilig sein dürfen, maskiert.

```
// Kommentar
.flows
t1: {} -> {2*s1,/*s2,*/ 3*s3}
```

Insgesamt könnte also ein Beispielnetz, wie in Abbildung 6.5 graphisch dargestellt, im APT-Format wie in Listing 6.8 aussehen.

```
//Beispielnetz
.name "testNet.net"
.type LPN

.places
s1[location="China"]
s2[location="China", _annotation="output"]
s3[_annotation="input"]
s4

.transitions
t1[label="b"]
t2[label="c"]
t3
t4[label="b"]

.flows
t1: {2*s2} -> {2*s1}
t2: {} -> {s4}
t3: {s1, s4, 3*s2, s3} -> {1*s3}
t4: {s3} -> {}

.initial_marking {2*s1, s3}
```

Listing 6.8: Beispiel für Petri-Netze

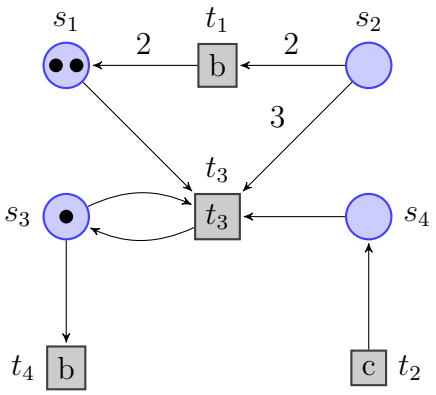


Abbildung 6.5: Das durch Listing 6.8 definierte Petri-Netz

Eine Grammatik wie diese einzelnen Definitionen in dem implementierten Parser umgesetzt werden, befindet sich in Listing 6.16.

Die einzelnen Sektionen dürfen in beliebiger Reihenfolge aufgeführt werden, wobei die Abschnitte *.name*, *.initial_marking* und *.description* null- oder genau einmal auftreten dürfen und die Abschnitte *.places*, *.transitions* und *.flows* nullmal oder beliebig häufig verwandt werden dürfen. Wenn sie mehrfach auftreten, dann werden alle Werte für die Definition des Netzes verwandt. Die Sektion *.type* muss genau einmal aufgeführt sein.

6.3.2 Beschriftete Transitionssysteme

Dieser Abschnitt beschäftigt sich mit dem APT-Format für beschriftete Transitionssysteme, dessen Struktur sehr ähnlich zu der eines Petri-Netzes gewählt wurde, um die Verwendung des Formates zu vereinfachen.

Der Name, wie auch eine optionale Beschreibung des Transitionssystem wird durch

```
.name "name"
.description "Eine etwas längere
Beschreibung"
```

angegeben, wobei die Typen der Zeichenketten wie für Petri-Netze definiert sind. Der Typ des Graphen wird für Transitionssysteme wie folgt angegeben.

```
.type LTS
```

Eine Besonderheit beim Erstellen der Zustände des Transitionssystems stellt die verpflichtende Angabe eines Initialzustands dar, die wie folgt realisiert werden kann:

```
.states
s0[initial, key="value"]
s1[key="value"]
s2
```

Ansonsten sind die Key-Value-Paare wieder optional und die Syntax der Bezeichner ist wie für Petri-Netze definiert. Des Weiteren lassen sich die Beschriftungen, die im Transitionssystem verwendet werden sollen, wie folgt angeben:

```
.labels
a[key="value"]
c
```

Auch hier ist die Syntax der Bezeichner wie bei Petri-Netzen gewählt.

Die gerichteten Kanten können in der Art „Anfangsknoten Label Endknoten“ aufgelistet werden:

```
.arcs
s0 a s1
s1 c s0
```

Abschließend könnte als das in Abbildung 6.6 dargestellte Transitionssystem wie in Listing 6.9 im APT-Format definiert werden.

```
.name "testLTS.aut"
.type LTS

.states
s0[initial]
s1
s2
s3
s4
s5

.labels
a[ext="error"]
c
e

.arcs
s0 a s1
s1 a s2
s2 c s3
s3 c s4
s4 e s5
s5 e s0
```

Listing 6.9: Beispiel für Transitionssysteme

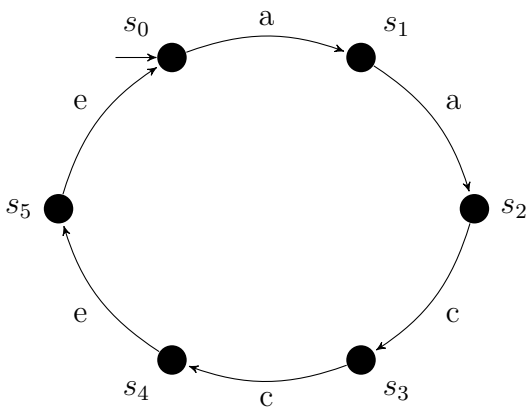


Abbildung 6.6: Das durch Listing 6.7 erzeugte Transitionssystem.

Die Sortierung der einzelnen Sektionen ist beliebig und die Abschnitte *.name* und *.description* dürfen null- oder genau einmal auftreten. Die Abschnitte *.states*, *.labels* und *.arcs* können nullmal oder beliebig häufig verwandt werden und werden bei Mehrfachauftreten wieder zusammengefügt. Die Sektion *.type* muss genau einmal aufgeführt

sein.

In Listing 6.17 findet sich eine Grammatik für den Parser, die die Definitionen dieses Formates umsetzt.

6.4 Parser und Renderer

Dieser Abschnitt stellt die in unserem Werkzeug verwendeten Parser wie auch Renderer vor und erklärt, wie sich mithilfe einer für ANTLR[14] (**AN**other **T**ool for **L**anguage **R**ecognition) erstellten Grammatikdatei eigene Parser einbinden lassen. Insgesamt bietet das Werkzeug Renderer für das APT-, Dot-, LoLA-, PNML-, Petrify- und Synetformat, wie auch Parser für das APT-, Synet- und Petrifyformat.

6.4.1 Parser

Ein Parser bietet die Möglichkeit aus einer Datei in einem speziellen Dateiformat ein Petri-Netz oder ein Transitionssystem zu erstellen. Dafür wird in unserem Werkzeug der Parsergenerator ANTLR verwandt. Dieser erstellt mithilfe einer Grammatikdatei eine Parser- und Lexerklasse in Java.

Eine Besonderheit der durch ANTLR erstellten Parser ist das Recovering. Dabei versucht der Parser bei nicht korrekten Eingaben die Datei trotzdem weiterzuparsen, indem zum Beispiel das nicht passende Zeichen ausgelassen wird. Solche Fälle stürzen auch in unserem Werkzeug nicht ab, sondern liefern lediglich Warnungen. Sollte jedoch die Datei nicht mehr parsebar sein, so werden Exceptions mit Fehlermeldungen, einschließlich Zeilen- und Zeichennummer, zur Verfügung gestellt.

Im Folgenden ist zunächst die Struktur des Parserpaketes erläutert, indem beispielhaft das Hinzufügen eines Parsers für das in Abschnitt 6.3.1 vorgestellte Dateiformat für

[14] `http://www.antlr.org/`, zuletzt aufgerufen am 26. 03. 2013

Petri-Netze mit und ohne Beschriftungen dargestellt ist. Anschließend befinden sich jeweils Grammatiken für die in Abschnitt 6.3 definierten Dateiformate, mit deren Hilfe die Dateien in unserem Werkzeug geparst werden.

Hinzufügen eines eigenen Parser

Folgender Abschnitt erklärt mithilfe des Parsers für das APT-Format für Petri-Netze mit und ohne Beschriftungen das Hinzufügen eines eigenen Parsers zu unserem Werkzeug. Gleichzeitig wird dabei auch der Aufbau und die Struktur des Parserpaketes dargestellt und motiviert.

Um möglichst generisch eigene Parser hinzufügen zu können, wurde eine Struktur mithilfe der Reflexion-API von Java entwickelt, welche die Ansteuerung von ANTLR übernimmt. Dafür muss der in Listing 6.10 aufgeführte Konstruktor zu der Grammatikdatei des Parsers hinzugefügt werden, welches hier dargestellt wird für das Beispiel des `AptPNFormatParser`s.

```
@members {
    private IPNParserOutput<?> out;

    public AptPNFormatParser(CommonTokenStream cts,
                    IPNParserOutput<?> out) {
        this(cts);
        this.out = out;
    }
}
```

Listing 6.10: Konstruktor und Output zum Parser hinzufügen

Der in Listing 6.10 verwendete `IPNParserOutput` bildet die Schnittstelle zu den internen Datenstrukturen des Parsers. Diese wurden zum einen implementiert, um das Parserpaket möglichst unabhängig in anderen Projekten verwenden und eigene Datenstrukturen einfügen zu können. Zum anderen werden damit spezielle Fehlermeldung der Datenstrukturen umgangen. Zum Beispiel ergibt sich so die Möglichkeit, im

APT-Format die Sektion für die Kanten vor der Definition der Knoten aufzuführen, wo sonst beim Hinzufügen von Kanten zwischen nicht existierenden Knoten Fehlermeldungen geworfen werden. Die Umwandlung von den internen Datenstrukturen des Parser zu denen des Werkzeugs wird in der `convertToDatastructure`-Methode, welche in der `APTPNParserOutput`-Klasse beziehungsweise `APTLTSParserOutput`-Klasse implementiert wird, realisiert. Um das Parserpaket für eigene Datenstrukturen verwenden zu können, muss somit lediglich von der `AbstractPNParserOutput`-Klasse beziehungsweise `AbstractLTSParserOutput`-Klasse geerbt und die besagte `convertToDatastructure`-Methode implementiert werden.

```
@rulecatch {
    catch (RecognitionException re) {
        recover(input,re);
        throw re;
    }
}
```

Listing 6.11: Exceptionhandling zum Parser hinzufügen

Damit die vom Parser geworfenen Exceptions vernünftig weiterverarbeitet werden können, muss zusätzlich in der Grammatikdatei noch die in Listing 6.11 aufgeführten Zeilen hinzugefügt werden.

Die Nutzung des Parser funktioniert dann wie in Listing 6.12 dargestellt für das Beispiel des `APTPNParser`. Dafür wird ein `ParserContext` benötigt, welcher die von ANTLR erstellten Klassen, die Klasse des gewünschten Outputs, sowie das Startsymbol der Grammatik benötigt.

```
ParserContext<PetriNet> ctx = new APTParserContext<>(
                  AptPNFormatLexer.class,
                  AptPNFormatParser.class,
                  APTPNParserOutput.class,
                  "start");
PetriNet pn = new ANTLRParser().parse(data, ctx);
```

Listing 6.12: Starten des Parser

Der generische Parameter des `ParserContext`es gibt dabei direkt an, ob ein Transitionssystem oder ein Petri-Netz zurückgeliefert werden soll.

Damit ANTLR die benötigten Dateien beim Erstellen des gesamten Werkzeuges miterzeugt, müssen die in Listing 6.13 dargestellten Zeilen angepasst und eingefügt werden. Dies geschieht in der `build.xml` in dem Bereich `<target name="antlr">`.

```
<java classpathref="project.class.path" classname="org.antlr.Tool" fork=
    "true" failonerror="true">
    <arg value="-verbose"/>
    <arg value="-make"/>
    <arg value="-o"/>
    <arg path="generated-src/uniol/apt/io/parser/impl/apt/"/>
    <arg path="src/uniol/apt/io/parser/impl/apt/AptPNFormat.g"/>
</java>
```

Listing 6.13: Anbindung an das Build-Skript

All dies ist für die Formate von Synet und APT für Petri-Netze und Transitionssysteme in dem Werkzeug schon implementiert und somit können Dateien in den jeweiligen Formaten zum Beispiel mit den in Listing 6.14 dargestellten Möglichkeiten geparst werden.

```
TransitionSystem ts = SynetLTSParser.getLTS("<pathToFile>");
PetriNet net = SynetPNParser.getPetriNet("<pathToFile>");
ts = APTLTSParser.getLTS("<pathToFile>");
net = APTPNParser.getPetriNet("<pathToFile>");
```

Listing 6.14: Implementierte Parser

Des Weiteren kann der Parser bei dem APT-Format mithilfe des Typ-Attributes selbst entscheiden, ob ein Transitionssystem oder ein Petri-Netz zu parsen ist. Diese Möglichkeit ist in Listing 6.15 dargestellt.

```
APTParser parser = new APTParser();
parser.parse("<pathToFile>");
// net == null, wenn es sich um ein Transitionssystem handelt.
PetriNet net = parser.getPn();
// ts == null, wenn es sich um ein Petri-Netz handelt.
TransitionSystem ts = parser.getTs();
```

Listing 6.15: Parser für Petri-Netze und Transitionssysteme

Grammatik für Petri-Netze mit und ohne Beschriftungen

In Listing 6.16 ist eine Grammatik in EBNF mit dem Startsymbol `S` angegeben, welche die korrekte Syntax von Petri-Netzen mit und ohne Beschriftungen im APT-Format definiert. Dabei ist zu beachten, dass die Besonderheit des Formates bezüglich der Häufigkeit des Auftretens der Sektionen nicht in der Grammatik übersichtlich dargestellt werden konnte und auch im implementierten Parser mithilfe von Algorithmen beim Parsevorgang gelöst ist. Ebenso ist der konkrete Umgang mit den Whitespaces wie auch den Kommentaren in der Grammatik nicht direkt dargestellt; diese werden zwar gelesen, aber für die weitere Verarbeitung verworfen.

Wichtig ist also zu beachten, dass die Reihenfolgen der einzelnen Sektionen zwar beliebig ist, jedoch können die Abschnitte für Stellen, Transitionen, Kanten und der Endmarkierung beliebig häufig angegeben werden. Die Sektionen für Namen und Beschreibung sind optional, wobei die Definition für den Typ genau einmal auftauchen muss. Die spezielle Sequenz `EOF` besagt, dass nach den Sektionen die Datei auch zu Ende sein soll. Dies ist für aussagekräftigere Fehlermeldungen hilfreich.

Grammatik für beschriftete Transitionssysteme

Analog zu Petri-Netzen ist auch die Grammatik für beschriftete Transitionssysteme mit derselben Syntax im Listing 6.17 angegeben.

```
S = { name | type | description | places | transitions
    | flows | initial_marking | final_markings }, ? EOF ?;
name = '.name', STR;
type = '.type', ('LPN' | 'PN');
description = '.description', (STR | STR_MULTI);
places = '.places', place;
place = | idi, [opts], place;
opts = '[', option, ']';
option = ID, '=', STR, [',', option];
transitions = '.transitions', transition;
transition = | idi, [opts], transition;
flows = '.flows', flow;
flow = | idi, ':', set, '->', set, flow;
set = '{', [objs], '}';
objs = obj, [',', objs];
obj = (INT, '*', idi) | idi;
initial_marking = '.initial_marking', [set];
final_markings = '.final_markings', final_marking;
final_marking = | set, final_marking;
idi = ID | INT;
INT = DIGIT, {DIGIT};
ID = (CHAR | '_'), {CHAR | DIGIT | '_'};
DIGIT = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
CHAR = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
     | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
     | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
       'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
     | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
     | 'u' | 'v' | 'w' | 'x' | 'y' | 'z';
STR = '"', {ALLCHAR - ('"' | '\n' | '\r' | '\t')}, '"';
STR_MULTI = '"', {ALLCHAR - ('"' | '\t')}, '"';
COMMENT = '//', {ALLCHAR - ('\n'|'\r')}, ['\r'], '\n';
        | '/*', {ALLCHAR} - '*/', '*/';
WHITESPACE = ' ' | '\n' | '\r' | '\t';
ALLCHAR = ? Alle zur verfügungstehenden Zeichen ?;
```

Listing 6.16: Grammatik für Petri-Netze mit und ohne Beschriftungen

```
S = {name | type | description | states | labels | arcs}, ? EOF ?;
name = '.name', STR;
type = '.type', 'LTS';
description = '.description', (STR | STR_MULTI);
states = '.states', state;
state = | idi, [opts], state;
opts = '[', option, ']';
option = (ID, '=', STR | 'initial'), [',', option];
labels = '.labels', label;
label = | idi, [opts], label;
arcs = '.arcs', arc;
arc   = | idi, idi, idi, arc;
idi = ID | INT;
INT = DIGIT, {DIGIT};
ID = (CHAR | '_'), {CHAR | DIGIT | '_'};
DIGIT = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
CHAR = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
     | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
     | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
       'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
     | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
     | 'u' | 'v' | 'w' | 'x' | 'y' | 'z';
STR = '"', {ALLCHAR - ('"' | '\n' | '\r' | '\t')}, '"';
STR_MULTI = '"', {ALLCHAR - ('"' | '\t')}, '"';
COMMENT = '//', {ALLCHAR - ('\n'|'\r')}, ['\r'], '\n';
        | '/*', {ALLCHAR} - '*/', '*/';
WHITESPACE = ' ' | '\n' | '\r' | '\t';
ALLCHAR = ? Alle zur verfügungstehenden Zeichen ?;
```

Listing 6.17: Grammatik für beschriftete Transitionssysteme

Hierbei dürfen die Sektionen für den Namen und die Beschreibung nur null- oder einmal auftreten, der Typ muss genau einmal vorhanden sein und die Bereiche für States, Labels und Arc können beliebig häufig angegeben werden.

6.4.2 Renderer

Die Renderer bilden das Gegenstück zu den im vorherigen Abschnitt vorgestellten Parsern. Das heißt, dass das Petri-Netz beziehungsweise Transitionssystem in dem jeweiligen Format in eine Datei geschrieben wird.

Hierzu werden die Daten einzeln zeilenweise herausgeschrieben. Dies geschieht entweder direkt über den von Java zur Verfügung gestellten `StringBuilder` – wie zum Beispiel im Fall des Synet-Renderers – oder wie im Fall der APT-Renderer und des LoLA-Renderers mithilfe der Bibliothek `StringTemplates`[15]. Dazu kann eine Vorlagendatei erstellt werden, die die Grundstruktur des Formates definiert und sich damit anschließend leicht ganze Listen von Daten in das richtige Format schreiben lassen.

Die Verwendung der einzelnen Renderer ist beispielhaft in Listing 6.18 aufgeführt.

```
PetriNet pn;
TransitionSystem ts;
APTRenderer renderer = new APTRenderer();
// Das Petri-Netz im APT-Format in einen String schreiben.
String content = renderer.render(pn);
// Das Transitionssystem im APT-Format in einen String schreiben.
content = renderer.render(ts);
```

Listing 6.18: Renderer für Petri-Netze und Transitionssysteme

Die anderen Renderer lassen sich ganz analog aufrufen, wobei für den PNML- und den LoLA-Renderer nur die Funktion für Petri-Netze definiert ist.

6.4.3 Parser und Renderer für Petrify

Damit die von Petrify erzeugten Petri-Netze auch in anderen Modulen funktionieren, wurden Parser geschrieben, die die Petri-Netze in die Datenstrukturen überführen.

[15] `http://www.stringtemplate.org/`, zuletzt aufgerufen am 26. 03. 2013

Die Hinrichtung, also die Überführung aus den Datenstrukturen in das Petrify Format, wird vom `PetrifyRenderer` durchgeführt.

Petrify-Parser

In Abschnitt 5.2.2 existiert bereits ein Beispiel zum Format von Petrify, der Parser liest die Datei Zeile für Zeile ein und reagiert auf folgende Schlüsselwörter:

- .inputs
- .marking

Die Labels, die auf `.inputs` folgen, werden in Transitionen innerhalb der Datenstrukturen umgewandelt. Der Teil innerhalb der geschweiften Klammern wird von `.markings` eingeleitet und beschreibt welche Stellen markiert werden sollen. Beispielsweise besagt `<c,a>`, dass die Stelle zwischen der Transition `c` und `a` einen Token enthalten soll.

Beinhaltet die Zeile folgende Schlüsselwörter:

- #
- .graph
- .end
- .model

Dann reagiert der Parser überhaupt nicht, ignoriert die Zeile und geht zur nächsten über. Enthält die Zeile keines der Schlüsselwörter und auch keinen Punkt, geht der Parser von davon aus in der eingelesenen Zeile Transitionen zu lesen. Die erste Transition ist der Vorbereich der folgenden Transitionen und wird dementsprechend in die Datenstrukturen übertragen.

Intern kann der Parser wie folgt aufgerufen werden:

```
PetriNet pn_;
PetrifyPNParser ps = new PetrifyPNParser();
ps.parse(net);
pn_ = ps.getPN();
```

Listing 6.19: Parser für Petri-Netze

Petrify-Renderer

Der Petrify-Renderer hat im Gegensatz zum Parser zwei Konstruktoren und kann sowohl Petri-Netze, als auch Transitionssysteme rendern. Dabei unterscheidet sich ein Transitionssystem von einem Petri-Netz nur durch die Zeile `.state graph` anstelle von `.graph`. Der Renderer ist das Gegenstück zum Parser und überträgt die Datenstrukturen in das Petrify-Format. Dabei geht der Renderer methodisch vor und baut die Datei Stück für Stück auf, angefangen vom Namen bis hin zum Abschließen der Datei mit `.end`. Der Renderer beachtet auch, dass keine Netze gerendert werden können, die mehr als einen Token in der Markierung haben.

Der Renderer hat als Ausgabe einen String, das Petri-Netz bzw. Transitionssystem im Petrify-Format und kann wie folgt benutzt werden:

```
PetriNet pn = (erstelltes/eingelesenes Petri-Netz/Transitionssystem im
    APT-Format);
PetrifyRenderer renderer = new PetrifyRenderer();
String pn = renderer.render(pn);
```

Listing 6.20: Renderer für Petri-Netze und Transitionssysteme

6.4.4 Parser und Renderer für Synet

Genau wie Petrify speichert auch Synet Transitionssysteme und Petri-Netze in einem eigenen Format. Es wird daher ein weiterer Parser benötigt, um Ausgaben von Synet

weiter verwenden zu können. Beispiele für die Struktur des von Synet verwendeten Dateiformats befinden sich im Abschnitt 5.2.1.

Synet-Parser

Der Parser für die von Synet verwendeten Ausgabeformate arbeitet analog zum weiter oben beschriebenen APT-Parser und verwendet die im Abschnitt 5.2.1 aufgeführten Grammatiken. Die Verwendung des Parsers für Petri-Netze wird in Listing 6.21 veranschaulicht, der Parser für Transitionssysteme funktioniert auf die gleiche Weise, er wird aber durch `new SynetLTSParser()` erzeugt.

```
PetriNet pn;
PetrifyPNParser ps = new SynetPNParser();
ps.parse(net);
pn = ps.getPN();
```

Listing 6.21: Parser für Petri-Netze

Synet-Renderer

Der Synet-Renderer gibt Petri-Netze und Transitionssysteme als *String* in einem für Synet lesbaren Format aus. Das Listing 6.22 zeigt ein Beispiel für die Verwendung des Synet-Renderers.

```
PetriNet pn = (erstelltes/eingelesenes Petri-Netz/Transitionssystem im
    APT-Format);
PetrifyRenderer renderer = new SynetRenderer();
String pn = renderer.render(pn);
```

Listing 6.22: Renderer für Petri-Netze und Transitionssysteme

7 Analyse-Algorithmen

In diesem Kapitel werden zu allen funktionalen Anforderungen Angaben über ihre Umsetzung gemacht. Einsortiert in die Abschnitte „Graphentheoretische Algorithmen“,„Algorithmen für beschriftete Transitionssysteme“,„Algorithmen für Petri-Netze“ und „Algorithmen für beschriftete Petri-Netze“, wird für jede funktionale Anforderung die Speicher- und Zeitkomplexität angegeben und gegebenenfalls der Algorithmus erläutert, wenn dies notwendig erscheint. Als notwendig wird die Beschreibung eines Algorithmus erachtet, wenn die Implementierung des Algorithmus nicht aus der Definition ersichtlich ist. Die Speicher- und Zeitkomplexität bezieht sich immer auf den eigentlichen Algorithmus, d. h. wenn es Vorbedingungen gibt, auf die getestet wird, wird deren Komplexität nicht in die Berechnungen miteinbezogen.

7.1 Graphentheoretische Algorithmen

In diesem Abschnitt werden die funktionalen Anforderungen betrachtet, die sich auf einen Graphen und damit sowohl auf beschriftete Transitionssysteme als auch auf Petri-Netze mit und ohne Beschriftungen beziehen.

7.1.1 Algorithmen zum Graphzusammenhang

Dieser Abschnitt stellt die verwendeten Algorithmen vor, die den Zusammenhang von Graphen untersuchen. Dies sind insgesamt die Anforderungen A10, A11, B5, B6, B7, B25, B26 und B27 aus Abschnitt 4.1. Die Aussagen zur Komplexität beziehen sich auf einen Graphen mit n Knoten und m Kanten.

Der Algorithmus zur Bestimmung isolierter Elemente überprüft alle Knoten des Graphen in $\mathcal{O}(n)$, also mit linearem Aufwand in Bezug auf die Anzahl der Knoten. Dies ist ohne zusätzlichen Speicherbedarf möglich.

Der Algorithmus für die schwachen Zusammenhangskomponenten ist ähnlich simpel. Alle noch nicht behandelten Knoten werden gespeichert. Solange diese Liste nicht leer ist, wird ein Knoten gewählt und seine Komponente bestimmt. Hierfür wird rekursiv jeder Knoten im Vor- und Nachbereich besucht. Damit der Algorithmus nicht in Endlosschleifen gerät, werden Knoten, die nicht mehr in der Liste der noch nicht behandelten Knoten enthalten sind, nicht weiter behandelt. Dieser Algorithmus hat eine Zeitkomplexität von $\mathcal{O}(n + m)$.

Da diese Knoten in derselben Komponente wie der gerade behandelte Knoten sind, muss einer der aktiven rekursiven Aufrufe diesen Knoten behandeln. Dieser Algorithmus berechnet die schwachen Zusammenhangskomponenten also korrekt.

Um starke Zusammenhangskomponenten zu bestimmen, wird der *Algorithmus von Tarjan* nach [Wil12] eingesetzt. Dieser Algorithmus führt eine Tiefensuche durch, die in einem zufällig gewähltem Knoten beginnt. Hierbei werden für jeden Knoten zwei Zahlen bestimmt. Eine Tiefensuchenummer nummeriert die Knoten fortlaufend in der Reihenfolge, in der sie besucht werden, durch. Zusätzlich gibt es eine *MinNum*, die die kleinste Tiefensuchenummer angibt, die durch die Tiefensuche von diesem Knoten aus erreicht wird. Dies wird wiederholt, bis alle Knoten des Graphen besucht sind.

Während der Tiefensuche wird ein Stack mit den besuchten Knoten aufgebaut. Wenn ein Knoten, nachdem er behandelt wurde, die gleiche Tiefensuchenummer und *MinNum* hat, dann wurde eine starke Zusammenhangskomponente erkannt. Alle Knoten oberhalb des aktuellen Knoten auf dem Stack bilden nun eine Zusammenhangskomponente.

Die Laufzeit des Algorithmus liegt in $\mathcal{O}(n + m)$.

Beide Algorithmen zur Bestimmung von Zusammenhangskomponenten haben einen linearen Speicherbedarf.

Tests

Die Implementierung der drei genannten Algorithmen besteht aus sieben Methoden, die getestet werden. Zu jedem Graph wird zunächst geprüft, ob die korrekte Menge an isolierten Elemente und die korrekten schwachen und starken Zusammenhangskomponenten bestimmt werden. Außerdem werden noch Methoden getestet, die nur prüfen, ob ein Graph stark beziehungsweise schwach zusammenhängend ist. Diese beiden letzten Prüfungen werden zusätzlich noch mithilfe des Modulsystems durchgeführt. Hierbei handelt es sich zwar eigentlich nicht um einen Unit-Test im engeren Sinne, jedoch wurde ein Fehler im zugehörigen Modul bei der Behandlung des leeren Petri-Netzes gefunden, weswegen diese zusätzliche Tests für nötig erachtet wurden.

Außer dem üblichen leeren Petri-Netz werden als weiterere Randfälle Petri-Netze getestet, die nur aus isolierten Elementen bestehen. Hierbei gibt es ein Petri-Netz, dass nur eine einzelne isolierte Stelle hat, aber auch ein Petri-Netz mit mehreren isolierten Stellen. Zusätzlich gibt es noch einige Tests der Zusammenhangskomponenten, wobei hier Transitionssysteme statt Petri-Netze als Graph eingesetzt werden. Schließlich werden einige Petri-Netz-Generatoren verwendet, um Graphen für Tests zu erhalten.

7.2 Algorithmen für beschriftete Transitionssysteme

In diesem Abschnitt wird die Umsetzung der funktionalen Anforderungen betrachtet, die sich auf beschriftete Transitionssysteme beziehen. Dabei werden zunächst die funktionalen Anforderungen im Abschnitt „Direkte Algorithmen“ aufgeführt, die keiner weiteren Erläuterung bezüglich des Algorithmus bedürfen. Anschließend werden alle weiteren funktionalen Anforderungen in Bezug auf beschriftete Transitionssysteme betrachtet.

7.2.1 Direkte Algorithmen

In der Tabelle 7.1 werden die funktionalen Anforderungen in Bezug auf beschriftete Transitionssysteme aufgeführt, bei denen eine ausführlichere Erläuterung des implementierten Algorithmus nicht notwendig ist, da sich das Vorgehen direkt aus der Definition ableiten lässt. Deshalb wird für jede funktionale Anforderung auch noch einmal auf die entsprechenden Definitionen verwiesen. Außerdem wird für jede funktionale Anforderung die Zeit- und Speicherkomplexität angegeben, die der implementierte Algorithmus besitzt.

Um die Komplexitätsspalten nicht zu riesig zu gestalten, werden folgenden Definitionen verwendet:

$|S|$ ist die Größe der Stellenmenge

$|K|$ ist die Größe der Kantenmenge

Name	Definition	Funktionale Anforderung	Zeit-komplexität	Speicher-komplexität
Deterministisch	3.2.4	A2	$\mathcal{O}(\|K\|)$	$\mathcal{O}(\|K\|)$
Total erreichbar	3.2.3	A3	$\mathcal{O}(\|S\|^2)$	$\mathcal{O}(\|S\|^2)$
Persistent	3.2.6	A4	$\mathcal{O}(\|S\|^3)$	$\mathcal{O}(\|S\|^2)$
Reversibel	3.2.5	A5	$\mathcal{O}(\|S\|^3)$	$\mathcal{O}(\|S\|^2)$
Überprüfe, ob alle kleinsten Kreise den gleichen Parikh-Vektor haben	3.2.2	A8	$\mathcal{O}(\|S\|^3 \cdot \|K\|^4)$	$\mathcal{O}(\|S\|^4 \cdot \|K\|^3)$
Überprüfe, ob alle kleinsten Kreise den gleichen oder gegenseitig disjunkten Parikh-Vektor haben	3.2.2	A9	$\mathcal{O}(\|S\|^3 \cdot \|K\|^4)$	$\mathcal{O}(\|S\|^4 \cdot \|K\|^3)$
Berechne die Parikh-Vektoren der kleinsten Kreise	3.2.2	A12	$\mathcal{O}(\|S\|^3 \cdot \|K\|^4)$	$\mathcal{O}(\|S\|^4 \cdot \|K\|^3)$

Tabelle 7.1: Direkte Algorithmen in Bezug auf beschriftete Transitionssysteme

7.2.2 Synthetisieren von Transitionssystemen

In diesem Abschnitt wird die Funktionsweise der Module zum Synthetisieren von Transitionssystemen, das in den Anforderungen A6 und A7 gefordert wurde, erläutert. Die dazu verwendeten Tools wurden in Abschnitt 5.2 vorgestellt.

Synthetisieren mit Petrify

Das Tool Petrify wurde im Abschnitt 5.2.2 beschrieben, auch hier wird nur der Synthesealgorithmus genutzt. Aufgerufen wird das Modul mit:

```
java -jar apt synthesizable <transitionsystem> [output] [dead]
```

Die Angaben in eckigen Klammern sind optionale Parameter, sind im Transitionssystem jedoch Deadlocks (3.2.7) vorhanden, muss der `dead`-Parameter benutzt werden. Der optionale Parameter `output` ist der Name der Datei, in die das Modul das synthetisierte Netz speichert.

Petrify wird von unserem Werkzeug wie folgt aufgerufen:

```
petrify -nolog -p <transitionsystem>
```

Sollte das Transitionssystem Deadlocks beinhalten, gibt Petrify dieses an das Modul weiter und das Modul teilt es dem Nutzer mit. Sollte der Benutzer das Modul mit dem Parameter `dead` aufrufen, startet das Modul Petrify mit:

```
petrify -nolog -p <transitionsystem> -dead
```

Durch den Parameter `-nolog` erstellt Petrify keine Log-Datei, diese wird bei nur einem Aufruf nicht benötigt und Fehler werden direkt vom Modulsystem an den Benutzer weitergegeben. Petrify erzeugt nach StandardOutput eine Ausgabe, welche der Renderer einliest und ins APT-Format konvertiert, somit bekommt der Benutzer die von Petrify erzeugte Ausgabe nicht zu sehen. Die Ausgabe vom Moduls wird entweder

nach StandardOutput geschrieben oder in einer Datei gespeichert.
Der Parameter `-p` sorgt dafür, dass Petrify ein pures Petri-Netz ohne Schleifen erzeugt.

Synthetisieren mit Synet

Das Programm Synet, welches hinter dem Modul steht, wurde in Abschnitt 5.2.1 beschrieben. Für dieses Modul wird nur der Synthesealgorithmus genutzt. Aufgerufen wird das Modul mit:

```
java -jar apt synthesize_distributed_lts <transitionsystem> [output]
```

Die Angabe in der eckigen Klammer ist der optionaler Parameter. Der optionale Parameter `output` ist der Name der Datei, in die gespeichert werden soll.
Intern wird Synet aufgerufen mit:

```
synet -r -o <output> [-d <location-file>] <transitionsystem>
```

Das Modul ruft den Teil innerhalb der eckigen Klammern optional auf, je nachdem, ob das eingegebene Transitionssystem Locations enthält oder nicht. Dieses Modul ist dabei das einzige welches auf das Schlüsselwort `Location` in einer APT-Datei reagiert, es hat einen internen Parser welcher die .dis-Datei aus dem übergebenen LTS generiert. Das von Synet erstellte Transitionssystem, bekommt der Nutzer nicht angezeigt. Der Output wird abschließend von einem Renderer ausgelesen und in das APT-Format übersetzt, welches der Nutzer entweder angezeigt bekommt oder das Modul in einer separaten Datei speichert.

In 5.2 werden die Grammatiken beschrieben. In dem Kapitel wird deutlich, dass sowohl Synet als auch Petrify tiefere Kenntnisse der Formate und der Nutzung selbst bedeuten würde. Der Vorteil unseres Moduls ist es, dass diese Formate nie vom Nutzer selbst eingegeben werden, sondern weiterhin das APT-Format verwendet werden kann. Alles was die Programme benötigen, kann und wird daraus generiert.
Ein Vorteil von Synet ist, dass es den Parameter *dead* nicht benötigt, dieser Fall wird von Synet intern behandelt.

Tests und Komplexität

Die Komplexitäten werden von den jeweils benutzten Algorithmen in Synet oder Petrify bestimmt. Die Aufrufe der Programme und das Abspeichern und Auslesen von StandardOutput in eine Datei liegen jedoch alle in $\mathcal{O}(n)$. Die Tests fallen eher minimalistisch aus, da die einzigen Fehler, die entstehen können, auf Benutzerseite liegen. Zum einen muss ein Schreibrecht im Temp-Verzeichnis vorliegen und zum anderen müssen die beiden Programme installiert sein. Das Schreibrecht wird benötigt um Temp-Dateien zu erstellen, in denen die gerenderten Petri-Netze stehen. Sollte eines davon nicht der Fall sein, sind das die einzigen Fälle, in denen etwas beim Aufruf fehlschlagen kann.

7.2.3 Algorithmen zur Erstellung von Erweiterungen von Transitionssystemen

Die Eingabe dieses Moduls ist eine beschriftetes Transitionssystem Z und eine Zahl $g \in \mathbb{N}$, die angibt, um wie viele Knoten t maximal erweitert werden darf. Für jede dieser Erweiterungen Z' wird überprüft, ob sie die folgenden drei Bedingungen erfüllt:

1. Z' ist reversibel

2. Z' ist persistent

3. alle kleinsten Kreise in Z' haben den gleichen Parikh-Vektor

Außerdem wird jede Erweiterung auf ihre Minimalität geprüft. Sie ist minimal, falls keine andere echte Erweiterung des ursprünglichen Systems Z existiert, die ebenfalls die drei obigen Bedingungen erfüllt und ein Teilgraph von Z' ist.

Das Modul bietet außerdem die Option, einen Parikh-Vektor, den sich alle kleinsten Kreise in Z' teilen sollen, explizit vorzugeben.

Repräsentation der Transitionssysteme

Um jede mögliche Erweiterung betrachten zu können, ist es nötig, diese in systematischer Folge zu betrachten. Zu diesem Zweck bietet sich eine Darstellung der Transitionssysteme an, die eine inhärente Reihenfolge enthält. Die Repräsentation, für die wir uns bei der Entwicklung dieses Moduls entschieden haben, ist eine Binärzahl. Diese wird durch die Java-Klasse `BitString` implementiert.

Jeder möglichen Kante einer Erweiterung von Z ist eine feste Position innerhalb des Binärstrings zugeordnet. Damit diese Position p eindeutig berechnet werden kann, müssen zu ihrer Berechnung beide verbundenen Knoten und die Beschriftung der Kante verwendet werden; die Formel für p enthält also drei Variablen s, t, $l \in \mathbb{N}$, die folgendermaßen belegt werden:

- s entspricht der Position der ID des Ursprungsknotens in einer alphabetisch sortierten Liste aller Knoten IDs.
- t entspricht der Position der ID des Zielknotens in der selben Liste.
- l entspricht der Position der Kantenbeschriftung in einer alphabetisch sortierten Liste aller möglichen Kantenbeschriftungen.

In einem Transitionssystem mit n Knoten und einem Alphabet der Größe m liegen die s, t und l also zwischen 0 und $n-1$ beziehungsweise $m-1$. Zur Berechnung von p kann nun die folgende Formel verwendet werden:

$$p = t \cdot n \cdot m + l \cdot n + s$$

Dieses Bit wird genau dann gesetzt, wenn die korrespondierende Kante in Z' vorhanden ist. Diese Darstellung als Binärzahl ermöglicht es, einige nützliche Operationen sehr effizient durchzuführen. Beispielsweise können Transitionssysteme und auch einzelne Teilgraphen bitweise auf Gleichheit verglichen werden.

Iterieren über mögliche Erweiterungen

Beim ersten Aufruf des Moduls erzeugt dieses den Bitstring, der das ursprüngliche Transitionssystem repräsentiert. Die Darstellung jeder später generierten Erweiterung besteht nur aus den Kanten, die für die Erweiterung hinzugefügt werden. Das tatsächliche erweiterte System ergibt sich, indem diese beiden Bitstrings durch das Anwenden eines bitweise ausgeführten OR-Operators zusammengefügt werden. Das Resultat enthält alle gesetzten Bits beider Teile und repräsentiert damit ein Transitionssystem mit den ursprünglichen und den neuen Kanten.

Wie bereits oben erwähnt, sollen sämtliche Erweiterungen getestet werden und es ist deshalb notwendig dies in einer systematischen Reihenfolge zu tun. Das Modul verfügt dafür über eine Methode, die für ein gegebenes Transitionssystem und seine Erweiterung eine nächste Erweiterung generiert. Das hier verwendete Verfahren entspricht grundsätzlich einer Breitensuche über die Anzahl hinzugefügter Transitionen. Das heißt, dass zunächst alle Erweiterungen mit einer festen zusätzlichen Kanten betrachtet werden. Wurden diese vollständig überprüft, so wird die Anzahl hinzugefügter Transitionen inkrementiert und das Verfahren solange fortgesetzt, bis das erweiterte Transitionssystem die maximale Anzahl von Kanten enthält. Auf diese Weise wird jede mögliche Erweiterung getestet.

Erfüllt eines der erweiterten Transitionssysteme die gestellten Bedingungen, so muss es außerdem auf seine Minimalität überprüft werden. Zu diesem Zweck erstellt das Modul eine Liste bereits gefundener minimaler Erweiterungen und stellt fest ob diese Teilgraphen der aktuell betrachteten sind. Ist dies nicht der Fall, so wird die neue Erweiterung der Liste hinzugefügt. Da die Iterationsreihenfolge sicherstellt, dass die Erweiterungen mit der geringsten Anzahl zusätzlicher Kanten zuerst getestet werden, ist garantiert, dass nicht erst zu einem späteren Zeitpunkt eine kleinere Erweiterung gefunden wird.

Die Liste minimaler Erweiterungen wird außerdem vom Modul in eine vom Benutzer anzugebende Datei gespeichert, sodass die Suche zu einem späteren Zeitpunkt fortgesetzt werden kann ohne die bisherigen Ergebnisse zu verlieren.

Komplexität

Da die Anzahl der zu untersuchenden Transitionssysteme exponentiell mit der Anzahl der Knoten und der Größe des verwendeten Alphabets ansteigt, ist der Rechenaufwand dieses Moduls notwendigerweise ebenfalls exponentiell. Genauer existieren zu einer gegebenen Knotenzahl n und einem Alphabet aus m Labels 2^{n^2m} verschiedene Transitionssysteme. Die Zeitkomplexität für das Überprüfen aller Systeme liegt also in $\mathcal{O}(2^{n^2m} \cdot \mathcal{C})$, wobei $\mathcal{C}$ der Aufwand für das Prüfen der gewünschten Eigenschaften ist. Dieser liegt im schlechtesten Fall bei $\mathcal{O}(n^3)$.

Die Speicherkomplexität des Moduls ist hingegen relativ gering, da zu jedem Zeitpunkt nur eine Erweiterung betrachtet wird. Der Speicherbedarf für ein Transitionssystem liegt dabei höchstens in $\mathcal{O}(n^2m)$.

Tests

Die Methoden zum Generieren der jeweils nächsten Erweiterung sowie zur Überprüfung von Gültigkeit und Minimalität wurden mit verschiedenen Randfällen getestet. Die exponentielle Laufzeit des Verfahrens ermöglichte allerdings Tests des kompletten Suchvorgangs nur an sehr kleinen Transitionssystemen.

7.3 Algorithmen für Petri-Netze

In diesem Abschnitt wird die Umsetzung der funktionalen Anforderungen betrachtet, die sich auf Petri-Netze beziehen. Dabei werden zunächst die funktionalen Anforderungen im Abschnitt „Direkte Algorithmen“ aufgeführt, die keiner weiteren Erläuterung bezüglich des Algorithmus bedürfen. Anschließend werden alle weiteren funktionalen Anforderungen in Bezug auf Petri-Netze betrachtet.

7.3.1 Direkte Algorithmen

Analog zu Abschnitt 7.2.1 werden in der Tabelle 7.2 die funktionalen Anforderungen in Bezug auf Petri-Netze aufgeführt, bei denen eine ausführlichere Erläuterung des implementierten Algorithmus nicht notwendig ist, da sich das Vorgehen direkt aus der Definition ableiten lässt.

Um die Komplexitätsspalten nicht zu riesig zu gestalten, werden folgenden Definitionen verwendet:

$|E|$ ist die Größe der Erreichbarkeitsmenge

$|T|$ ist die Größe der Transitionenmenge

$|S|$ ist die Größe der Stellenmenge

$|K|$ ist die Größe der Kantenmenge

$|N|$ ist die Größe der Nebenbedingungenmenge

C_t ist die Zeitkomplexität für die Berechnung des Überdeckungsgraphs

C_p ist die Platzkomplexität für die Berechnung des Überdeckungsgraphs

Name	Definitionen	Funktionale Anforderung	Zeit-komplexität	Speicher-komplexität
Schlicht	3.4.1	B2	$\mathcal{O}(\|S\|)$	$\mathcal{O}(1)$
Pur	3.4.2	B3	$\mathcal{O}(\|S\|)$	$\mathcal{O}(1)$
Nicht pur aber nur mit einfachen Nebenbedingungen	3.4.2 3.3.2	B4	$\mathcal{O}(\|S\| \cdot \|K\| + \|N\|)$	$\mathcal{O}(N)$
T-Netz	3.4.13	B10	$\mathcal{O}(\|S\|)$	$\mathcal{O}(1)$
S-Netz	3.4.12	B11	$\mathcal{O}(\|T\|)$	$\mathcal{O}(1)$
Free-choice	3.4.14	B12	$\mathcal{O}(\|S\| \cdot \|T\|^2)$	$\mathcal{O}(1)$
Output-nonbranching	3.4.17	B14	$\mathcal{O}(\|S\|)$	$\mathcal{O}(1)$

Name	Definitionen	Funktionale Anforderung	Zeit-komplexität	Speicher-komplexität
konfliktfrei	3.4.18	B15	$\mathcal{O}(\|S\| \cdot \|T\|)$	$\mathcal{O}(1)$
restricted free-choice	3.4.15	B16	$\mathcal{O}(\|S\| \cdot \|T\|^2)$	$\mathcal{O}(1)$
BCF	3.4.19	B18	$\mathcal{O}(\|E\|^2 \cdot \|T\|^2 + C_t)$	$\mathcal{O}(\|T\| + C_p)$
BiCF	3.4.20	B19	$\mathcal{O}(\|E\|^3 \cdot \|T\|^2 + C_t)$	$\mathcal{O}(\|T\| + C_p)$
Persistent	3.4.21	B20	$\mathcal{O}(\|E\|^3 + C_t)$	$\mathcal{O}(\|E\|^2 + C_p)$
Reversibel	3.4.4	B21	$\mathcal{O}(\|E\|^3 + C_t)$	$\mathcal{O}(\|E\|^2 + C_p)$
Kreiseigenschaften wie bei LTS	3.2.2	B22	$\mathcal{O}(\|S\|^3 \cdot \|K\|^4 + C_t)$	$\mathcal{O}(\|S\|^3 \cdot \|K\|^4 + C_p)$
Rückwärtsmatrix	3.3.1	B28	$\mathcal{O}(\|S\| \cdot \|T\|)$	$\mathcal{O}(\|S\| \cdot \|T\|)$
Vorwärtsmatrix	3.3.1	B28	$\mathcal{O}(\|S\| \cdot \|T\|)$	$\mathcal{O}(\|S\| \cdot \|T\|)$
Inzidenzmatrix	3.3.8	B28	$\mathcal{O}(2 \cdot (\|S\| \cdot \|T\|))$	$\mathcal{O}(2 \cdot (\|S\| \cdot \|T\|))$
Nebenbedingungen	3.3.2	B29	$\mathcal{O}(\|S\| \cdot \|K\|)$	$\mathcal{O}(N)$
Berechne das größte k, für das M_0 eine k-Markierung ist	3.4.22	B33	$\mathcal{O}(\|S\|)$	$\mathcal{O}(\|S\|)$

Tabelle 7.2: Direkte Algorithmen in Bezug auf Petri-Netze

7.3.2 Algorithmus zur Berechnung des Überdeckungsgraphen

Die Berechnung des Überdeckungsgraphen erfolgt durch den in Listing 7.1 dargestellten Algorithmus. Der Überdeckungsgraph eines Petri-Netzes ist nicht eindeutig, da die Reihenfolge in *unvisited* aufgrund der gewählten Realisierung nicht stabil ist. Bei dieser Variante ergibt sich ein weiterer Nichtdeterminismus dadurch, dass nur der Weg, auf dem ein Zustand erreicht wurde, auf der Suche nach Überdeckungen genutzt wird.

In einem ersten Versuch wurde der Algorithmus als rekursive Tiefensuche durch den

Erreichbarkeitsgraph implementiert. Dies hatte den Vorteil, dass es einfach zu implementieren war, jedoch wurde bei größeren Petri-Netzen die maximal erlaubte Rekursionstiefe überschritten und nur eine Fehlermeldung produziert. Daher wurde auf eine iterative Suche umgestellt, die eine Warteschlange der noch nicht behandelten Knoten enthält.

Die finale Version ist lazy implementiert. Dies bedeutet, dass über ein `Iterator`-Interface bereits auf manche Teile des Überdeckungsgraphen zugegriffen werden kann, bevor der Überdeckungsgraph vollständig berechnet wurde. Hierzu werden nur bei Bedarf neue Knoten des Überdeckungsgraphen berechnet. So wird beispielsweise beim Prüfen der Beschränktheit eines Petri-Netzes nur der Überdeckungsgraph bis zur ersten unbeschränkten Stelle berechnet, was einen enormen Zeitvorteil nach sich ziehen kann.

Leider benötigen die meisten Algorithmen für ihre Analysen vollständige Überdeckungsgraphen, weshalb eine Methode bereitgestellt wird, die den vollständigen Graphen liefert. Beispielsweise wird die Lebendigkeit von Transitionen durch Graph-Zusammenhang untersucht, wofür eingehende Kanten von Knoten benötigt werden. Diese können nicht lazy bestimmt werden.

Für diesen Algorithmus gibt es keine schöne Komplexitätsangabe, da er nicht einmal in EXPSPACE und damit auch nicht in EXPTIME liegt.

Tests

Wie für alle anderen Teile der Implementierung gibt es auch für den Überdeckungsgraph Unit-Tests. Hierbei wurden natürlich einige Randfälle betrachtet, wie zum Beispiel leere Netze oder Netze, die nur aus einer einzelnen Stelle oder Transition bestehen.

Ein besonderer Randfall, der erst spät gefunden wurde, betrifft Mehrfachkanten im Überdeckungsgraph von beschrifteten Petri-Netzen. Dieser Randfall wird in Abbildung 7.1 dargestellt. Hierbei gibt es zwei Transitionen, die beide die gleiche Kante im Erreichbarkeitsgraphen erzeugen. Da die Implementierung eine Zuordnung von den

```
V := ∅
unvisited := M_0
E := ∅
while unvisited ≠ ∅
    wähle ein M̂ ∈ unvisited
    unvisited := unvisited ∖ {M̂}
    foreach t ∈ T
        if M̂[t⟩
            berechne M̃ ∈ (ℕ_ω)^S mit M̂[t⟩M̃
            foreach M ∈ (ℕ_ω)^S auf dem Weg von M_0 nach M̂, auf dem M̂ ursprü
                nglich erreicht wurde
                if ∃s' ∈ S: M(s') < M̃(s') < ω ∧ ∀s'' ∈ S ∖ {s'}: M(s'') ≤ M̃(s'')
                    foreach s ∈ S mit M(s) < M̃(s)
                        M̃(s) := ω
                    endforeach
                    break
                fi
            endforeach
            if M̃ ∉ V
                V := V ∪ {M̃}
                unvisited := unvisited ∪ {M̃}
            fi
            E := E ∪ {(M̂, t, M̃)}
        fi
    endforeach
endwhile
```

Listing 7.1: Algorithmus zur Konstruktion des Überdeckungsgraphen

Abbildung 7.1: Ein Petri-Netz und sein Erreichbarkeitsgraph. Allerdings gibt es zwei Transitionen, die beide zur selben Kante im Erreichbarkeitsgraphen gehören. Dieser Randfall muss in der Implementierung berücksichtigt werden.

Kanten des Erreichbarkeitsgraphen zu den zugehörigen Transitionen erlaubt, führte diese Situation zu Problemen. In der finalen Version der Implementierung wird in diesem Fall die zweite, identische Kante einfach ignoriert. Dies bedeutet insbesondere, dass die Zuordnung zu Transitionen nicht deterministisch ist.

Dieser Algorithmus liefert im Gegensatz zu anderen Algorithmen kein einfach zu überprüfendes Ergebnis. In den Tests muss geprüft weden, ob zu einem Petri-Netz ein korrekter Überdeckungsgraph erzeugt wird. Hierfür wurde beim Schreiben der Tests für jedes für die Tests verwendete Petri-Netz von Hand der zugehörige Überdeckungsgraph bestimmt. Die Tests prüfen also nicht, ob der Überdeckungsgraph ein korrekter Überdeckungsgraph ist, sondern sie prüfen, ob er genau einem bekanntem Überdeckungsgraphen entspricht. Dies ist hier jedoch kein Problem, da der vom implementiertem Algorithmus ermittelte Überdeckungsgraph eindeutig ist.

Bei den Tests ergab sich noch das Problem, wie man prüfen kann, ob ein Überdeckungsgraph einem vorgegebenem entspricht, ohne das die Tests vollkommen unlesbar werden. Hierfür wurde eine externe Bibliothek namens Hamcrest[1] in der Version 1.3 zur Hilfe genommen. Diese bietet ein Matcher-Konzept, dass es ermöglicht, relativ übersichtlich zu beschreiben, auf was für Eigenschaften Objekte geprüft werden sollen. Die von Hamcrest bereitgestellten Matcher wurden von uns um spezialisierte Matcher für unsere Datenstrukturen erweitert, sodass Eigenschaften von diesen geprüft werden können. Ein Test auf einen korrekten Überdeckungsgraphen besteht mit den Matchern aus drei Teilen: Als erstes muss der initiale Zustand stimmen, als zweites müssen die Zustände stimmen und als drittes müssen auch die Zustandsübergänge stimmen. Da die Repräsentation des Überdeckungsgraphen aus relativ vielen Objekten besteht, bei denen manche Eigenschaften, wie zum Beispiel die interne Bezeichung, nicht deterministisch sind, werden Zustände durch Vergleich der Markierung geprüft. Entsprechend erfolgt bei Zustandsübergängen ein Vergleich der Markierungen von

[1] `http://hamcrest.org/JavaHamcrest/`, zuletzt besucht am 26. 03. 2013

Start- und Ziel-Zustand.

Für die Tests wurden nur relativ kleine Petri-Netze verwendet, weil die Überdeckungsgraphen sonst nicht mehr einfach von Hand bestimmbar waren und damit auch die Tests nicht mehr wirklich überprüfbar wären. Da allerdings darauf geachtet wurde, Randfälle abzudecken und auch kleine Petri-Netze nicht triviale Überdeckungsgraphen haben können, ist dies nicht als Lücke in der Testabdeckung anzusehen.

7.3.3 Algorithmus zur Berechnung des Erreichbarkeitsgraphen

Die Bestimmung des Erreichbarkeitsgraphen erfolgt durch den Überdeckungsgraphenalgorithmus, der in Abschnitt 7.3.2 beschrieben wird. Hierbei wird ausgenutzt, dass für beschränkte Netze der Erreichbarkeits- und der Überdeckungsgraph identisch sind. Anschließend wird geprüft, ob der berechnete Überdeckungsgraph eine Markierung erreicht, die ein ω enthält. In diesem Fall ist das Petri-Netz unbeschränkt und der Erreichbarkeitsgraph ist unendlich groß. Dies bedeutet insbesondere nach Definition 3.3.9, dass die Menge der Zustände unendlich ist. Daher kann in diesem Fall kein vollständiger Erreichbarkeitsgraph bestimmt werden.

7.3.4 Tests

Da die Bestimmung des Erreichbarkeitsgraohen durch den Überdeckungsgraphenalgorithmus erfolgt, gibt es keinen Erreichbarkeitsgraphen-Algorithmus, der getestet werden könnte.

7.3.5 Algorithmen zu den Eigenschaften *beschränkt* und *sicher*

Als nächstes soll kurz der Algorithmus zu den Anforderungen B16 und B17 von Seite 38 vorgestellt werden. Diese Eigenschaften wurden in Definition 3.4.5 und 3.4.6 auf Seite 27 eingeführt.

Der Algorithmus verwendet den Überdeckungsgraphen, um den benötigten Teil der Erreichbarkeitsmenge zu berechnen. Sobald eine unbeschränkte Stelle gefunden wurde, kann die Berechnung beendet werden. Für unbeschränkte Netze wird also nicht der gesamte Überdeckungsgraph benötigt.

Zu jeder im Graphen erreichbaren Markierung wird die größte Tokenanzahl bestimmt, die diese auf eine Stelle legt. Falls ein ω auftritt, ist das untersuchte Petri-Netz unbeschränkt. Somit hat dieser Algorithmus konstanten Speicherbedarf und eine Laufzeitkomplexität von $\mathcal{O}(|E| \cdot |S|)$ für ein Petri-Netz mit Stellenmenge S und Erreichbarkeitsmenge E. Allerdings wir noch zusätzliche Zeit und zusätzlicher Speicher benötigt, um den Überdeckungsgraph zu bestimmen.

Zusätzlich zur höchsten erreichbaren Tokenzahl liefert der Algorithmus noch die Stelle, auf der diese Tokenanzahl auftritt, und eine Feuersequenz, mit der diese Markierung erreicht wird. Die Feuersequenz wird durch den Algorithmus zur Bestimmung des Überdeckungsgraphen geliefert.

Um zu prüfen, ob ein Petri-Netz sicher ist, wird es auf 1-Beschränktheit untersucht.

Tests

Für eine gute Testabdeckung dieser Implementierung werden nur wenige Unit-Tests benötigt, da die existierende und separat getestete Implementierung des Überdeckungsgraphen eingesetzt werden kann. Zunächst wurden verschiedene Randfälle implementiert. Diese sind ein leeres Petri-Netz, eine einzelne, isolierte Stelle und analog eine isolierte Transition. Außerdem werden zwei verschiedene unbeschränkte Netze und diverse k-beschränkte Petri-Netze für verschiedene k getestet.

7.3.6 Algorithmen zur Lebendigkeit

Es werden drei Arten von Lebendigkeit untersucht, die in Definition 3.4.7 nachgelesen werden können und der funktionalen Anforderung B23 entsprechen. Für schwache und

starke Lebendigkeit wird der Erreichbarkeitsgraph untersucht, womit nur beschränkte Petri-Netze als Eingabe erlaubt sind.

Im Folgenden wird beschrieben, wie die Lebendigkeit einer einzelnen Transition untersucht wird. Falls untersucht werden soll, ob ein Petri-Netz lebendig ist, wird jede Transition des Netzes getrennt untersucht.

Einfache Lebendigkeit

Wenn eine Transition einfach lebendig ist, dann gibt es im Überdeckungsgraph mindestens eine Kante zu dieser Transition. Die Implementierung erzeugt also den Überdeckungsgraphen und prüft alle Kanten, ob sie zur gesuchten Transition gehören.

Da hierbei nur alle Kanten des Graphen benötigt werden, muss nicht der ganze Überdeckungsgraph bestimmt werden, sondern die Berechnung wird abgebrochen, sobald eine passende Kante gefunden wurde.

Hierfür wird lineare Zeit in der Anzahl der Kanten des Überdeckungsgraphen benötigt und nur konstant viel Speicher.

Schwache Lebendigkeit

Für die Untersuchung von schwacher Lebendigkeit werden graphentheoretische Überlegungen auf den Erreichbarkeitsgraphen angewendet.

Da das zu untersuchende Petri-Netz beschränkt ist, gibt es nur endlich viele erreichbare Markierungen und eine unendliche Feuersequenz muss Markierungen mehrfach besuchen. Somit gibt es im Erreichbarkeitsgraphen eine Kante, die mit der gesuchten Transitionen beschriftet ist und deren beiden Knoten in derselben starken Zusammenhangskomponente liegen.

Wenn umgekehrt die beiden Knoten einer Kante in derselben starken Zusammen-

hangskomponente liegen, dann muss die zugehörige Transitionen schwach lebendig sein. Da der Erreichbarkeitsgraph untersucht wird, müssen beide Markierungen erreichbar sein. Da ihre Knoten in der gleichen Zusammenhangskomponente liegen, gibt es einen Weg zurück zum Anfang der Kante. Somit erhält man eine unendliche Feuersequenz, die die gewünschte Transition unendlich oft enthält.

Der Algorithmus, der die schwache Lebendigkeit prüft, erzeugt also die starken Zusammenhangskomponenten des Erreichbarkeitsgraphen und prüft, ob es eine Kante gibt, die zu der gesuchten Transition gehört und deren beiden Knoten in der gleichen Zusammenhangskomponenten liegen. Dies hat eine Laufzeitkomplexität von $\mathcal{O}(|E|^2 \cdot |T|)$. Dieser Algorithmus hat selbst nur konstanten Speicherbedarf, baut aber auf andere Algorithmen mit hohem Speicherbedarf auf. Der Algorithmus für die Zusammenhangskomponenten wird in Abschnitt 7.1.1 beschrieben. Leider muss hierfür der vollständige Überdeckungsgraph bestimmt werden.

Starke Lebendigkeit

Auch für die starke Lebendigkeit wird der Erreichbarkeitsgraph untersucht.

Zunächst wird eine Menge E mit allen Knoten des Graphen erstellt. Nun werden alle Kanten des Graphen betrachtet, die zu der zu untersuchenden Transition gehören. Der Vorbereich des Ausgangsknoten der Kante wird nun rekursiv aus E entfernt. Um Endlosschleifen zu vermeiden, wird die Rekursion abgebrochen, falls ein Knoten erreicht wird, der schon aus E entfernt wurde. Es werden also alle Knoten entfernt, die eine Kante mit passender Beschriftung erreichen können.

Die betrachtete Transition ist genau dann stark lebendig, wenn E anschließend die leere Menge ist, da alle Markierungen entfernt wurden, von denen aus irgendwann die betrachtete Transition gefeuert werden könnte.

Die Laufzeitkomplexität dieses Ansatzes liegt in $\mathcal{O}(|T|^2 \cdot |E|^2)$ und die Speicherkomplexität ist in $\mathcal{O}(1)$. Auch für diesen Ansatz muss der Überdeckungsgraph vollständig berechnet werden.

Tests

Da Lebendigkeit den Überdeckungs- bzw. Erreichbarkeitsgraphen benutzt und dieser separat getestet wird, müssen die Tests nur bezüglich der Lebendigkeitsprüfung eine gute Testabdeckung erreichen. Da dieser Code jedoch relativ einfach ist, reicht dafür ein relativ üblicher Satz an Netzen, für den die verschiedenen Arten von Lebendigkeit von Hand untersucht wurden. Die Tests vergleichen, ob die Implementierung zu den gleichen Ergebnissen kommt.

7.3.7 Algorithmen zur Separierbarkeit

In diesem Abschnitt soll der Algorithmus für die starke und die schwache k-Separierbarkeit gemäß der Anforderung B24 vorgestellt werden. Die formalen Definitionen können unter 3.4.23 und 3.4.24 nachgelesen werden. Der Parameter k bestimmt, in wie viele Teile die initiale Markierung des eingegebenen Petri-Netzes separiert wird. Um herauszufinden, ob ein Petri-Netz stark / schwach k-separierbar ist, wird der Erreichbarkeitsgraph untersucht, weshalb nur beschränkte Petri-Netze genutzt werden können.

Ist ein Petri-Netz mit der initialen Markierung $k \cdot M_0$ stark k-separierbar, so kann es durch k identische Petri-Netze mit der initialen Markierung M_0 ersetzt werden, wobei die möglichen feuerbaren Sequenzen erhalten bleiben.

Implementierung

Die Grundidee der Implementierung ist es, den Erreichbarkeitsgraphen einmal für das gegebene Petri-Netz $k \cdot N$ mit der initialen Markierung $k \cdot M_0$ und einmal für das Petri-Netz N, welches dem Petri-Netz $k \cdot N$ gleicht, jedoch die initiale Markierung M_0 besitzt, zu erzeugen. Bei der Überprüfung auf starke Separierbarkeit geht der Algorithmus die möglichen Sequenzen des Petri-Netzes $k \cdot N$ durch und versucht, diese durch Sequenzen aus dem Petri-Netz N zu realisieren. Die Sequenzen aus N können dabei k-mal verwendet werden, wobei zwischen den Sequenzen aus N beliebig

gewechselt werden darf, so lange die Reihenfolge innerhalb einer Sequenz erhalten bleibt.

Bei der schwachen Separierbarkeit gilt vom Grundprinzip her das Gleiche, jedoch darf hier die Reihenfolge innerhalb einer Sequenz beliebig getauscht werden.

Ein Petri-Netz kann u. U. unendlich lange und unendlich viele verschiedene Sequenzen feuern, daher würde es unendlich lange dauern, alle möglichen Sequenzen zu überprüfen. Aus diesem Grund kann der Algorithmus entweder gezielt eine gegebene Sequenz oder alle Sequenzen bis hin zu einer gegebenen Länge auf k-Separierbarkeit testen.

Der Algorithmus testet neben der k-Separierbarkeit für das angegebene k auch die d-Separierbarkeit für jeden Divisor d von k. Wird kein k angegeben, so wird das maximal mögliche k gewählt, welches der größte gemeinsame Teiler aller Token ist.

Laufzeitkomplexität

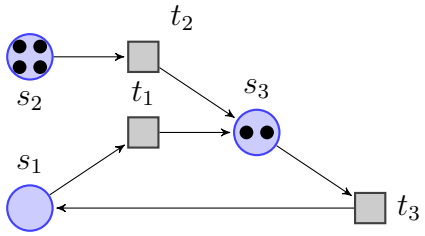


Abbildung 7.2: Beispiel-Petri-Netz $k \cdot N$

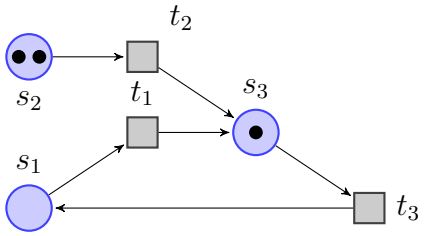


Abbildung 7.3: Beispiel-Petri-Netz N

Die Komplexität soll anhand eines Beispiels veranschaulicht werden. Abbildung 7.2 zeigt das Ausgangs-Petri-Netz. In diesem Beispiel gilt „$k = 2$“. Um nun das Petri-Netz N zu berechnen – siehe Abbildung 7.3 – müssen die Token jeder Stelle angepasst werden.

Gewählte Sequenz: „t2; t3; t1; t2“

- Berechnung des Petri-Netzes N.
  Entscheidend dafür ist die Anzahl der Stellen $|S|$.

- Berechnung des Erreichbarkeitsgraphen von N.
  Die Zeitkomplexität für den Erreichbarkeitsgraphen ist C_t.
- Für jeden Schritt der Sequenz gibt es k Möglichkeiten (hier: 2), da eines der k-mal vorhandenen Petri-Netze N gewählt werden muss. Dabei ist die Länge der Sequenz l (hier: 4) und somit die Komplexität für diesen Punkt k^l (hier: 2^4).

Es liegt also eine Komplexität $\mathcal{O}(|S| + C_t + k^l)$ vor.

In der Praxis ist die Komplexität deutlich kleiner, da viele Wege verworfen werden können, weil in ihnen die geforderte Transition nicht gefeuert werden kann. Dazu können verschiedene Wege zusammengefasst werden. So ist zum Beispiel der Weg „N_1, N_2, N_2“ der gleiche wie „N_2, N_1, N_1“, da die Namen der Petri-Netze N beliebig sind. Beide Optimierungsmöglichkeiten werden von dem Algorithmus genutzt. In wie weit die Optimierungsmöglichkeiten zum Tragen kommen, hängt vom konkreten Petri-Netz ab. Der erste Schritt kann jedoch in jedem Fall optimiert werden, so dass nach der Optimierung eine Komplexität $\mathcal{O}(|S| + C_t + k^{(l-1)})$ vorliegt.

Mit maximaler Länge (l_{max}) Wird eine maximale Länge angegeben, so wird jede mögliche Sequenz ($|Seq|$) bis zur maximalen Länge getestet. Dadurch kommt der Faktor $|Seq|$ hinzu. Dazu muss für l der ungünstigste Fall l_{max} angenommen werden.

Daraus ergibt sich die Komplexität zu $\mathcal{O}(|S| + (C_t + k^{(l_{max}-1)}) \cdot |Seq|)$.

Speicherkomplexität

Da sich im ungünstigsten Fall alle Wege gemerkt werden müssen, ist die Speicherkomplexität ähnlich zu der Laufzeitkomplexität. Jedoch fällt $|S|$ weg, da diese Berechnung vorweg geschieht und dabei kein zusätzlicher Speicher benötigt wird. Es wird der Speicherbedarf des Erreichbarkeitsgraphen (C_p) benötigt. Nach der Überprüfung einer Sequenz können die Zwischenergebnisse für diese Sequenz verworfen werden, weshalb der Faktor $|Seq|$ wegfällt.

Somit beträgt die Speicherkomplexität für eine bestimmte Sequenz $\mathcal{O}(C_p + k^{(l-1)})$ und für eine maximale Länge $\mathcal{O}(C_p + k^{(l_{max}-1)})$.

Tests

Für das Separierbarkeits-Modul gibt es insgesamt gut 50 Tests. Darunter sind auch Tests für das zum Separierbarkeits-Paket gehörende Modul, welche das größte k, für das M_0 eine k-Markierung ist, berechnet. Auch gibt es Tests für die Funktion, welche den größten gemeinsamen Teiler zweier Zahlen mithilfe des Euklidischen Algorithmus ermittelt.
Die starke und die schwache k-Separierbarkeit werden mit je 16 verschiedenen Petri-Netzen getestet. Für die Test-Petri-Netze wurde die Klasse „TestNetsForSeparation“ erzeugt. Ziel dieser Petri-Netze ist es, ein möglichst breites Spektrum an verschiedenen Testfällen zu erzeugen. Sie orientieren sich an den Petri-Netzen aus [BW12] und den Petri-Netzen zur Separierbarkeit von Herrn Prof. Dr. Best aus dem Git-Repository. Zu den Test-Petri-Netzen gehören:

- Triviale Petri-Netze
- Nicht k-separierbare Petri-Netze
- k-separierbare Petri-Netze
- Nicht stark k-separierbare, jedoch schwach k-separierbare Petri-Netze
- a-separierbare, nicht b-separierbare, nicht c-separierbare Petri-Netze mit $a \cdot b = c$, sowohl für $a < b$ als auch für $b < a$

7.3.8 Algorithmen zur Berechnung der S- und T-Invarianten

Nach der Definition von S- beziehungsweise T-Invarianten (siehe Definition 3.4.9 bzw. 3.4.10) muss für die Bestimmung von Invarianten das Gleichungssystem $C^T \cdot x = 0$ beziehungsweise $C \cdot x = 0$ ganzzahlig gelöst werden. In der Literatur wird dafür häufig auf den Algorithmus von Farkas [Far02] verwiesen [Por97, Las09, MS81]. Eine spezielle und hier implementierte Variante für S-Invarianten wird in Listing 7.2 dargestellt.

```
D_0 := (C | E_n)
for i := 1 to m do
  for d_1, d_2 Zeilen in D_{i-1} mit sign(d_1(i)) · sign(d_2(i)) == -1 do
    d := | d_2(i) | · d_1 + | d_1(i) | · d_2; /* d(i) = 0 */
    d := d/ggT(d(1), d(2), ..., d(m + n)) /* d "normieren" */
    Füge zu D_{i-1} als letzte Zeile d hinzu
  od
  D_i ergibt sich aus D_{i-1} indem alle Zeilen z mit z(i) ≠ 0 gelöscht werden
od
Lösche die ersten m Spalten von D_m
```

Listing 7.2: Farkas-Algorithmus zur Berechnung von S- bzw. T-Invarianten

Dabei wird zuerst die Inzidenzmatrix C um eine Einheitsmatrix der Größe n von rechts erweitert; wobei n die Anzahl der Zeilen, also der Stellen, ist. Anschließend wird iterativ in m-Schritten – wobei m die Anzahl der Spalten, also der Transitionen, ist – ein ganzzahliger Lösungsraum für das homogene Gleichungssystem bestimmt.

Analog zum Gaußschen Eliminationsverfahren befindet sich das Ergebnis in der erweiterten Hälfte der Matrix. Dabei stellen die Zeilen dieser Matrix ein Erzeugendensystem der Invarianten mit der Besonderheit, dass die maximale Menge von linear unabhängigen Zeilen die minimalen semipositiven S-Invarianten sind, dar. Hierbei ist zu beachten, dass das Herausfiltern der maximalen Menge an linear unabhängigen Zeilen nicht mehr algorithmisch gelöst wird, da dies für die Kosten der Überprüfung von linearer Unabhängigkeit aller Kombinationen einen zu geringen Mehrwert liefert. Ein Erzeugendensystem für die Invarianten reicht in den meisten Fällen aus und sonst ist die Ausgabe auch gut parsebar, sodass sich diese – in den Fällen, wo die minimalen Invarianten nicht direkt abgelesen werden können – auch weiterverarbeiten lässt.

Zur Berechnung von T-Invarianten braucht lediglich die Inzidenzmatrix transponiert werden. Selbstverständlich passen sich damit auch n und m an.

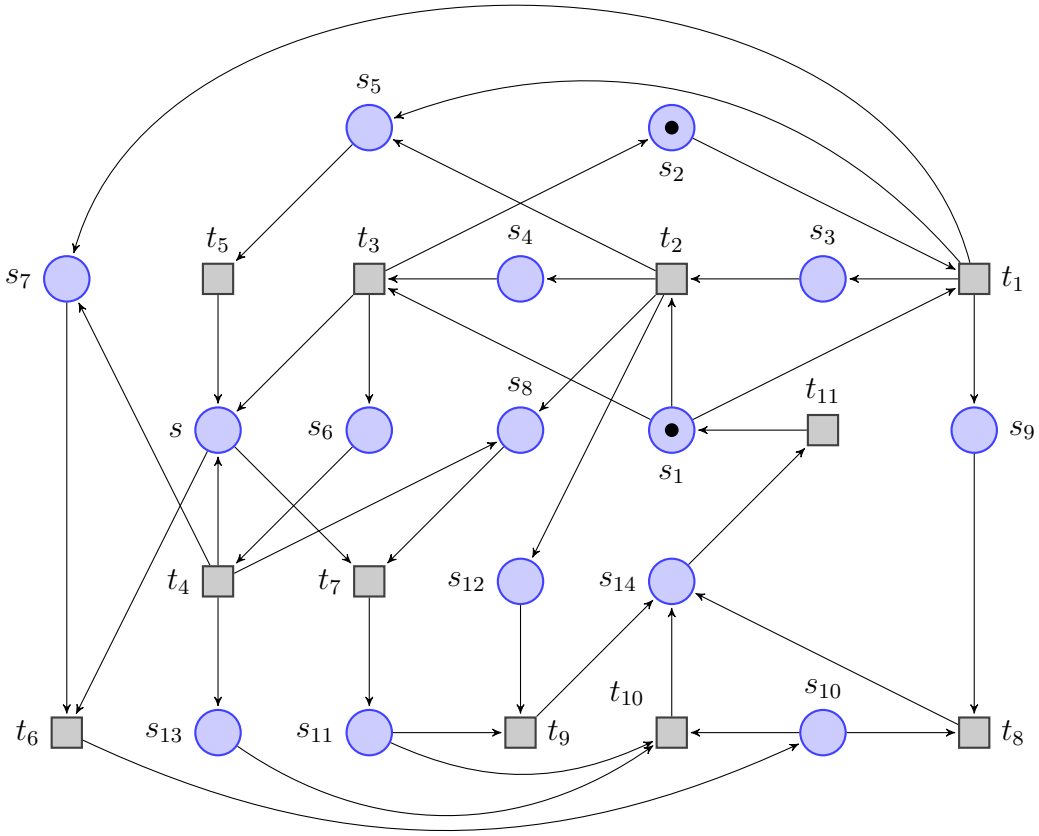


Abbildung 7.4: Petri-Netz mit hoher Zeitkomplexität für Algorithmus aus Listing 7.2

Ein Problem der speziellen Implementierung des Algorithmus aus Listing 7.2 stellt die innere for-Schleife dar. Hier wird die zu untersuchende Matrix in jedem Schleifendurchlauf um eine Zeile erweitert, die auch im nächsten Durchlauf zusätzlich betrachtet werden muss. Dies erschwert die Untersuchung der Komplexität des Algorithmus, da uns die maximale Anzahl an Zeilen, die dieser Matrix hinzugefügt werden können, nicht ersichtlich gewesen ist. Dies zeigt sich auch bei dem Petri-Netz aus Abbildung 7.4. Bei der Berechnung der T-Invarianten müssen ab einem Zeitpunkt 107845 Zeilen untersucht werden; was sehr zeitintensiv ist, aber auch in diesem Fall – nach genügend langer Wartezeit – das korrekte Ergebnis liefert. Bei der Berechnung der S-Invarianten müssen sogar 557025 Zeilen untersucht werden und bei diesem Durchlauf werden so viele weitere Zeilen zum Untersuchen hinzugefügt, dass Java mit einer `OutOfMemoryError`-Exception abstürzt und bis dahin 29 GB Speicher benötigt hat. Dabei wird lediglich die zu untersuchende Matrix gespeichert.

Wegen dieser Problematik ist eine weitere Methode zur Invarianten Berechnung implementiert, welche sich in Listing 7.3 wiederfindet. Das Invariantenmodul bietet bei dem Aufruf die Möglichkeit zwischen beiden Algorithmen zu wählen.

Dieser Algorithmus kommt auch mit dem in Abbildung 7.4 dargestellten Petri-Netz sehr gut zurecht, sodass er in kürzester Zeit die Invarianten berechnen kann. Eine Analyse der Komplexität wurde erneut durch das Problem des Hinzufügens von Spalten zu der Matrix (Phase 2) erschwert. Zumindest lässt sich sagen, dass die Phase 2 exponentielle Zeitkomplexität besitzt[BCC+03].

Auf der Grundlage dieses Algorithmus ließen sich die Anforderungen bezüglich der Invarianten (Anforderungen: B8, B9, B30) lösen.

Ein interessanter Ansatz für weiteres Arbeiten in diesem Bereich ist eine parallelisierte Version mit möglicher Auslagerung auf die Graphikkarte. Hierzu lässt sich der Ansatz für eine Parallelisierung des Algorithmus in dem Papier von Marinescu, Beaven, und Stansifer nachlesen[MBS91].

Tests

Getestet wurde dieses Modul mit allen Netzen aus der *TestNetCollection*-Klasse. Diese umfasst auch Sonderfälle wie das leere Netz, ein Netz mit nur einer Stelle sowie das Netz mit nur einer Transition. Des Weiteren wurden größere Netze mit bis zu 15 Stellen und 11 Transitionen getestet. Insgesamt wurden 26 Netze für dieses Modul in den automatischen Tests geprüft.

7.3.9 Algorithmus zu Fallen und Siphons

In diesem Abschnitt wird der Algorithmus zu den Anforderungen B31 und B32 (siehe Seite 40) beschrieben. Als Grundlagendefinition ist Definition 3.4.16 zu betrachten.

```
C ∈ ℤ^{m×n} // Inzidenzmatrix
B ∈ ℤ^{n×n} // Einheitsmatrix
// Phase 1:
while C hat eine Komponente ungleich 0 do
    Sei P_h^+ = {j | C_hj > 0} und P_h^- = {j | C_hj < 0}
    if eine Zeile h_1 von C mit P_{h_1}^+ = ∅ xor P_{h_1}^- = ∅ existiert then
        lösche alle Spalten aus C und B mit dem Index j ∈ P_{h_1}^+ ∪ P_{h_1}^-
    else
        if eine Zeile h_2 von C mit |P_{h_2}^+| = 1 ∨ |P_{h_2}^-| = 1 existiert then
            Sei k der eindeutige Spaltenindex von P_{h_2}^- bzw. P_{h_2}^+.
            for j ∈ P_{h_2}^- (bzw. j ∈ P_{h_2}^+) do
                Ersetze jede Komponente (Index r) in der Spalte j der
                Matrix C wie auch B mit der Linearkombination:
                C_rj = C_rj · |C_hk| + C_rk · |C_hj|
            od
            Lösche von der Matrix C wie auch B die Spalte
            mit dem Index k.
        else
            Sei h der Index einer Zeile in C, die keine Nullzeile ist
            und sei k der Index der Spalte, sodass C_hk ≠ 0.
            for j ∈ {1,...,n} mit j ≠ k und C_hj ≠ 0 do
                β = |C_hk|
                α = sign(C_hj) · sign(C_hk) < 0 ? |C_hj| : −|C_hj|
                Ersetze jede Komponente (Index r) in der Spalte j der
                Matrix C wie auch B mit der Linearkombination:
                C_rj = C_rj · β + C_rk · α
            od
            Lösche von der Matrix C wie auch B die Spalte
            mit dem Index k.
        fi
    fi
od

// Phase 2:
while B hat eine Zeile h mit negativen Elementen do
    if P_h^+ ≠ ∅ do
        for (j,k) ∈ P_h^+ × P_h^- do
            A = −B_hk · B_j + B_hj · B_k
            B = (B | A)
        od
    fi
    Lösche alle Spalten mit Index x ∈ P_h^- von B
od
Die Invarianten sind die Zeilen von B. Teile jede Zeile durch ihren
ggT, um sie zu Minimalisieren.
```

Listing 7.3: Algorithmus zur Berechnung von S- bzw. T-Invarianten nach [DT88]

Um minimale Fallen und Siphons in Petri-Netzen zu finden, wurde ein iterierter SAT-Algorithmus genutzt, welcher im diesem Abschnitt zugrunde liegenden Papier [NFMS12] gefunden werden kann. Um mit diesem Algorithmus zu arbeiten, wird zunächst zu einem gegebenen Petri-Netz N ein Boolesches Modell entwickelt und in diesem Booleschen Modell sucht ein SAT-Solver direkt nach minimalen Fallen und Siphons.

Fallen im Netz N entsprechen stets Siphons im dualen Petri-Netz N', wobei das duale Petri-Netz N' entsteht, indem alle Kanten aus N umgekehrt werden. Beispielsweise ist das duale Petri-Netz zu $N = \{\{s0, s1\}, \{a\}, \{(s0, a), (a, s1)\}\}$ gegeben durch $N' = \{\{s0, s1\}, \{a\}, \{(s1, a), (a, s0)\}\}$ (Zusätzliche Informationen über duale Petri-Netze: [DE95]). Da somit das Verfahren minimale Fallen zu bestimmen analog zu dem Verfahren minimale Siphons zu bestimmen funktioniert, genügt es in diesem Abschnitt den Algorithmus für Siphons zu betrachten.

Vorgehen

Zunächst wird zu dem gegebenen Petri-Netz ein Boolesches Modell erstellt. Dieses Modell beschreibt Siphons in einem Netz N durch die Zugehörigkeit oder Nicht-Zugehörigkeit jeder Stelle zu einem Siphon D.

Für das Netz N mit n Stellen und m Transitionen kann jedes Siphon D somit durch einen Booleschen Vektor V der Form $\{0, 1\}^n$ dargestellt werden. Hierbei gilt für alle $i \in \{1, 2, \ldots, n\}$ genau dann $V_i = 1$, wenn $s_i \in D$.

Da ein Siphon eine Menge von Stellen ist, für die gilt, dass Stellen aus ihrem Vorbereich ebenfalls im Nachbereich vorkommen, ergibt sich somit folgender Ausdruck:

$$
\begin{aligned}
& \forall i,\, V_i = 1 \implies {}^\bullet s_i \subseteq \left(\bigcup_{V_j = 1} \{s_j\} \right)^\bullet \\
\Leftrightarrow\ & \forall i,\, V_i = 1 \implies \forall t \in T,\, t \in {}^\bullet s_i \implies t \in \left(\left(\bigcup_{V_j = 1} \{s_j\} \right)^\bullet \right) \\
\Leftrightarrow\ & \forall i,\, V_i = 1 \implies (\forall t \in T,\, t \in^\bullet s_i \implies \exists p_j \in^\bullet t,\, V_j = 1)
\end{aligned}
$$

Dieses kann in Klauselform notiert werden als:

$$\forall i,\, V_i = 1 \implies \bigwedge_{t \in {}^\bullet s_i} \left(\bigvee_{s_j \in {}^\bullet t} V_j = 1 \right)$$

Damit die leere Menge als minimales Siphon ausgeschlossen werden kann, wird zudem folgender Ausdruck der Klauselform oben hinzugefügt:

$$\bigvee_i V_i = 1$$

Diese Klauseln sind duale Hornformeln und haben daher jeweils maximal ein negatives Literal.

Über der oben gegebenen Klauselmenge wird nun iteriert. Im ersten Schritt bestimmt der SAT-Solver eine Belegung, mit der die eingegebene Klauselform erfüllt ist. Um sicherzustellen, dass in den folgenden Iterationsschritten nur minimale Siphons gefunden werden, müssen, sobald ein Siphon D gefunden wurde, alle weiteren Obermengen dieses Siphons als potentielle Siphons ausgeschlossen werden. Daher wird folgende Klausel zur vorhandenen Klauselform hinzugefügt:

$$\bigvee_{s_i \in D} V_i = 0.$$

Anschließend wird solange über der in jedem Schritt erweiterten Klauselform iteriert, bis alle möglichen minimalen Siphons ermittelt wurden.

Implementierung

Bei der Implementierung wurden zur Konstruktion des Booleschen Modells von einem Petri-Netz N die Indizes aller Stellen, wie sie zu Beginn des Algorithmus in einer sortierten Liste vorliegen, als Literale gewählt. Es existieren nun zwei Fälle, um die

Konjunktive Normalform aufzustellen:

Minimale Siphons sollen bestimmt werden: Betrachte zu jeder Stelle p im gegebenen Petri-Netz ihren Vorbereich und erstelle eine Klausel, die den negierten Index von p, sowie die nicht-negierten Indizes aller Stellen im Vorbereich von p enthält.

Minimale Fallen sollen bestimmt werden: Betrachte zu jeder Stelle p im gegebenen Petri-Netz ihren Nachbereich und erstelle eine Klausel, die den negierten Index von p, sowie die nicht-negierten Indizes aller Stellen im Nachbereich von p enthält.

Über der hieraus resultierenden konjunktiven Normalform wird anschließend – wie zuvor im Algorithmus beschrieben – iteriert.

SAT-Solver

Als SAT-Solver wird die Java-Bibliothek Sat4j [2] in der Version 2.3.2 verwendet. Sat4j ist eine robuste und flexible Bibliothek, die Open-Source unter der GNU-LGPL-Lizenz verfügbar ist. Weitere Informationen können gefunden werden unter [LBP10].

Komplexität

Die Laufzeit des Algorithmus setzt sich aus zwei Teilen zusammen:

Bestimmung der konjunktiven Normalform: Die Laufzeit dieses Teils ist polynomiell. Dieses ergibt sich daraus, dass über allen n Stellen p im Petri-Netz N iteriert wird und zu jeder Stelle der Vorbereich, welcher im Worst-Case genau n Stellen groß ist, iteriert wird.

SAT-Solver: SAT ist NP-vollständig; Ob auch SAT $\in$ P und damit P = NP gilt,

[2] `http://www.sat4j.org/`, zuletzt aufgerufen am 26. 03. 2013

ist eine der größten ungelösten Fragen der Komplexitätstheorie. Für den hier beschriebenen Algorithmus gilt daher, dass die Untersuchung auf Erfüllbarkeit der Konjunktiven Normalform durch den SAT-Solver in exponentieller Zeit geschieht.

Damit ist die Laufzeitkomplexität dieses Algorithmus exponentiell. Die Speicherplatzkomplexität ist kleiner, da das initiale Boolesche Modell – wie bei der Laufzeitkomplexität schon beschrieben – maximal polynomiellen Speicherplatz benötigt. Die Menge aller gefundenen Fallen und Siphons nähert sich theoretisch im schlechtesten Fall an die Anzahl aller möglichen Teilmengen der Stellenmenge, also an den Wert 2^n, an. Da durch den oben beschriebenen Algorithmus allerdings nur minimale Fallen und Siphons bestimmt werden, ist die Menge aller gefundenen minimalen Fallen und Siphons wesentlich kleiner als 2^n. Ferner ist nicht bekannt, wie viel Speicher der SAT-Solver benötigt.

Tests

Um sicherzustellen, dass das Modul wirklich zu einem beliebigen Petri-Netz minimale Fallen und Siphons bestimmt, wurden einige Testfälle geschrieben. Dabei wurde versucht ausreichend viele interessante Fälle von Beispielnetzen für dieses Modul zu testen.

Die Art der getesteten Netze und die Begründung, warum es sinnvoll ist, diese Netze zu testen, sind der folgenden Auflistung zu entnehmen:

Petri-Netze mit grundlegenden Eigenschaften: Netze mit nur einer Stelle und keiner Transition, keiner Stelle und einer Transition, leere Netze, Netze mit Mehrfachkanten, Netze mit isolierten Elementen, etc.

Petri-Netz mit einer Stelle und einer Schleife: Dieses Netz ist interessant, da die einzige vorhandene Stelle sich selber im Vor-, bzw. im Nachbereich hat. Es musste sichergestellt werden, dass der SAT-Solver hier nicht in einer Endlosschleife hängen bleibt.

(nicht) restricted free choice-Netze: restricted free choice-Netze (siehe Definition 3.4.15) sind für Fallen und Siphons strukturell interessant, da sich im Vorbereich jeder Transition nur eine Stelle befindet (oder auch nicht).

Vollständig (gar nicht) verbunden: Interessant, da im vollständig (gar nicht) verbundenen Netz eine abhängig von der Stellenanzahl maximal große (kleine) konjunktive Normalform entsteht (Da jede Stelle alle anderen Stellen im Vor- und Nachbereich hat, bzw. alle Stellen eine leeren Vor- und Nachbereich haben).

Duale Petri-Netze: Ein Petri-Netz N und sein duales Petri-Netz N', um zu überprüfen, ob Siphons in N wirklich Fallen in N' entsprechen und umgekehrt.

7.3.10 Petri-Netz-Generatoren

Es wurden Generatoren für verschiedene Typen von Petri-Netzen implementiert. Diese Generatoren lassen sich in zwei Arten einteilen. Generatoren der erste Art generieren zu gegebenen Parametern ein einzelnes Petri-Netz. Die zweite Art von Generatoren erzeugt hingegen eine Serie von Petri-Netzen, da die Parameter nur eine Familie beschreiben, die je nach Anzahl der Parameter sehr viele Petri-Netze enthalten kann.

Generatoren für einzelne Petri-Netze

Die Generatoren für einzelne Netze sind als Klassen implementiert, bei denen man eine Methode aufruft, um ein Netz zu generieren. Die Klasse muss instantiiert werden, weil teilweise Vererbung genutzt wird, um zum Beispiel bei den Philosophen die Zustände anzupassen.

Kreis-Generator Der Kreis-Generator generiert ein Petri-Netz, das nur aus einem Kreis der als Parameter angegebenen Größe besteht. Es werden also abwechselnd

Stellen und Transitionen erzeugt, die durch eine einfache Kante verbunden werden. Am Ende wird eine Kante von der letzten Transition zur ersten Stelle angelegt.

n-Bit-Netz-Generatoren Dieser Generator generiert n-Bit-Netze, wie sie in Definition 3.3.13 beschrieben wurden.

Philosophen-Netz-Generatoren In Definition 3.3.14 wurde ein Philosophen-Petri-Netz mit drei Zuständen pro Philosoph vorgestellt. Diese Art von Netzen wird durch den `TristatePhilNetGenerator` umgesetzt.

Diese Definition wurde noch erweitert. Hier gibt es zunächst Philosophen, die nicht beide Gabeln gleichzeitig zurücklegen, sondern ihre Gabeln nacheinander zurücklegen. Dies geschieht in der umgekehrten Reihenfolge, in der die Gabeln genommen wurden. Diese Netzklasse wird durch die `QuadstatePhilNetGenerator`-Klasse erzeugt.

Schließlich gibt es noch den `BistatePhilNetGenerator`. Diese Netzklasse enthält Philosophen, die beide benötigten Gabeln gleichzeitig nehmen und zurücklegen. Dieses Modell ist verklemmungsfrei und unterscheidet sich somit wesentlich von den beiden anderen Arten von Philosophen-Netzen.

Generator für das inverse Netz Dieser Generator zeigt, dass es auch möglich ist, ein ganzes Petri-Netz als Parameter zu nutzen. Er erzeugt aus diesem Netz ein Petri-Netz, bei dem die Richtungen aller Kanten umgekehrt sind.

Generatoren für Petri-Netz-Familien

Diese Klasse von Generatoren generiert mehrere Netze. Die Generatoren sind in Form von *Iteratoren* implementiert, so dass sie erst dann ein weiteres Petri-Netz generieren, wenn eines benötigt wird. Dadurch können die Generatoren schneller die ersten Petri-Netze zurück liefern und verringern den Speicherverbrauch, da sie nur einen internen Zustand, nicht aber alle Petri-Netze, speichern müssen. Solange die Generatoren nicht

direkt als Modul aufgerufen werden, ist ihre Komplexität vernachlässigbar, da diese je generiertem Netz gesehen linear bezüglich der maximalen Anzahlen an Transitionen und Stellen ist, wohingegen Algorithmen, die die Generatoren nutzen, meistens für jedes erzeugte Netz aufwändigere Untersuchungen durchführen. Außerdem wird von den meisten Algorithmen die Generierung abgebrochen, sobald ein geeignetes Netz gefunden wurde, sodass die teilweise sehr stark wachsende Anzahl von Netzen nur beim direkten Aufruf der Generatoren auffällt.

Die folgenden Generatoren erhalten ein Petri-Netz als Eingabe. Zusätzlich gibt es eine weitere Klasse, die einen beliebigen andereren Generator als Eingabe verwendet werden. In diesem Fall werden aus jedem Netz der Eingabe durch den Generator mehrere neue Netze erzeugt. Dies ermöglicht eine höhere Flexibilität als eine direkte Umsetzung der Anforderungen, so dass man die Generatoren besser auf weitere Probleme zuschneiden kann.

Generator für Markierungen Dieser Generator bekommt ein Petri-Netz und eine maximale Anzahl an Token als Parameter. Es werden Kopien des gegebenen Netzes generiert, die um alle möglichen Markierungen, die die gegebene maximale Anzahl an Token nicht überschreitet, erweitert wurden. Sollte das Eingabenetz bereits eine Markierung haben, können die neuen Markierungen wahlweise auch summiert werden, statt die bisherige Markierung zu ersetzen.

T-Netz-Generator Das Herzstück der Generatoren für Petri-Netz-Familien ist der T-Netz-Generator. Die anderen Generatoren können zwar auch ohne diesen genutzt werden, sind jedoch durch die Aufteilung der T-Netz-Generierung in kleinere Teilaufgaben ins Leben gerufen worden. Trotzdem ist der T-Netz-Generator nicht trivial, so dass seine Funktionsweise im Folgenden beschrieben wird. Zunächst ist dafür jedoch festzustellen, was T-Netze eigentlich sind; Definition 3.4.13 beschreibt sie als Netze, bei denen jede Stelle maximal eine Transition im Vor- und Nachbereich hat. In der Literatur findet sich auch eine abweichende Definition, die besagt, dass jeweils genau eine Transition im Vor- und Nachbereich einer Stelle sein soll. Der Generator verwendet die zweite Definition, die auch für E1 und E2 benötigt wird. Für diese Anforderungen müssten ansonsten die zusätzlich erlaubten T-Netze gleich wieder

verworfen werden, nachdem sie generiert wurden.

Ein einfacher Ansatz für den T-Netz-Generator wäre das Generieren mehrerer Stellen, denen anschließend genau eine Transition im Vor- und Nachbereich zugewiesen wird. Hierbei werden Transitionen mehrfach verwendet. Dieser Ansatz erzeugt genau die Petri-Netze, die laut der Definition T-Netze sind. Allerdings ergibt sich das Problem, dass sehr viele der erzeugten T-Netze nicht wesentlich verschieden sein werden. Daher würden die Nutzer dieses Algorithmus viele Netze unnötigerweise weiter untersuchen, wodurch dieser einfache Algorithmus im Zusammenspiel mit anderen Algorithmen, die diesen nutzen, nicht sehr effizient ist.

Der implementierte Algorithmus besteht aus zwei Teilen. Der erste Teil erzeugt Transitionen und legt die Größe ihrer Vorbereiche fest, so dass implizit bereits alle Stellen erzeugt werden. Im zweiten Teil wird jeder Stelle genau eine Transition als Vorbereich zugewiesen, wobei hier zusätzliche Transitionen, die im folgendem Überhangstransitionen genannt werden, erzeugt werden können.

Der erste Teil bekommt als Eingabe die maximale Transitionen- und Stellenanzahl und iteriert über alle aufsteigend geordneten Listen von positiven, ganzen Zahlen, deren Summe die maximale Anzahl an Stellen nicht überschreitet. Wenn für jeden Listeneintrag eine Transition erzeugt wird und in ihrem Vorbereich so viele Stellen enthalten sind, wie die Zahl im Listeneintrag angibt, dann werden damit alle Möglichkeiten von Transitionen mit Stellen in ihren Vorbereichen abgedeckt, ohne eine mehrfach zu erzeugen. Durch die positiven Listeneinträge wird verhindert, dass es Probleme damit gibt, isolierte Stellen zu erzeugen, die im zweiten Schritt die Anzahl gleicher Petri-Netze erhöhen würden.

Für maximal drei Transitionen und maximal fünf Stellen ergibt sich also die folgende Liste:

$$(111), (112), (113), (122), (11), (12), (13), (14), (22), (23), (1), (2), (3), (4), (5)$$

Jeder Eintrag wird zwar im zweitem Schritt noch erweitert, der bereits durch ihn beschriebene Teilgraph des T-Netzes bleibt dabei jedoch erhalten. So wird zum Beispiel das in Abbildung 7.5 gezeigte Teil-Petri-Netz, das durch den Listeneintrag (122)

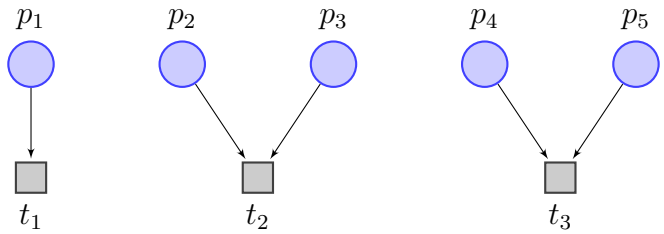


Abbildung 7.5: Durch (122) im erstem Schritt beschriebenes Teil-Petri-Netz

beschrieben wird, in allen Beispielen zum zweitem Schritt erhalten sein.

Für die Stellen wird im zweiten Teil bei den Stellen im Vorbereich einer Transition darauf geachtet, dass nur wenige isomorphe Möglichkeiten generiert werden, indem für jede Transition beachtet wird, dass die im Vorbereich dieser Transition liegenden Stellen entsprechenden Listeneinträgen aufsteigend geordnet sind. Hier würde eine Abweichung von dieser Sortierung einer Vertauschung von Stellen entsprechen, so dass zwar Stellen anders benannt wären, die Petri-Netze aber ansonsten isomorph wären. Es bleibt zusätzlich das Problem, dass es Transitionen geben kann, die nicht wesentlich verschieden sind, da ihre Vorbereiche gleich viele Stellen enthalten, die selbst isomorph sind, so dass es bei der Wahl der einen oder der anderen Transition als Vorbereich einer Stelle zu isomorphen Petri-Netzen kommt. Weiterhin gibt es auch bei den Vorbereichen von Stellen, die in den Vorbereichen verschiedener Transitionen liegen, Möglichkeiten, isomorphe Netze zu erhalten, da die Vertauschung von Vorbereichen solcher Stellen zu keinen wesentlichen Änderungen führt, wenn die Transitionen, in deren Vorbereichen die Stellen liegen, nicht wesentlich verschieden sind.

Für den oben bereits als dargestellten Listeneintrag (122) ergibt sich in diesem zweitem Schritt die folgende Liste, bei der die inneren Klammern die Stellen, deren Nach-

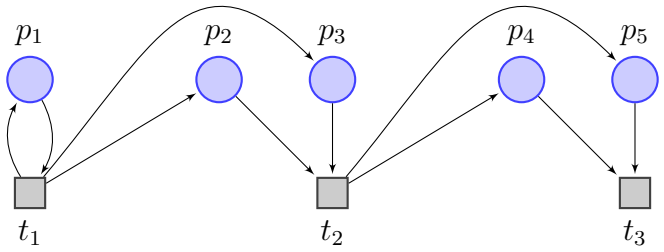


Abbildung 7.6: Petri-Netz, das im erstem Schritt mit (122) und im zweitem Schritt mit ((1)(11)(22)) beschrieben wird.

bereich gleich ist, zusammenfassen:

((1)(11)(11)),	((1)(11)(12)),	((1)(11)(13)),
	((1)(11)(22)),	((1)(11)(23)),
		((1)(11)(33)),
((1)(12)(11)),	((1)(12)(12)),	((1)(12)(13)),
	((1)(12)(22)),	((1)(12)(23)),
		((1)(12)(33)),
[...]		
((1)(33)(11)),	((1)(33)(12)),	((1)(33)(13)),
	((1)(33)(22)),	((1)(33)(23)),
		((1)(33)(33)),
((2)(11)(11)),	((2)(11)(12)),	((2)(11)(13)),
[...]		
		((3)(33)(33))

In den Abbildungen 7.6 und 7.7 sind beispielhaft die Netze, die sich für die Vorbereiche ((1)(11)(22)) und ((1)(33)(11)) ergeben, dargestellt. Wenn man bei diesen die Transitionen t_2 und t_3 sowie deren Vorbereiche vertauscht, ergibt sich das jeweils andere Netz. Diese beiden Vorbereiche beschreiben also isomorphe Netze.

Es ist also insgesamt festzustellen, dass der erste Teil garantiert keine isomorphen

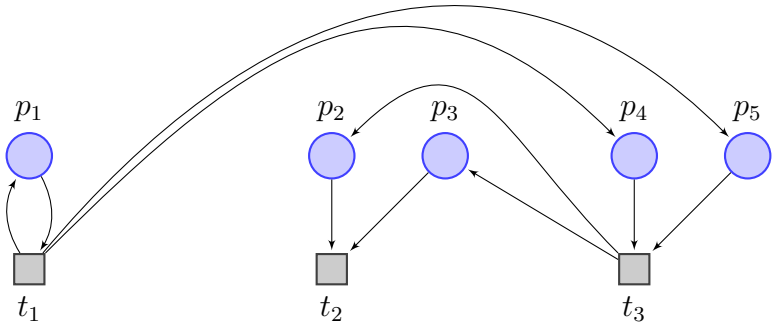


Abbildung 7.7: Petri-Netz, das im erstem Schritt mit (122) und im zweitem Schritt mit ((1)(33)(11)) beschrieben wird.

Teilnetze erzeugt, der zweite Teil jedoch bei der Erweiterung eines solchen Teilnetzes mehrere isomorphe Petri-Netze erzeugt. Dies wird insbesondere bei der Erzeugung größerer T-Netze problematisch. Es ist jedoch festzustellen, dass selbst ohne diese isomorphen Netze die Anzahl der T-Netze, die vorgegebene Schranken für die Anzahl an Transitionen und Stellen nicht überschreiten, bereits für ziemlich kleine Schranken ziemlich groß ist. Abbildung 7.8 stellt einige Ergebnisse von Zählungen dar, wobei die Zahlen des naiven Ansatzes mithilfe einfacher Kombinatorik[3] bestimmt wurden.

Für die Umsetzung von „Algorithmen zur Analyse von Petri-Netzen“ wurden zusätzliche Einflussmöglichkeiten geschaffen. Da Überhangtransitionen nur einen Nach-, aber keinen Vorbereich haben, führt deren Erzeugung dazu, dass das generierte Petri-Netz unbeschränkt ist. Da solche T-Netze in der genannten Umsetzung nicht verwendet werden können, wurde der Algorithmus erweitert, so dass die Erzeugung von Überhangtransitionen abgeschaltet werden kann. Da für die genannte Umsetzung T-Netze mit einer bestimmten Anzahl von Transitionen benötigt werden, wurde der erste Teil so erweitert, dass wahlweise die Transitionenanzahl genau erfüllt wird, statt sie als obere Grenze zu verwenden. Da die genannte Umsetzung zusätzlich die Erzeugung von Überhangtransitionen abschaltet, führt dies dazu, dass die generierten T-Netze die Transitionenanzahl genau erfüllen.

[3] Die genaue Formel lautet: $\sum_{m=1}^{n_t} \sum_{t=1}^{m} \sum_{s=1}^{2m} t^{2s}$

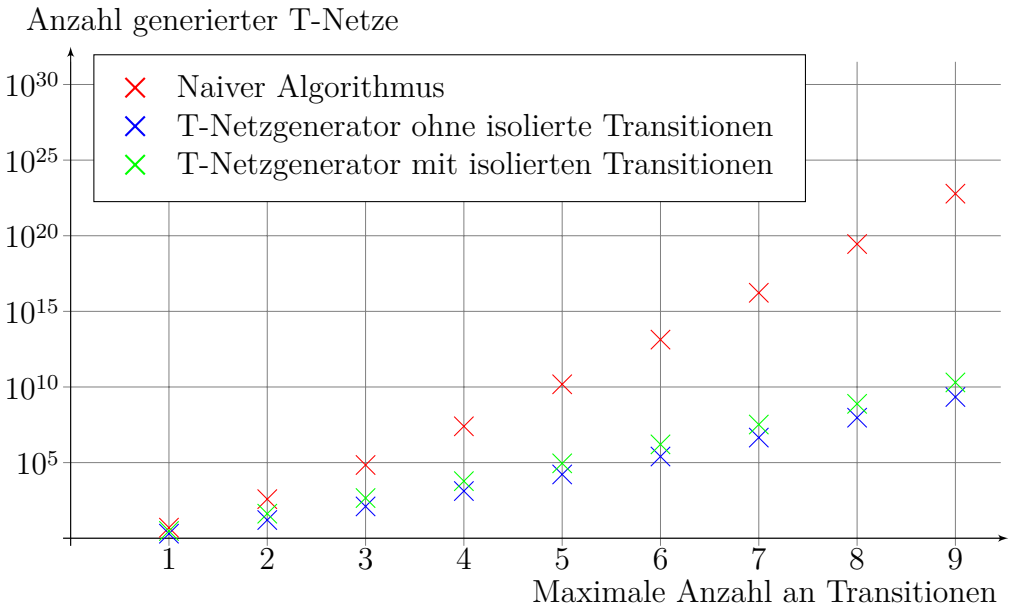


Abbildung 7.8: Anzahl von T-Netzen, die durch verschiedene Strategie generiert werden, wobei maximal doppelt so viele Stellen wie Transitionen generiert werden dürfen.

Generator für isolierte Transitionen Da der T-Netz-Generator nur T-Netze ohne isolierte Transitionen generiert, gibt es einen zusätzlichen Generator, der ein Petri-Netz und eine Angabe zur maximalen Transitionenanzahl bekommt. Dieser Generator erzeugt Netze, die aus dem ursprünglichen Netz mit zusätzlichen isolierten Transitionen bestehen. Dieser Generator wurde für die Verwendung mit dem T-Netz-Generator konstruiert, kann jedoch auch mit Petri-Netzen, die nicht von diesem stammen, verwendet werden. Dieser Generator findet tatsächlich keine Anwendung. Er wird zwar benötigt, um alle möglichen T-Netze zu generieren, allerdings hat sich gezeigt, dass T-Netze mit isolierten Transitionen nicht innerhalb der Projektgruppe benötigt werden.

Tests

Weil die Generatoren für einzelne Netze sehr einfach sind und Tests für diese im wesentlichen dadrin bestanden hätten, ein zweites Mal die Struktur der erzeugten Netze in Form von Programmcode darzustellen, wurde für diese größtenteils auf Testfälle verzichtet und nur manuell geprüft, ob die generierten Netze korrekt sind. Die wesent-

liche Ausnahme hierbei ist der Generator für das inverse Netz, bei dem ähnlich wie beim Überdeckungsgraphenalgorithmus Paare von bekannten Eingabe-Petri-Netzen und Ausgabe-Petri-Netzen als Testfälle genutzt und die Ausgabe auf Korrektheit geprüft wird.

Für die Generatoren für Petri-Netz-Familien gibt es Tests. Beim Generator für Markierungen wird geprüft, ob das Petri-Netz unverändert bleibt und ob für jede Markierung aus einer von Hand bestimmten Liste ein Petri-Netz mit dieser erzeugt wird. Beim Generator für isolierte Transitionen wird geprüft, ob die Ausgabe-Netze die korrekte Anzahl an Transitionen haben; ob die hinzugefügten Transitionen auch wirklich isoliert sind, ist hingegen schwer zu prüfen, da für den Test nicht einfach erkennbar ist, welche Transitionen die isolierten Transitionen sein sollen. Der T-Netz-Generator hat keine Test, die ihn in seiner Gesamtheit testen; stattdessen werden die einzelnen Komponenten getestet, indem die erzeugten Listen zu bestimmten Parameter-Tupeln mit von Hand bestimmten erwarteten Listen verglichen werden.

7.3.11 Algorithmen zur Analyse von Petri-Netzen

In diesem Abschnitt soll die Vorgehensweise zu den funktionalen Anforderungen in E1 und E2 vorgestellt werden. Darin soll ein gegebenes Petri-Netz N mit Anfangsmarkierung M_0 zunächst auf bestimmte Eigenschaften geprüft werden. Wenn diese erfüllt sind, sollen für alle T-Systeme (N', M_0') bis zu einer Größe g überprüft werden, ob ihr Erreichbarkeitsgraph isomorph zum Erreichbarkeitsgraph des gegebenen Petri-Netzes ist. Falls ein isomorphes T-System gefunden wurde, soll dieses T-System ausgegeben werden. Die letzten beiden Bedingungen werden in E2 derart abgeändert, dass nur genau ein zufällig gewähltes T-System überprüft wird. Dieses soll eine Möglichkeit zur schnelleren Überprüfung darstellen.

Neben der erwähnten Größe g, die als Parameter übergeben werden muss, kann optional auch noch eine maximale Tokenanzahl k angegeben werden, bis zu der die entsprechenden T-Systeme erzeugt werden. Anderenfalls wird bis zu einer Anzahl von 10 Token getestet bzw. in den später erwähnten Kreis-T-Systeme auf alle möglichen Token getestet. Um diese Größe wurden die Anforderungen erweitert, um eine genauere Eingrenzung der T-Systeme seitens der Nutzer zu ermöglichen.

Zu Beginn wird das gegebene Petri-Netz auf die zu überprüfenden Eigenschaften geprüft. Dazu wird für die jeweilige Eigenschaft der schon durch andere funktionale Anforderungen entstandene Algorithmus benutzt, um auf diese Eigenschaft zu prüfen. Ist eine der Eigenschaften nicht erfüllt, so wird eine entsprechende Fehlermeldung ausgegeben, die anzeigt, an welcher Vorbedingung die Ausführung gescheitert ist.

Ist dieser erste Schritt erfolgreich gewesen, so werden die Erreichbarkeitsgraphen für infrage kommende T-Systeme erzeugt und mit dem, durch das Testen der Vorbedingungen bereits entstandenen, Erreichbarkeitsgraph des gegebenen Petri-Netzes auf „schwache“ Isomorphie getestet. Dabei ist mit „schwacher“ Isomorphie gemeint, dass die Erreichbarkeitsgraphen nur auf strukturelle Isomorphie verglichen werden, bei der die Beschriftungen der Transitionen nicht betrachtet werden. Der Grund dieses Vorgehens ist das Fehlen von Beschriftungen der Transitionen im T-System, wodurch in seinem Erreichbarkeitsgraph die Transitionen autobeschriftet wurden und diese unter keinen Umständen mit denen des gegebenen Petri-Netzes übereinstimmen. Daher wird das Setzen der Beschriftungen im T-System nach der erfolgreichen Überprüfung der Isomorphie nachgeholt, welches in Abschnitt „Setzen von Beschriftungen nach erfolgreicher Überprüfung auf Isomorphie“ beschrieben wird.

Da es für die gegebenen Größen g und k unter Umständen sehr viele T-Systeme gibt, die überprüft werden müssen, werden im Vorfeld schon bestimmte T-Systeme gar nicht erst erzeugt. Dabei werden die zu überprüfenden T-Systeme in zwei Gruppen unterschieden: die Kreis-T-Systeme und die Nicht-Kreis-T-Systeme. Zunächst werden die Kreis-T-Systeme überprüft und wenn dabei kein passendes T-System gefunden werden, werden die Nicht-Kreis-T-Systeme überprüft. Zu diesen beiden Gruppen wird in den nächsten zwei Abschnitten erläutert, welche Einschränkungen gemacht werden konnten und wie solche T-Systeme erzeugt werden. Damit die Speicherkomplexität so gering wie möglich gehalten wird, werden bei der Überprüfung dieser beiden Gruppen immer nur ein T-Netz generiert und überprüft bevor ein neues erzeugt wird. Dies wurde durch die Implementierung des `Iterator`-Interfaces realisiert.

Auf die Implementierung für die zufällige Auswahl eines T-Systems wird in „ Zufälliges Auswählen eines T-Systems “ eingegangen.

Einschränkungen bei Kreis-T-Systemen

Kreis-T-Systeme, also Petri-Netze, in denen für alle Stellen $s \in S$ $|{}^\bullet s| = 1 = |s^\bullet|$ und für alle Transitionen $t \in T$ $|{}^\bullet t| = 1 = |t^\bullet|$ gilt, besitzen zu jeder Zeit die gleiche Anzahl an Token. Dies bedeutet, dass die Token nur auf den Stellen verschoben werden können und die Anzahl der Zustände des Erreichbarkeitsgraphen eines Kreis-T-Systems ohne die Erzeugung des Erreichbarkeitsgraphen berechnet werden kann. Dies ist in Bezug auf die Speicher- und Laufzeitkomplexität von Vorteil, da es billiger ist, anhand von zwei Eingabegrößen (Stellen- und Tokenanzahl) zu berechnen, wie viele Zustände der Erreichbarkeitsgraph hätte als diesen erst zu erzeugen und dann festzustellen, ob die Zustandsanzahl der des Erreichbarkeitsgraphen des gegebenen Petri-Netzes entspricht. Zur Berechnung der Anzahl an Zuständen des Erreichbarkeitsgraphes eines T-Systems wird der Binomialkoeffizienten $\binom{n_i+k_j-1}{k_j}$ für Kombination mit Wiederholung benutzt, wobei n_i die momentane Anzahl der Stellen des T-Systems ist und k_j die momentane Anzahl an Token im T-System beschreibt mit $0 \leq n_i \leq n$ und $0 < k_j \leq k$, falls k gegeben ist.

Konkret bedeutet dies nun, dass der Algorithmus bis zu Größe g hoch zählt und in jedem Schritt überprüft, ob es ein Kreis-T-System für eine Anzahl von Token gibt, dass dann die gleiche Anzahl an Zuständen im Erreichbarkeitsgraphen hat wie der Erreichbarkeitsgraph des gegebenen Petri-Netzes. Dafür wird die Tokenanzahl ebenso wie die Größe g hochgezählt und zu Bestimmung der Anzahl der Zustände die oben genannte Vorgehensweise genutzt. Übersteigt das Ergebnis der Berechnung des Binomialkoeffizienten die Anzahl an Zuständen im Erreichbarkeitsgraph des gegebenen Petri-Netzes oder ist bei Angabe einer maximalen Tokenanzahl k diese erreicht, so kann direkt zum nächsten Größen-Schritt übergegangen werden.

Ist ein T-System gefunden worden, wird geprüft, ob es einen Zustand im Erreichbarkeitsgraphen des T-Systems gibt, der die gleiche Anzahl an ausgehenden Kanten hat wie der Initialzustand des Erreichbarkeitsgraphen des gegebenen Petri-Netzes. Dies kann aufgrund des regelmäßigen Aufbaus des Erreichbarkeitsgraphen des T-Systems auch ohne die Erzeugung des T-Systems berechnet werden, wenn die Stellenanzahl und die Tokenanzahl des T-Systems bekannt ist. Nur in diesem Fall wird das T-System weiter betrachtet und erzeugt. Die Markierung des T-Systems wird dabei so erzeugt, dass die Initialzustände der Erreichbarkeitsgraphen die gleiche Anzahl an

ausgehenden Kanten besitzen.

Einschränkungen bei den Nicht-Kreis-T-Systemen

Bei der Erzeugung der Nicht-Kreis-T-Systeme wird auf den T-Netz-Generator aus Abschnitt 7.3.10 zurückgegriffen, dem die maximale Stellenanzahl übergeben wird. Dieser erzeugt lediglich T-Netze, die also noch keine Initialmarkierung besitzen, aber schon auf Ausschlusseigenschaften untersucht werden können, wodurch einige T-Netze ausgesiebt werden können. Wie schon im Abschnitt 7.3.10 beschrieben, produziert dieser Generator keine T-Netze, in denen Transitionen einen leeren Vorbereich haben können, wodurch schon einmal unbeschränkte T-Netze aussortiert werden konnten. Außerdem sollten alle Kreis-T-Netze übergangen werden, da diese ja schon überprüft wurden. Dieses kann dadurch geschehen, dass für ein erzeugtes T-Netz überprüft wird, ob es sich um ein S-Netz handelt. Da aber für diese Überprüfung die Definition 3.4.12 umgesetzt wurde, werden nicht nur Kreis-T-Netze ausgesiebt, sondern ebenfalls T-Netze, deren Transitionen im Nachbereich maximal eine Stelle haben. Dies hat den Vorteil, dass schon mal ein Teil derjenigen T-Netze aussortiert wird, die aufgrund eines leeren Nachbereichs einer ihrer Transitionen für die Weiterverwendung nicht in Frage kommen, da deren Erreichbarkeitsgraph dann mindestens ein Zustand besäße, der keine ausgehende Kante besitzen würde und dies mit der *reversibel*-Eigenschaft des gegebenen Petri-Netzes kollidieren würde.

Für die übrig gebliebenen T-Netze wird nun der aus Abschnitt 7.3.10 bekannte Markierungsgenerator genutzt, um für eine maximale Anzahl an Token alle T-Systeme zu generieren.

Zufälliges Auswählen eines T-Systems

Wie eingangs erwähnt, kostet dieses Verfahren viel Rechenzeit, da die zu testenden T-Systeme immer in derselben Reihenfolge von den mit den wenigsten Stellen bis zu denen mit der Stellenanzahl g getestet werden. Dies macht es für große g nicht handhabbar. Das zufällige Auswählen eines T-Systems umgeht aus diesem Grund

die Generatoren und erstellt ein Netz mit einer gleichverteilten Anzahl an Stellen $s_\#$ mit $1 \leq s_\# \leq g$. Die Anzahl der Transitionen wird ebenso gleichverteilt aus dem Intervall eins bis $2 \cdot s_\# + 1$ gezogen. Die obere Grenze ist zum einen auf Grund der Eigenschaft des T-Systems gewählt, da jede Stelle genau eine Transition im Vor- wie auch im Nachbereich besitzen muss. Zum anderen kann es maximal eine tote Transition geben, da sonst der Erreichbarkeitsgraph an jedem Knoten mehrere Schlaufen besitzt. Dies besagt in dem Fall, dass sie unterschiedlich beschriftet sind, dass nicht alle kleinsten Kreise denselben Parikh-Vektor haben. Da das Vergleichsnetz jedoch diese Eigenschaft besitzt und für die Isomorphieprüfung auf das zu erstellende Netz überträgt, ist ein Netz mit mehr als einer toten Transition nicht relevant für den Vergleich. Sind die toten Transitionen gleich beschriftet, werden durch mehrere tote Transitionen keine neuen Erreichbarkeitsgraphen erhalten und somit können die relevanten Fälle mit einer toten Transition abgedeckt werden.

Des Weiteren werden noch die Transitionen mit einem leeren Vor- beziehungsweise Nachbereich aussortiert, da diese – wie schon im vorherigen Abschnitt erklärt – gegen die Beschränktheit beziehungsweise die *reversibel*-Eigenschaft des gegebenen Petri-Netzes, welche sich ebenso aufgrund der Isomorphieprüfung auf das zu suchende Netz vererbt, verstoßen.

Setzen von Beschriftungen nach erfolgreicher Überprüfung auf Isomorphie

Hat ein T-System die „schwache“ Isomorphieprüfung bestanden, so müssen nun noch die Beschriftungen der Transitionen gesetzt werden. Dazu wird die von dem Isomorphie-Modul bereitgestellte Menge aus zueinander isomorphen Zustände der Erreichbarkeitsgraphen genutzt. Ausgehend von dem Paar aus den beiden initialen Zuständen der Erreichbarkeitsgraphen werden nun die zueinander isomorphen Zustandspaare ermittelt, die von dem Initialzustandspaar durch Feuern genau einer Transition erreichbar sind. Für diese wird in den einzelnen Erreichbarkeitsgraphen ermittelt, welche beiden Transitionen daran beteiligt sind und die Beschriftung wird von der Transition des Erreichbarkeitsgraphen des gegebenen Petri-Netzes auf die Transition des Erreichbarkeitsgraphen des T-Systems übertragen, sowie auf alle Transitionen in diesem Erreichbarkeitsgraphen, die die selbe Autobeschriftung besitzen, da diese auf die gleiche Transition im T-System zurückzuführen sind. Ebenso wird im T-

System die Transition beschriftet. Dies wird per Tiefensuche nun für alle isomorphen Zustandspaare durchgeführt.

Speicher- und Zeitkomplexität

Die Speicher- und Zeitkomplexität unterscheidet sich darin, ob nur auf ein zufällig gewähltes T-System getestet werden soll oder auf alle T-Systeme bis zu einer bestimmten Größe g und einer maximalen Tokenanzahl k, falls letztere gegeben ist. In beiden Fällen wird bei der Angabe von Zeit- und Speicherkomplexität nicht das Prüfen der Vorbedingungen mit eingerechnet, da bei den entsprechenden Algorithmen die dazugehörige Speicher- und Zeitkomplexität gefunden werden kann. Bei beiden Varianten muss der Erreichbarkeitsgraph erzeugt werden, dessen Zeit- und Speicherkomplexität für die Berechnung im entsprechenden Abschnitt 7.3.3 erläutert wird.

Für den Fall, dass auf alle T-Systeme bis zu einer bestimmten Größe g und einer maximalen Tokenanzahl k, falls letztere gegeben ist, getestet werden soll, hat diese Variante eine Speicherkomplexität von $\mathcal{O}(|S_{T-Sys}|+|T_{T-Sys}|+|A_{T-Sys}|+|k_{T-Sys}|+|iso^2|\cdot\left|T_{E_{T-Sys}}\right|)$. Diese ergibt sich, weil jedes untersuchte T-System für die Untersuchungszeit gespeichert werden muss und dessen Größe $|S_{T-Sys}| + |T_{T-Sys}| + |A_{T-Sys}| + |k_{T-Sys}|$, also die Summe aus Stellen, Transitionen, Kanten und Marken, ist. Beim Setzen der Beschriftungen müssen im schlimmsten Fall für jedes isomorphe Paar die nachfolgenden isomorphen Paare bestimmt werden, wobei dann nochmal alle isomorphen Paare durchgegangen werden. Dies erklärt das $|iso^2|$. Außerdem muss im schlimmsten Fall für jedes isomorphe Paar alle Transitionen des T-Systems durchgegangen werden, um ihnen gegebenenfalls eine neue Beschriftung zu geben. Deshalb ist die Multiplikation mit $\left|T_{E_{T-Sys}}\right|$ notwendig. Die Zeitkomplexität liegt in $\mathcal{O}((|g^3|\cdot|k^2|+|N|) * |iso^2|\cdot \left|T_{E_{T-Sys}}\right|\cdot\sum_{k=1}^{iso}(2\cdot|T_{k1}|+|T_{k2}|))$. Dabei fällt $|g^3|\cdot|k^2|$ für die Berechnung aller möglichen Kreis-T-Systeme an, N stellt die Zeit für die Berechnung alles T-Systeme durch den T-Net-Generator dar und $|iso^2|\cdot\left|T_{E_{T-Sys}}\right|\cdot\sum_{l=1}^{iso}(2\cdot|T_{l1}|+|T_{l2}|)$ ergibt sich durch das Setzen der Beschriftungen, wobei T_{l1} die Anzahl der ausgehenden Transitionen des 1. Zustands des isomorphen Paares l und T_{l2} dann analog die Anzahl der ausgehenden Transitionen des 2. Zustands darstellt.

Für den Fall, dass auf ein zufällig gewähltes T-System getestet werden soll kommt zu der oben erwähnten Speicherkomplexität für das Erstellen des T-Systems noch einmal eine Komplexität aus $\mathcal{O}(|T_{T-Sys}|)$ hinzu, um eine möglichst gute Verteilung zu erreichen und noch einmal $\mathcal{O}(|T_{T-Sys}|)$ um die Transitionen mit leerem Vor- beziehungsweise Nachbereich aussortieren zu können. Insgesamt liegt die Speicherkomplexität dieses Algorithmus also in $\mathcal{O}(|S_{T-Sys}|+|T_{T-Sys}|+|A_{T-Sys}|+|k_{T-Sys}|+|iso^2|\cdot\left|T_{E_{T-Sys}}\right|)$.

Zu der Zeitkomplexität addiert sich im Fall eines zufällig gewählten T-Systems noch eine Komplexität von $\mathcal{O}(|T_{T-Sys}|^2+|S_{T-Sys}|)$ für das Erstellen des Systems. Dabei wird die Zeit aus $\mathcal{O}(|S_{T-Sys}|)$ benötigt, um initial die Vor- und Nachbereiche der Stellen zu belegen, $\mathcal{O}(|T_{T-Sys}|)$, um die toten Transitionen zu entfernen und sich die Transitionen mit leeren Vor- beziehungsweise Nachbereich zu merken. Anschließend wird im schlimmsten Fall, wenn genau die Hälfte der Transitionen einen leeren Vor- und die anderen einen leeren Nachbereich besitzen, eine Komplexität in $\mathcal{O}(|T_{T-Sys}|^2)$ benötigt, um die leeren Vor- beziehungsweise Nachbereiche möglichst zufällig zusammenzufügen, ohne die erwähnten Netzeigenschaften zu zerstören. Insgesamt liegt die Zeitkomplexität für den Fall des zufälligen Wählens eines T-Systems in $\mathcal{O}(|T_{T-Sys}|^2+|S_{T-Sys}|+|iso^2|\cdot\left|T_{E_{T-Sys}}\right|\cdot\sum_{l=1}^{iso}(2\cdot|T_{l1}|+|T_{l2}|))$.

Tests

Aufgrund der sehr starken Vorbedingungen, welche an ein zu übergebendes Netz gestellt werden, wurde der größte Teil dieses Algorithmus lediglich mit zwei Netzen getestet. Dabei handelt es sich zum einem um ein etwas größeres Petri-Netz aus [BD11], welches im Grundlagen-Kapitel dargestellt wurde, zum anderen um ein Petri-Netz mit nur einer Stelle, auf der sich 5 Token befinden. Im Fall, dass auf alle T-Systeme bis zu einer bestimmten Größe g und einer maximalen Tokenanzahl k getestet wird, wird ein isomorphen Kreis-T-System gefunden, wodurch der zweite Teil des Algorithmus, in dem die T-Systeme untersucht werden, die durch den T-Netz-Generator erzeugt werden, nicht ausgeführt wird.

Der Algorithmus für das zufällige Erstellen von T-Systemen wurde für den Parameter g von eins bis hundert getestet.

7.3.12 Algorithmen zur Erstellung von Petri-Netzen mit bestimmten Eigenschaften

Zum Auffinden von Petri-Netzen nach Anforderungen G1 und G2 wurde das Check-Modul geschrieben. Das Check-Modul kann mithilfe verschiedener Generatoren Petri-Netze finden, die vom Nutzer gewünschte Eigenschaften erfüllen. Der Nutzer kann dabei frei aus den unterstützten Eigenschaften wählen und diese mit einem beliebigen Generator kombinieren, welcher dann zur Erzeugung der Petri-Netze genutzt wird.

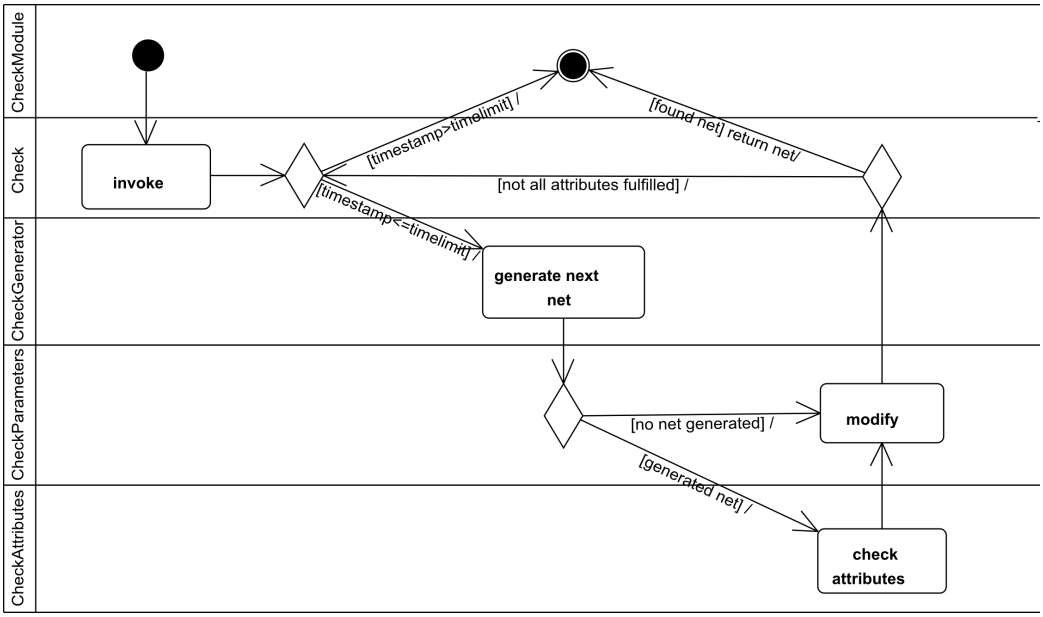


Abbildung 7.9: Ablauf der Suche nach Petri-Netzen über das Check-Modul

Der Ablauf der Suche ist in einem Aktivitätsdiagramm dargestellt (siehe Abbildung 7.9). Beim Aufruf des Moduls werden die vom Nutzer gewünschten Parameter an das Check-Modul übergeben. Das Check-Modul ruft den vom Benutzer gewählten Generator auf, der ein Petri-Netz erzeugt. Im folgenden Schritt wird das Petri-Netz auf die vom Benutzer geforderten Eigenschaften überprüft. Die Parameter für den Generator-Aufruf werden modifiziert, so dass der Generator beim nächsten Aufruf ein anderes, im Optimalfall passenderes Petri-Netz zurückgibt. Für die Modifikation der Parameter wird je nach Generator entweder ein simpler Zähler oder die Anzahl der bisher erfüllten Eigenschaften übergeben, so dass auf die aktuelle Entwicklung des Petri-Netzes reagiert werden kann. Erfüllt das erzeugte Petri-Netz alle Eigenschaften, liefert es das Check-Modul an den Benutzer zurück. Für den Fall, dass nicht alle

Eigenschaften erfüllt wurden, wird entweder das nächste Petri-Netz generiert oder beim Überschreiten des vom Nutzer gesetzten Zeitlimits die Suche abgebrochen.

War es nicht möglich, ein passendes Petri-Netz zu erzeugen, so werden dem Nutzer die für das Check-Modul maximal vereinbarten Eigenschaften ausgegeben. Hierdurch soll die Analyse, weshalb kein Petri-Netz gefunden wurde, unterstützt werden.

Spezielle Generatoren

Neben den Generatoren aus Abschnitt 7.3.10 wurden auch Generatoren speziell zum Finden von Petri-Netzen mit bestimmen Eigenschaften implementiert. Diese Generatoren arbeiten zufallsgesteuert und versuchen, ein möglichst breites Spektrum an Petri-Netzen mit verschiedenen Eigenschaften zu erzeugen.

Chance Generator Der Chance Generator ist ein relativ einfacher, zufallsgesteuerter Generator, der jedoch in der Lage ist, Petri-Netze zu erzeugen, die verschiedene Eigenschaften erfüllen. Die Basis für die generierten Petri-Netze bildet immer eine Struktur aus zwei Stellen verbunden durch eine Transition. Diese Grundstruktur wird um bis zu sechs weitere Stellen und Transitionen erweitert, welche Kanten zu allen zu dem Zeitpunkt vorhandenen Elementen haben können. Die so erzeugte erste Version des Petri-Netzes wird an das Check-Modul zurückgegeben. Anhand der Rückmeldung vom Check-Modul wird der Generator im Weiteren das Petri-Netz modifizieren oder verwerfen und ein neues Petri-Netz bauen, wenn das vorhandene Petri-Netz auch nach mehreren Modifikationen nicht zu dem gewünschten Ergebnis führt. Die Anzahl der Modifikationsschritte ist davon abhängig, ob sich das Petri-Netz durch die bisherigen Modifikationen dem gewünschten Ergebnis angenähert hat. Bei der Modifikation des Petri-Netzes kann eine weitere Stelle mit Kanten zu Transitionen ergänzt werden, Token können platziert, Kantengewichte modifiziert oder Transitionen mit Kanten zu Stellen hinzugefügt oder entfernt werden.

Beispiele für erstellte Petri-Netze Abbildung 7.10 zeigt ein vom Chance Generator erzeugtes Petri-Netz, welches die Eigenschaften „!isolated, !reversible, !snet und tnet“

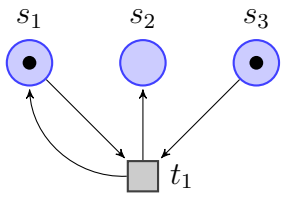


Abbildung 7.10: Erzeugtes Petri-Netz

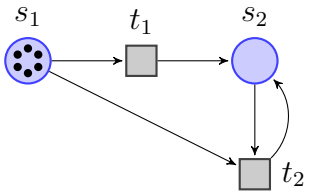


Abbildung 7.11: Erzeugtes Petri-Netz

erfüllt. Abbildung 7.11 zeigt ein weiteres vom Chance Generator erzeugtes Petri-Netz, welches die Eigenschaften „!strongly_3-separable und 2-marking“ erfüllt.

Smart Chance Generator Der Smart Chance Generator basiert auf einem ähnlichen Ansatz wie der Chance Generator. Bei der Generierung eines neuen Petri-Netzes wird ein Basisnetz, bestehend aus sechs Stellen und Transitionen, erzeugt und diese über Kanten verbunden. Das so erstellte Basisnetz wird an das Check-Modul zurückgegeben. Erfüllt das Petri-Netz nicht die vom Nutzer gewünschten Eigenschaften, wird das Check-Modul eine Rückmeldung an den Generator übergeben. Es wird dann eine Modifikation des Petri-Netzes vorgenommen. Eine Modifikation kann dabei das Hinzufügen einer Stelle, einer Transition, einer Kante, das Erhöhen eines Kantengewichts oder das Setzen einer zusätzlichen Markierung sein. Die Modifikation wird in einer Historie gespeichert. Das so modifizierte Petri-Netz wird wieder an das Check-Modul zurückgegeben. Stellt sich eine Verbesserung, bzw. eine Verschlechterung der Ergebnisse ein, werden die Wahrscheinlichkeiten, mit der die Modifikationen zufällig ausgewählt werden, angepasst. Das Petri-Netz wird so schrittweise modifiziert. Stellen sich nach mehreren Modifikationen keine Verbesserungen ein, wird das Petri-Netz durch die Historie in einen früheren Zustand mit dem bisher besten Ergebnis zurückgesetzt und es wird mit den Modifikationen fortgefahren. Stellt sich nach mehreren Rücksprüngen keine Verbesserung ein, wird das Petri-Netz endgültig verworfen und es wird ein neues Basisnetz generiert.

Unterstützte Eigenschaften und Generatoren

In Tabelle 7.3 sind die Eigenschaften aufgelistet, auf die Check automatisch prüfen kann und Tabelle 7.4 zeigt die Auswahlmöglichkeiten der Generatoren an.

Parameter	Prüft, ob das Petri-Netz ...
bounded (+ !bounded)	... beschränkt ist.
freeChoice (+ !freeChoice)	... ein FC-Netz ist.
isolated (+ !isolated)	... isolierte Knoten hat.
k-marking (+ !k-marking)	... eine k-Markierung als initiale Markierung hat.
persistent (+ !persistent)	... persistent ist.
plain (+ !plain)	... schlicht ist.
pure (+ !pure)	... pur ist.
reversible (+ !reversible)	... reversibel ist.
snet (+ !snet)	... ein S-Netz ist.
stronglyLive (+ !stronglyLive)	... stark lebendig ist.
!strongly_k-seperable	... nicht stark k-separierbar ist.
tnet (+ !tnet)	... ein T-Netz ist.
!weakly_k-seperable	... nicht schwach k-separierbar ist.

Tabelle 7.3: Eigenschaften, auf die das Check-Modul testen kann

Generator	Generiert ...
bitnet	... ein Bit-Netz mit variabler Anzahl an Bits.
chance	... zufallsgesteuert ein Petri-Netz und reagiert dabei auf den Fortschrift der bisher erzeugten Petri-Netze.
cycle	... einen Kreis mit zunehmender Anzahl an Stellen und Transitionen.
quadPhilgen	... Petri-Netze zum Philosophenproblem mit variabler Anzahl an Philosophen und vier Zuständen.
smartchance	... ähnlich wie der Chance Generator Petri-Netze, jedoch mit komplexerer Reaktion auf den Fortschritt der bisher erzeugten Petri-Netze.
tnetgen2	... T-Netze mit variabler Anzahl an Stellen und Transitionen – die Anzahl der Token wird anhängig von der Anzahl der Stellen gesetzt.
tnetgen3	... T-Netze mit variabler Anzahl an Stellen, Transitionen und Token.
triPhilgen	... Petri-Netze zum Philosophenproblem mit variabler Anzahl an Philosophen und drei Zuständen.

Tabelle 7.4: Auswahlmöglichleiten bei den Generatoren

Laufzeit- und Speicherkomplexität

Die Laufzeitkomplexität und der Speicherbedarf hängen von der Wahl der Eigenschaften und des Generators ab.
Geht man davon aus, dass die Laufzeitkomplextität der gewählten Eigenschaften $\mathcal{O}(E_i)$ ist und n_e verschieden Eigenschaften gewählt wurden, dann ergibt sich für die einmalige Überprüfung der Eigenschaften eine Komplextität von $\sum_{i=1}^{n_e} \mathcal{O}(E_i)$. Hinzu kommt die Laufzeitkomplextität des gewählten Generators $\mathcal{O}(G)$ und die Laufzeitkomplextität für die Modifikation der Parameter für den Aufruf des Generators. Diese ist abhängig vom Generator. Bei den aktuell vorhandenen Genratoren kann sie im ungünstigsten Fall abhängig von der Anzahl der gewählten Eigenschaften n_e sein. Somit ergibt sich für das einmalige Erzeugen und Auswerten eines Petri-Netzes die Komplexität:

$$\mathcal{O}(G) + \sum_{i=1}^{n_e} (\mathcal{O}(E_i)) + \mathcal{O}(n_e)$$

Wie viele Schritte (n_{steps}) nötig sind, um ein gesuchtes Petri-Netz zu finden, ist von der Kombination aus Generator und Eigenschaften abhängig und kann allgemein nicht angegeben werden. Sollte ein Generator prinzipiell kein Petri-Netz mit den geforderten Eigenschaften erzeugen können, so kann n_{steps} gegen unendlich gehen. Dies ist zum Beispiel der Fall, wenn ein T-Netz-Generator ein Petri-Netz mit der Eigenschaft „kein T-Netz“ erzeugen soll. Im Fall der zufallsgesteuerten Generatoren kann n_{steps} mit jedem Aufruf variieren.
Für den gesamten Check-Modul-Aufruf ergibt sich eine Laufzeitkomplextität von:

$$\sum_{j=1}^{n_{steps}} \left(\mathcal{O}(G) + \sum_{i=1}^{n_e} (\mathcal{O}(E_i)) + \mathcal{O}(n_e) \right)$$

Bei der Speicherkomplexität interessieren die Speicherkomplexitäten der Eigenschaften ($\mathcal{O}(EP_i)$) und der Generatoren $\mathcal{O}(GP)$. Da alle Schritte in dem Modul sequenziell abgearbeitet werden, ist die Speicherkomplexität abhängig von dem Teil, der am meisten Speicher benötigt. Zusätzlich muss sich im ungünstigsten Fall für jede Eigenschaft ein Parameter gemerkt werden. Die Anzahl der Schritte hat keinen Einfluss auf den Speicherbedarf. Die Speicherkomplexität lautet:

$$\max\left(\mathcal{O}(GP), \mathcal{O}(EP_1), \mathcal{O}(EP_2), \mathcal{O}(EP_3), ...\right) + \mathcal{O}(n_e)$$

Test

Das Check-Modul erzeugt ein Petri-Netz, welches vom Nutzer geforderte Eigenschaften erfüllen soll. Um zu testen, ob das erstellte Petri-Netz zu der Menge der Petri-Netze gehört, die die geforderten Eigenschaften erfüllen, muss das erstellte Petri-Netz analysiert werden. Zu diesem Zweck wäre ein Analysewerkzeug für Petri-Netze mit passend hoher Bandbreite an Analyse-Algorithmen nötig.

Die Analyse-Algorithmen des APT Projekts sind ein Analysewerkzeug für Petri-Netze mit passend hoher Bandbreite, erfüllen somit diese Anforderungen und werden zum Testen der vom Check-Modul erstellten Petri-Netze genutzt.

Das Check-Modul, welches intern die Analyse-Algorithmen des APT Projekts nutzt, mit den Analyse-Algorithmen des APT Projekts zu testen, klingt erstmal unpassend. Bei einem Fehler in den Analyse-Algorithmen kann das Check-Modul ein falsches Petri-Netz erzeugen. Da der Fehler wahrscheinlich auch beim Test mit dem entsprechenden Analyse-Algorithmus auftritt, wird der Fehler nicht erkannt. Jedoch ist es nicht die Aufgabe der Tests für das Check-Modul Fehler in den Analyse-Algorithmen festzustellen. Die Analyse-Algorithmen besitzen ihre eigenen Tests. Die Aufgabe der Tests für das Check-Modul ist es Fehler im Check-Modul zu finden und hierfür können die Analyse-Algorithmen des APT Projekts genutzt werden.

Sollte ein Test fehlschlagen, so ist zur Fehleranalyse das konkrete, erstellte Petri-Netz erforderlich. Bei der Verwendung eines zufallsgesteuerten Generators ist der Testfall sonst nicht rekonstruierbar, was die Fehlersuche erschwert. Dazu gibt es eine hohe Anzahl an verschiedenen Testfällen. Aus diesen Gründen wurde neben den Unit-Tests, welche die grundsätzliche Funktionsfähigkeit des Check-Moduls sicher stellen sollen, ein Shell-Skript entworfen, welches automatisch verschiedene Eigenschaften an das Check-Modul übergibt, das erzeugte Petri-Netz speichert und die Eigenschaften von den entsprechenden Analyse-Modulen des APT Projekts analysieren lässt. Anhand einer formatierten Log-Dateien, welche auch Verknüpfungen zu den gespeicherten Petri-Netzen beinhaltet, kann überprüft werden, ob alle Tests in Ordnung waren.
(Das Shell-Skript „check_test.sh“ befindet sich im Ordner des Unit-Tests vom Check-Modul. Dort gibt es auch zwei Beispiel-Ordner, die je eine Log-Datei von einem Testlauf über 15 Tests inklusive der erzeugen Petri-Netze beinhalten.)

7.4 Algorithmen für beschriftete Petri-Netze

In diesem Abschnitt wird die Umsetzung der funktionalen Anforderungen betrachtet, die sich auf beschriftete Petri-Netze beziehen.

7.4.1 Algorithmus für „Wort in Petri-Netz-Sprache“

Hier stellen wir den Algorithmus zur Überprüfung, ob ein bestimmtes Wort von einem gegebenen Petri-Netz erzeugt werden kann. Dieser Algorithmus setzt Anforderung C2 um.

Um zu prüfen, ob ein gegebenes Wort in der von einem Petri-Netz erzeugten Sprache ist, wird eine Tiefensuche durch den Erreichbarkeitsgraphen durchgeführt. Hierzu werden in jeder Markierung nur die Transitionen betrachtet, die das nächste Symbol[4] des Wortes produzieren. Sobald das geforderte Wort erzeugt werden konnte, ist es in der Petri-Netz-Sprache und die zugehörige Feuersequenz wird zurückgegeben. Falls jedoch die Tiefensuche erfolglos abbricht, so kann das gegebene Petri-Netz das Eingabewort nicht generieren.

Im schlechtesten Fall liegt die Laufzeit des Algorithmus in $\mathcal{O}(|T|^n)$, wobei n die Länge des zu untersuchenden Wortes ist und $|T|$ die Anzahl der Transitionen des Petri-Netzes. Falls die Transitionsbeschriftungen injektiv sind, also eine Beschriftung nur von genau einer Transition erzeugt wird, ist eine bessere Laufzeit ($\mathcal{O}(n)$) möglich. Dies stellt auch den bestmöglichen Fall für allgemeine Beschriftungen dar. Außerdem wird für die Tiefensuche und eine Zuordnung von Symbolen zu Transitionen bis zu $\mathcal{O}(n + |T|)$ Speicher benötigt.

Dieser Ansatz wurde gewählt, da er für injektive Beschriftungsabbildungen bereits eine optimale asymptotische Laufzeit erreicht. Außerdem lässt sich dieser Algorithmus leicht umsetzen, wie an der kurzen Implementierung gesehen werden kann.

[4] Eine Erweiterung der Petri-Netz-Sprachen sind ϵ-Beschriftungen. Eine Transition mit solch einer Beschriftung erzeugt kein Symbol, wenn sie feuert. Diese Erweiterung wird nicht unterstützt.

Tests

Ein Problem, dass es schwerer macht die Implementierung zu testen, liegt im nichtdeterministischen Ergebnis dieses Algorithmus. Dies liegt daran, dass verschiedene Feuersequenzen das gleiche Wort erzeugen können. Daher müssen in den Tests eventuell mehrere verschiedene Feuersequenzen als korrektes Ergebnis akzeptiert werden.

Zunächst wird das leere Petri-Netz als Randfall untersucht. Dieses Petri-Netz kann natürlich das leere Wort erzeugen, jedoch keine weiteren Wörter. Das leere Wort kann auch von anderen Netzen erzeugt werden. Schließlich werden noch auf mehreren Petri-Netzen verschiedene Wörter getestet, die nicht alle in der Petri-Netz-Sprache enthalten sind. Hierbei werden auch Wörter getestet, die ein Wort der Sprache als Präfix haben, um eine teilweise erfolgreiche Tiefensuche und das nötige Backtracking zu testen.

7.4.2 Algorithmus für die Sprachäquivalenz von Petri-Netzen

In Anforderung C4 wird gefordert, dass zwei Petri-Netze auf Sprachäquivalenz untersucht werden können.

Zwei Petri-Netze sind sprachäquivalent, wenn sie die gleiche Sprache erzeugen. Dieses kann durch Algorithmen aus der Automatentheorie untersucht werden. Hierzu kann der Erreichbarkeitsgraph als nichtdeterministischer endlicher Automat (NEA) aufgefasst werden, bei dem jeder Zustand akzeptierend ist. Somit kann der folgende Algorithmus zum Prüfen der Sprachäquivalenz von NEAs eingesetzt werden.

Zunächst werden sprachäquivalente deterministische endliche Automaten (DEA) erzeugt. Hierzu wird die allgemein bekannte Potenzmengenkonstruktion verwendet. Zu diesen beiden Automaten wird der Produktautomat betrachtet, der die Sprache $L = L_1 \setminus L_2 \cup L_2 \setminus L_1$ akzeptiert. Zustände in diesem Automat bestehen aus zwei Zuständen der ursprünglichen Automaten, wobei die Zustandsübergänge komponentenweise erhalten bleiben.

Die beiden Petri-Netze aus der Eingabe haben genau dann identische Sprachen, wenn es keinen akzeptierenden Zustand im Produktautomaten ergibt. Dieses bedeutet, dass es keinen erreichbaren Zustand gibt, in dem nur einer der beiden Teilzustände akzeptierend ist.

Es reicht aus, wenn der erreichbare Teil des Produktautomaten durch eine Tiefensuche beginnend im Startzustand bestimmt wird. Hierdurch wird weniger Zeit benötigt, insbesondere wenn die Sprachen nicht äquivalent sind. Außerdem kann ein Wort angegeben werden, das nur in einer der beiden Sprachen ist, indem der Pfad zum Startzustand zurückverfolgt wird.

Zusätzlich können die DEAs noch verkleinert werden, indem ihr Minimalautomat konstruiert wird. Hierdurch wird der Produktautomat kleiner, was eine weitere Zeitersparnis bedeutet. Für diesen Schritt wurde der allgemein bekannte Algorithmus von Hopcroft verwendet, welcher Mengen nicht-unterscheidbarer Zustände bestimmt und diese zusammenfasst.

Eine Einschränkung dieses Algorithmus ist es, dass nur beschränkte Petri-Netze analysiert werden können. Andernfalls ist der Erreichbarkeitsgraph unendlich groß und kann daher nicht als NEA interpretiert werden.

Die Komplexität dieses Algorithmus soll hier nicht näher untersucht werden. Es reicht zu wissen, dass das Erstellen des Minimalautomaten bereits einen exponentiellen Zeitbedarf hat. Zusammen mit der ähnlich hohen Komplexität zum Erzeugen des Erreichbarkeitsgraphen wird klar, dass dieser Algorithmus eine äußerst hohe Komplexität hat. Leider konnte kein effizienterer Ansatz gefunden werden.

Tests

Der vorgestellte Algorithmus besteht aus mehreren Teilen, die unabhängig voneinander getestet werden können. Zunächst wird geprüft, ob die Konstruktion des deterministischen endlichen Automaten das erwartete Ergebnis liefert. Als nächstes wird die Automatenminimierung getestet, indem als Eingabe ein Transitionssystem verwendet wird, dass deterministisch ist, aber nicht benötigte Zustände enthält.

Es wird auch der gesamte Algorithmus getestet. Hierzu wird ein Transitionssystem um eine zusätzliche Kante erweitert, die ein weiteres Wort zur Sprache des Systems hinzufügt. Dieses Wort wird anschließend durch den Algorithmus bestimmt. Auf ähnliche Weise werden auch verschiedene andere Transitionssysteme verglichen. Zusätzlich wird jedes dieser Transitionssysteme mit dem Transitionssystem verglichen, dass nur aus einem einzelnen Zustand besteht.

7.4.3 Algorithmen zur Isomorphie

In diesem Abschnitt wird der Algorithmus zur Anforderung C5 (siehe Seite 41) beschrieben. Als Grundlagendefinition ist Definition 3.2.8 zu betrachten.

Sollen zwei Petri-Netze N_1 und N_2 auf Isomorphie zueinander getestet werden, so genügt es, ihre Erreichbarkeitsgraphen G_1 und G_2 auf Isomorphie zueinander zu testen, da diese das Verhalten der Petri-Netze widerspiegeln. Sind die Erreichbarkeitsgraphen nicht endlich, so ist der Test auf Isomorphie nicht möglich.

Es gibt also für den Isomorphie-Test zwei Schritte zu erledigen:

1. Erstelle die Überdeckungsgraphen G_1 und G_2 zu N_1 und N_2

2. Falls G_1 und G_2 endlich, prüfe diese beiden Graphen auf Isomorphie zueinander

Schritt 1 ist einfach erledigt, da hierfür das vorhandene Modul zum Erstellen von Überdeckungsgraphen genutzt werden kann (siehe Abschnitt 7.3.2). Für Graphenisomorphie wurde der VF2-Algorithmus umgesetzt (siehe [CFSV01, CFSV99]), adaptiert von einer Implementierung in C++, die in einem Fremdsystem umgesetzt wurde (siehe Abschnitt 5.4.2). Dieser Algorithmus für Graphenisomorphie wird im nächsten Abschnitt näher beschrieben.

Zu Erwähnen sei an dieser Stelle noch, dass das Isomorphie-Modul zwei verschiedene Arten von Isomorphie überprüfen kann, die in diesem Abschnitt wie folgt benannt sein sollen:

strukturelle Isomorphie: Überprüft die Erreichbarkeitsgraphen der beiden Petri-Netze auf strukturelle Isomorphie, was bedeutet, dass Beschriftungen im LTS ignoriert werden.

Isomorphie: Überprüft die Erreichbarkeitsgraphen der beiden Petri-Netze auf Isomorphie (siehe Definition 3.2.8), was bedeutet, dass Beschriftungen beachtet werden. Wenn ein solcher Isomorphismus gefunden wird, impliziert dieses strukturelle Isomorphie, während die Rückrichtung dieser Implikation nicht gilt.

Die Anforderung C5 forderte nur die mit Definition 3.2.8 gegebene Isomorphie, da es allerdings für den in Abschnitt 7.3.11 beschriebenen Algorithmus nötig war, beim Isomorphie-Test der beschrifteten Transitionssysteme die Beschriftungen ignorieren zu können, wurde diese zusätzliche Funktion implementiert. Um auszuwählen, ob für die Isomorphie Beschriftungen beachtet werden sollen oder nicht, muss beim Aufruf des Konstruktors lediglich die boolesche Variable `checkLabels` auf `true` oder `false` gesetzt werden.

Der VF2-Algorithmus

Das Ziel des Algorithmus ist es, für zwei gegebene Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ eine Zuordnungsfunktion zu finden, welche jeden Knoten aus G_1 auf genau einen Knoten aus G_2 abbildet. Wenn eine solche Zuordnungsfunktion existiert, so werden G_1 und G_2 als isomorph zueinander bezeichnet. Die Zuordnungsfunktion ist gegeben durch:

$$M = \{(n, m) \in V_1 \times V_2 \mid \text{n wird abgebildet auf m}\}.$$

Ein charakteristisches Merkmal des VF2-Algorithmus ist, dass dieser zustandsorientiert arbeitet. Für den initialen Zustand werden stets die Startknoten n_0 von G_1 und m_0 von G_2 als Knotenpaar auf Isomorphie zueinander geprüft, indem die Funktion `Match(s)` aufgerufen wird (siehe Listing 7.4). Weiterhin sei die Menge $M(s)$ die partielle Abbildung zu M im Zustand s; $M(s)$ enthält also alle im Zustand s bereits gefundenen isomorphen Knotenpaare aus G_1 und G_2 und ist somit Teilmenge des

```
procedure Match(s)
 if M(s) enthält alle Knoten von G2 then
  output M(s);
 else
  Berechne P(s);
  for ∀p ∈ P(s) do
   if feasible(s,p) then
    Berechne s' durch Hinzufügen von p zu M(s);
    Match(s');
   end if
  end for
 end if
end procedure
```

Listing 7.4: VF2-Algorithmus [CFSV99]

potentiell zu findenden Isomorphismus. Ferner sei mit $M_1(s)$ die Menge definiert, in der alle im Zustand s gewählten Knoten aus G_1 enthalten sind. Analog ergibt sich $M_2(s)$.

In jedem weiteren Zustand s des Algorithmus werden nach einem später erläuterten Auswahlverfahren einige Knotenpaare (n_i, m_j) zunächst in die Menge $P(s)$ aufgenommen. Danach wird mit der Methode `feasible(s, p)` für alle Elemente $p \in P$ im Zustand s überprüft, ob jene in die partielle Abbildung $M(s)$ aufgenommen werden können, also isomorphe Knotenpaare darstellen. Sobald ein neues Knotenpaar in $M(s)$ aufgenommen wird, ergibt sich aus dieser erweiterten partiellen Abbildung $M(s')$ der Folgezustand s' und auch hierfür wird die Funktion `Match(s)` aufgerufen.

Konstruktion von $P(s)$

Bevor erklärt wird, wie die Menge $P(s)$ konstruiert wird, müssen einige weitere Mengen definiert werden:

- $T_1^{out}(s)$: Menge aller Knoten in G_1, welche nicht in $M_1(s)$ enthalten sind, aber Nachfolger eines Knotens in $M_1(s)$ sind.
- $T_1^{in}(s)$: Menge aller Knoten in G_1, welche nicht in $M_1(s)$ enthalten sind, aber Vorgänger eines Knotens in $M_1(s)$ sind.

Analog werden die beiden Mengen $T_2^{out}(s)$ und $T_2^{in}(s)$ gebildet. Zudem seien $T_1(s) = T_1^{in}(s) \cup T_1^{out}(s)$ und $T_2(s)$ analog notiert.

Bei der Konstruktion von $P(s)$ existieren vier verschiedene Fälle:

Fall 1: Sowohl $T_1^{out}(s)$ als auch $T_2^{out}(s)$ sind nicht leer, dann:

$$P(s) = T_1^{out}(s) \times \{\min T_2^{out}(s)\}$$

Hierbei ist $\min T_2^{out}(s)$ der Knoten aus $T_2^{out}(s)$ mit dem niedrigsten Label, wobei auch eine beliebige andere Sortierung möglich ist.

Fall 2: Sowohl $T_1^{out}(s)$ als auch $T_2^{out}(s)$ sind leer, aber $T_1^{in}(s)$ und $T_2^{in}(s)$ sind beide nicht leer, dann:

$$P(s) = T_1^{in}(s) \times \{\min T_2^{in}(s)\}$$

Fall 3: Alle vier T-Mengen sind leer, dann:

$$P(s) = (N_1 - M_1(s)) \times \{\min(N_2 - M_2(s))\}$$

Fall 4: Nur eine der T_{in}-Mengen oder nur eine der T_{out}-Mengen ist leer: In diesem Fall kann der Zustand s verworfen werden, da er nicht Teil des Isomorphismus sein kann.

Diese Konstruktion von $P(s)$ verhindert weiterhin, dass ein Zustand mehrfach betrachtet wird.

Funktion `feasible(s,p)`

Wie in Listing 7.4 zu sehen, wird nach der Konstruktion der Menge $P(s)$ jedes Element p dieser Menge mit der Funktion `feasible(s,p)` daraufhin untersucht, ob es

in die partielle Abbildung $M(s)$ aufgenommen werden darf und somit Teil der Isomorphie ist.

Um $p = (n_i, m_j)$ daraufhin zu untersuchen, ob es ein isomorphes Knotenpaar ist, wird nicht nur der aktuelle Zustand (Look-Ahead 0), sondern auch der nächste (Look-Ahead 1) und der übernächste Zustand (Look-Ahead 2) betrachtet. Die Aufnahmekriterien sind Tabelle 7.5 zu entnehmen.

Look-Ahead	Name	Bedingung
0	R_pred	Für alle Vorgänger $n' \in M(s)$ von n, ist der zugeordnete Knoten m' ein Vorgänger von m und umgekehrt.
	R_succ	Für alle Nachfolger $n' \in M(s)$ von n, ist der zugeordnete Knoten m' ein Nachfolger von m und umgekehrt.
1	R_termin	Die Anzahl der Vorgänger (Nachfolger) von n in $T_1^{in}(s)$ ist gleich der Anzahl Vorgänger (Nachfolger) von m in $T_2^{in}(s)$
	R_termout	Die Anzahl der Vorgänger (Nachfolger) von n in $T_1^{out}(s)$ ist gleich der Anzahl Vorgänger (Nachfolger) von m in $T_2^{out}(s)$
2	R_new	Die Anzahl der Vorgänger (Nachfolger) von n, die weder in $M_1(s)$ noch in $T_1(s)$ vorkommen, ist gleich der Anzahl Vorgänger (Nachfolger) von m, die weder in $M_2(s)$ noch in $T_2(s)$ vorkommen.

Tabelle 7.5: Aufnahmeregeln der Funktion `feasible(s,p)` [CFSV99]

Komplexität

In diesem Abschnitt soll kurz auf die Speicher- und Laufzeitkomplexität des Algorithmus eingegangen werden. Für detailliertere Informationen können [CFSV01] und [CFSV99] herangezogen werden. Ein interessanter Vergleich des VF2-Algorithmus

mit vier anderen Algorithmen zur Graphenisomorphie kann zudem in [FSV01] nachgeschlagen werden.

Die Laufzeitkomplexität gliedert sich in mehrere Teilinformationen auf. Zunächst wird die Zeit betrachtet, die benötigt wird, um in einem Zustand die neuen potentiellen Folgezustände zu ermitteln. Hierbei muss zunächst in $T_1^{out}(s)$ der Zustand mit dem kleinsten Label ermittelt werden und anschließend müssen in G_2 unter allen Knoten diejenigen gesucht werden, die zu $T_2^{out}(s)$ gehören. Da davon ausgegangen wird, dass beide Graphen die gleiche Knotenanzahl besitzen, ist die Komplexität hier (n).

Weiterhin muss betrachtet werden, wie lange es dauert einen einzelnen Zustand zu untersuchen. Hierbei wird Zeit benötigt, um zu überprüfen, ob die Regeln der Funktion `feasible(s,p)` erfüllt werden und es wird Zeit benötigt, um die Mengen $T_1^{in}(s)$, $T_2^{in}(s)$, etc. zu bilden. Hierfür wird eine Laufzeit von (n) veranschlagt.

Somit liegt die Laufzeitkomplexität im Best Case bei (n^2). Im Worst Case liegt die Laufzeitkomplexität bei $(n!n)$. Der Worst Case kann eintreten, wenn die Graphen nahezu vollständig verbunden sind.

Die Speicherplatzkomplexität ist recht einfach zu ermitteln, da Tiefensuche zusichert, dass sich bei einem Graphen mit n Knoten maximal n Zustände zugleich im Speicher befinden können. Zudem müssen in jedem Zustand selber nur einige wenige Mengen gespeichert werden, was einen konstant kleinen Speicherplatz beansprucht. Die Speicherkomplexität liegt somit bei (n).

Tests

Bei den Unit-Tests für das Isomorphie-Modul gibt es zunächst drei übergeordnete Fälle:

1. Fall: Die eingegebenen Petri-Netze sind isomorph zueinander (und implizit strukturell isomorph).

2. Fall: Die eingegebenen Petri-Netze snd strukturell isomorph zueinander (aber

nicht isomorph).

3. Fall: Die Petri-Netze sind nicht strukturell isomorph zueinander (und damit auch nicht isomorph)

Ein Beispiel-Testfall für vorhandene Isomorphie sind zwei Petri-Netze, bei denen lediglich die Beschriftungen untereinander anders benannt wurden und ein Beispiel für nicht-Isomorphie sind zwei Petri-Netze, die „fast“ isomorph sind, bei denen aber nach einigen Feuersequenzen doch noch nicht isomorphe Markierungen gefunden werden.

Die Anforderung C5 besagt, dass die Startknoten der Erreichbarkeitsgraphen gegebener Petri-Netze isomorph zueinander sein sollen. Daher wurde neben den genannten drei Arten von Testfällen hierfür ein Testfall formuliert, in welchem die Erreichbarkeitsgraphen strukturell isomorph sind, aber die beiden Startknoten nicht isomorph zueinander sind, weshalb die beiden LTS nicht isomorph sind.

7.4.4 Algorithmen zur Bisimulation

In diesem Abschnitt werden die einzelnen Schritte erläutert, die zu den verwendeten Algorithmen für die Bisimulation führen, sowie die Algorithmen selbst aufgeführt. Als Grundlagendefinition ist Definition 3.2.9 zu betrachten.

Um flexible Einsatzmöglichkeiten dieses Moduls gewährleisten zu können, werden, abweichend von der Definition C6, folgende Eingabemöglichkeiten für das Modul erlaubt:

- Zwei beschriftete und beschränkte initiale Petri-Netze
- Zwei beschriftete Transitionssysteme
- Ein beschriftetes und beschränktes Petri-Netz und ein beschriftetes Transitionssystem

Für ein Petri-Netz als Eingabe wird der Erreichbarkeitsgraph erzeugt und zur Überprüfung der Bisimilarität genutzt. Aus diesem Grund müssen die Algorithmen also

für beschriftete Transitionssysteme ausgelegt sein. Daher fiel die Entscheidung auf die Umsetzung der Algorithmen aus [FM90], woraus auch die folgenden Definitionen stammen, die an die in der Projektgruppe verwendete Definition von Transitionssystemen angepasst wurden. Auch die Algorithmen wurden angepasst, um nicht berücksichtigte Randfälle behandeln zu können und bei nicht gegebener Bisimilarität einen Fehlerpfad ausgeben zu können.

Definitionen

Im Folgenden werden Definitionen gegeben, auf denen die kommenden Algorithmen aufbauen und die für deren Verständnis benötigt werden. Wenn ein Petri-Netz als Eingabe erfolgt, so wird mit dem Erreichbarkeitsgraph gearbeitet, weshalb dann für die folgenden Definitionen gilt, dass T gleich Σ gilt. Dazu wird für jede Beschriftung a und für jeden Zustand s eines beschrifteten Transitionssystems $N = (S, \rightarrow, T, s_0)$ die Bildmenge benötigt und wie folgt definiert:

$$\rightarrow_a [s] = \{s' \in S \mid (s, a, s') \in \rightarrow\}$$

.

Abweichend des Gebrauchs von σ in [FM90] wird hier dafür γ genommen, um keine Verwirrung bezüglich σ als Feuersequenz zu stiften.

Definition 7.4.1 (Ausführungssequenz) *Sei $N = (S, \rightarrow, T, s_0)$ ein beschriftetes Transitionssystem und $s \in S$ ein Zustand. Die Menge der endlichen Ausführungssequenzen von s $(Ex(s))$ wird wie folgt definiert:*
$Ex(s) = \{\gamma \in S^* \mid \gamma(0) = s \wedge \forall i \text{ mit } 0 \leq i \leq |\gamma| \, \exists t_i \in T \gamma(i) \xrightarrow{t_i} \gamma(i+1)\}$

Eine Ausführungssequenz ist elementar, wenn all ihre Zustände verschieden sind. So wird die Teilmenge von $Ex(s)$, die die elementaren Ausführungssequenzen von Zustand s enthält, mit $Ex_e(s)$ definiert.

Es wird nun eine Definition des Produkts zweier beschrifteter Transitionssysteme gegeben, mit dem es einfacher ist zu untersuchen, ob eine Bisimulation zwischen den beiden beschrifteten Transitionssystemen existiert oder nicht.

Definition 7.4.2 *Seien $N_i = (S_i, \rightarrow_i, T_i, s_{0i})$ mit $i = 1, 2$ zwei beschriftete Transitionssysteme. Das Produkt $N_1 \times N_2$ ist wie folgt definiert:*
$N = (S, \rightarrow, T, (s_{01}, s_{02}))$ mit $S \subseteq (S_1 \times S_2) \cup \{fail\}$, $T = (T_1 \cap T_2) \cup \{\phi\}$ und $\rightarrow \subseteq S \times T \times S$, wobei $fail \notin (T_1 \cap T_2)$ und $\phi \notin (S_1 \cup S_2)$ ist. S und $\rightarrow$ sind als kleinste Mengen definiert, die folgende Regeln erfüllen:

- $(s_{01}, s_{02}) \in S$
- $$\frac{(s_1,s_2)\in S, \quad s_1 \xrightarrow{t}_1 s_1', \quad s_2 \xrightarrow{t}_2 s_2'}{(s_1',s_2')\in S, \quad \{(s_1,s_2) \xrightarrow{t} (s_1',s_2')\} \in \rightarrow}$$
- $$\frac{(s_1,s_2)\in S, \quad s_1 \xrightarrow{t}_1 s_1', \quad \rightarrow_{2t}[s_2] = \varnothing}{\{fail\}\in S, \quad \{(s_1,s_2) \xrightarrow{\phi} fail\} \in \rightarrow}$$
- $$\frac{(s_1,s_2)\in S, \quad s_2 \xrightarrow{t}_2 s_2', \quad \rightarrow_{1t}[s_1] = \varnothing}{\{fail\}\in S, \quad \{(s_1,s_2) \xrightarrow{\phi} fail\} \in \rightarrow}$$

Mithilfe dieses Produktes kann Bisimilarität von N_1 und N_2 charakterisiert werden:

Proposition 7.4.1 *Seien N_1 und N_2 zwei beschriftete Transitionssysteme, sei $N = (S, \rightarrow, T, (s_{01}, s_{02}))$ das Produkt $N_1 \times N_2$. Dann gilt $s_{01} \not\sim s_{02}$ genau dann, wenn es eine elementare Ausführungssequenz γ von S ($\gamma \in Ex_e((s_{01}, s_{02})))$ gibt, so dass:*

- $\gamma = \{(s_{01}, s_{02}) = (p_0, q_0), (p_1, q_1), ..., (p_k, q_k), fail\}$
- $\forall i$ *mit* $0 \leq i \leq k$ *gilt* $p_i \not\sim^{k-i+1} q_i$ □

Diese Proposition kann man noch verstärken, wenn mindestens eines der beschrifteten Transitionssysteme deterministisch ist:

Proposition 7.4.2 *Seien N_1 und N_2 zwei beschriftete Transitionssysteme, sei $N = (S, \rightarrow, T, (s_{01}, s_{02}))$ das Produkt $N_1 \times N_2$ und seien N_1 oder N_2 deterministisch. Dann gilt:*

$$S_1 \not\sim S_2 \Leftrightarrow \exists \gamma \in Ex((s_{01}, s_{02})) : \exists k > 0 : \gamma(k) = fail$$ □

Dies bedeutet, dass, wenn mindestens eines der beschrifteten Transitionssysteme deterministisch ist, N_1 und N_2 genau dann nicht bimiliar sind, wenn es eine Ausführungssequenz von $N_1 \times N_2$ gibt, die*fail* enthält.

Mithilfe der Nutzung dieser beiden Propositionen wurden zwei Algorithmen entworfen, die auf Bisimilarität testen. Im folgenden Abschnitt werden jetzt die Vorgehensweise sowie die entworfenen Algorithmen, die dann auch im Tool umgesetzt wurden, erläutert.

Algorithmen

In diesem Abschnitt werden die Algorithmen aufgeführt, die im Analysewerkzeug zum Testen der Bisimilarität eingesetzt werden.

Zu Beginn der Ausführung wird zunächst geprüft, ob Petri-Netze eingegeben wurden. Wenn dies der Fall ist, werden für diese die Erreichbarkeitsgraphen erzeugt. Danach existieren für alle Eingabemöglichkeiten immer zwei beschriftete Transitionssysteme (N_1 und N_2), mit denen weitergearbeitet werden kann. Nun wird für diese beiden Transitionssysteme geprüft, ob sie bisimilar sind. Für den Fall, dass mindestens eines der Transitionssysteme deterministisch ist, ist die Prüfung auf Bisimilarität einfacher als für den generellen Fall. Dies spiegelt sich auch in der Ausführungszeit und im Speicherplatzbedarf des Codes wider, denn wie Tests gezeigt haben, kann bis zu einer Millisekunde an Zeit gespart werden, wenn es eine Fallunterscheidung gibt. Dabei wurden nur relativ kleine Netze getestet, wodurch bei größeren Netzen die Zeitersparnis größer sein könnte. Deshalb wird für die beiden beschrifteten Transitionssysteme getestet, ob mindestens eines von ihnen deterministisch ist, und dementsprechend der passende Algorithmus gewählt wird.

In beiden später erläuterten Algorithmen ist es nötig, für ein Paar (p, q) mit p aus S_1 und q aus S_2 die möglichen direkten Nachfolger zu bestimmen. Diese werden anhand der Regeln von Definition 7.4.2 bestimmt, wobei vorher klar sein muss, welche direkten Nachfolger die einzelnen Zustände in den jeweiligen beschrifteten Transitionssystemen haben. Diese Bestimmung der möglichen direkten Nachfolger wird in den Algorithmen durch die `getSuccessor`-Funktion realisiert, die wie folgt definiert ist:

$$\texttt{getSuccessor(p,q)} = \{(\texttt{a}, (\texttt{p}', \texttt{q}')) \mid \texttt{p} \xrightarrow{\texttt{a}}_1 \texttt{p}' \wedge \texttt{q} \xrightarrow{\texttt{a}}_2 \texttt{q}'\}$$

Für zwei nichtdeterministische Transitionssysteme bedeutet dies auch, dass, wenn

beide Zustände eines Paars (p, q) mehrere ausgehende Transitionen mit gleicher Beschriftung besitzen, die Menge aller direkten Nachfolger alle Kombinationen (p', q') umfasst, die möglich sind. Des Weiteren musste die `getSuccessor`-Funktion derart erweitert werden, dass der direkte Nachfolger von *fail* immer *fail* ist mit der Transition mit dem Label ϕ. Dies ist nötig, da die später erläuterten Algorithmen sonst für den Fall, dass das Initial-Zustandspaar sofort *fail* als direkten Nachfolger hat, nicht funktionieren.

Für den Fall, dass die beiden Transitionssysteme nicht bisimilar sind, wird ein Gegenbeispiel in Form eines Fehlerpfad ausgegeben. Um diesen zu erstellen, gibt es die `constructErrorPathPairs`-Funktion, die einen Stack übergeben bekommt, der als Elemente jeweils ein Zustandspaar mit deren Nachfolgern besitzt und die aktuelle Ausführungssequenz repräsentiert, bis wohin der Pfad bisimilar war. Für den Fehlerpfad werden die Zustandspaare der einzelnen Elemente in der Reihenfolge vom untersten zum obersten Element des Stacks aneinandergereiht, und dieser wird dann zurückgegeben. In den kommenden Algorithmen wird dann lediglich noch das Element hinzugefügt, das nicht mehr bisimilar ist.

Für den deterministischen Fall besitzt der entsprechende Algorithmus zwei Datenstrukturen:

$\texttt{St}_1$: Dies ist ein Stack, auf dem die momentane Ausführungssequenz gespeichert wird, die zu dieser Zeit analysiert wird.

`W`: Dies ist eine Menge aller schon besuchten Zustände.

Dann sieht der Algorithmus für den deterministischen Fall wie folgt aus:

```
Algorithm1
    ErrorPath := ∅
    St1 := {((s01, s02), getSuccessors(s01, s02))}
    W := {(s01, s02)}
    while St1 ≠ ∅
        ((q1,q2), l) := top(St1)
        if l = ∅ //wenn alle Nachfolger überprüft wurden
            pop(St1) {then backtrack}
        else
            choose and remove (q'1,q'2) in l
            if ¬((q'1,q'2) -φ-> fail)
                if (q'1,q'2) ∉ W //Dies ist ein neues Zustandspaar
                    insert (q'1,q'2) in W
                    push {((q'1,q'2), getSuccessor(q'1,q'2))} in St1
                fi
            else
                ErrorPath := constructErrorPathPairs(St1);
                if((q'1,q'2) != fail)
                    insert(q'1,q'2) in ErrorPath
                fi
                return FALSE //fail ist erreichbar
            fi
        fi
    endwhile
    return TRUE //fail ist nicht erreichbar
end.
```

Listing 7.5: Algorithmus für den deterministischen Fall

Wie anhand des Algorithmus zu sehen ist, wird über Tiefensuche vorgegangen. Dadurch muss nie der komplette Graph gespeichert werden, sondern lediglich nur die momentan zu analysierende Ausführungssequenz und die schon besuchten Zustände. Letzteres wird gemacht, um die zeitliche Effizienz zu steigern. Denn wenn ein Zustand aus W noch einmal besucht wurde, muss dieser nicht nochmal analysiert werden, da dann schon bekannt ist, dass von ihm kein Zustand erreichbar ist, der aus einem Paar zweier nicht-bisilarer Zustände besteht. Das Paar (s_{01}, s_{02}) stellt den Initial-Zustand von $N_1 \times N_2$ dar. Zeit- und Speicherkomplexität des Algorithmus liegen laut [FM90]

in $\mathcal{O}(n)$, wenn n die Anzahl der Zustände von $N_1 \times N_2$ ist.

Im generellen Fall muss die Proposition 7.4.1 umgesetzt werden. Dafür muss festgestellt werden können, ob $p \not\sim^i q$ für irgendein i für jeden Zustand (p, q) in S gilt. Dazu wird wie bei dem deterministischen Fall die Tiefensuche eingesetzt und dabei Folgendes überprüft:

- Die Relation $p \not\sim^1 q$ kann leicht überprüft werden.
- Für jeden analysierten Zustand (p, q) ist das Ergebnis $(p \not\sim^i q)$ für seine Vorgänger hergestellt worden und wird während des Backtracking-Verfahrens analysiert.

Aus diesem Grund muss im Algorithmus gespeichert werden können, ob für einen Zustand (p, q) die Nachfolger bisimilar sind. Dies wird durch eine Map `M` realisiert, die die Größe $(\rightarrow_1 |p|) + (\rightarrow_2 |q|)$ besitzt, die einzelnen Nachfolger als Keys hat und deren Werte aus der Menge $\{0, 1\}$ stammen. Initial besitzt `M` für jeden Nachfolger den Wert 0. Solch eine Map `M` wird für einen Zustand (p, q) auf einen Stack gelegt, wenn dieser Zustand das erste Mal in der Ausführungssequenz auftaucht. Während der Analyse des Zustands werden alle Nachfolger (p', q') durch die `getSuccessor`-Funktion bestimmt. Durch die Tiefensuche werden nun zuerst alle (p', q') analysiert, und wenn $p' \sim q'$ gilt, so wird `M[p']` = `1` und `M[q']` = `1` gesetzt. Ist dann das Backtracking-Verfahren auf den Zustand (p, q) zurückgekehrt, so gilt $(p \sim q)$, wenn alle Elemente von `M` den Wert 1 besitzen.

Für den generellen Fall besitzt der entsprechende Algorithmus also folgende Datenstrukturen:

`St`$_1$: Dies ist ein Stack, auf dem die momentane Ausführungssequenz gespeichert wird, die zu dieser Zeit analysiert wird.

`St`$_2$: Dies ist der Stack, der die oben erwähnten Maps M beinhaltet.

`W`: Dies ist eine Menge von Zuständen, deren Status „$\not\sim$“ ist, also deren enthaltenen Zustände paarweise nicht bisimilar sind.

R: Dies ist eine Menge von Zuständen der aktuellen Ausführungssequenz, die schon mehr als einmal besucht wurden.

V: Dies ist eine Menge von Zuständen, die alle schon besuchten Zustände beinhaltet.

Dann sieht der Algorithmus für den generellen Fall wie folgt aus:

```
1  Algorithm3
2      W := ∅
3      repeat
4          result := partial_DFS //Führe eine Tiefensuche durch
5      until result ∈ {TRUE, FALSE}
6      return result
7  end.
8
9  function partial_DFS
10     ErrorPath := ∅
11     V := ∅ ; R := ∅ ; stable := false
12     St1 := {((s01, s02), getSuccessors(s01, s02))}
13     St2 := ∅
14     push into St2 a map of size 2 //für (s01, s02)
15     push into St2 a map of size (|→1 [s01]| + |→2 [s02]|)
16     while St1 ≠ ∅
17         stable := true
18         ((q1,q2), l) := top(St1)
19         M := top(St2)
20         if l ≠ ∅
21             choose and remove (q1',q2') in l
22             if (q1',q2') ∉ V ∪ W
23                 if (q1',q2') ∉ St1 //Dies ist ein neues Zustandspaar
24                     if ¬((q1',q2') -φ-> fail)
25                         push {(q1',q2'), getSuccessors(q1',q2')} in St1
26                         push into St2 a map of size (|→1 [q1']| + |→2 [q2']|)
27                     fi
28                 else
29                     //Dieses Paar ist mehr als einmal besucht worden
30                     insert (q1',q2') in R
31                     M[q1'] := 1 ; M[q2'] := 1
```

```
32              fi
33          else // wurde schon überprüft
34              if (q'1, q'2) ∉ W
35                  M[q'1] := 1 ; M[q'2] := 1 //q'1 ~ q'2
36              fi
37          fi
38      else
39          pop(St2)
40          insert (q1, q2) in V //Ein neues Zustandspaar wurde analysiert
41          M' := top(St2)
42          if M[q'] = 1 for all q' in (→1[q1] ∪ →2[q2])
43              M'[q1] := 1 ; M'[q2] := 1 //q1 ~ q2
44          else
45              insert (q1, q2) in W //q1 ≁ q2
46              if (q1, q2) ∈ R
47                  stable := false //Es wurde ein falscher Status
                        angenommen
48              fi
49              if (ErrorPath ≠ ∅)
50                  ErrorPath := constructErrorPathPairs(St1)
51                  k := getSuccessors(q1, q2)
52                  for each element(q'1, q'2) of k
53                      if (M[q'1] = 0 || M[q'2] = 0)
54                          if ((q'1, q'2) --φ--> fail)
55                              insert(q'1, q'2) in ErrorPath
56                              leave for-loop
57                          fi
58                      fi
59                  endfor
60              fi
61          fi
62          pop(St1)
63      fi
64  endwhile
65  M := top(St2)
66  if M[q01] ≠ 1 and M[q02] ≠ 1
67      return FALSE //q01 ≁ q02
```

```
68      else
69          if stable
70              return TRUE //q01 ~ q02
71          else
72              return UNRELIABLE //Eine weitere Tiefensuche muss durchgeführt
                    werden
73          fi
74      fi
75  end.
```

Listing 7.6: Algorithmus für den generellen Fall

Wie im Algorithmus 3 zu sehen ist, kann es drei verschiedene Zustände geben: `TRUE, FALSE, UNRELAIBLE`. Dabei drücken `TRUE` und `FALSE` aus, ob die Bisimilarität der beiden Transitionssysteme besteht, während `UNRELIABLE` angibt, dass das Ergebnis der `partial_DFS`-Funktion noch nicht zuverlässig ist und deshalb eine weitere Tiefensuche durchgeführt werden muss. Die Zeitkomplexität des Algorithmus liegt laut [FM90] in $\mathcal{O}(n^2)$ und die Speicherkomplexität in $\mathcal{O}(n)$, wobei n die Anzahl an Zuständen von $N_1 \times N_2$ ist.

Beispiel Der Algorithmus für den generellen Fall ist sehr lang und komplex. Aus diesem Grund wird in diesem Abschnitt der Algorithmus anhand eines Beispieles demonstriert. Die beiden Transitionssysteme, die als Eingabe benutzt werden, werden in Abbildung 7.12 und 7.13 gezeigt.

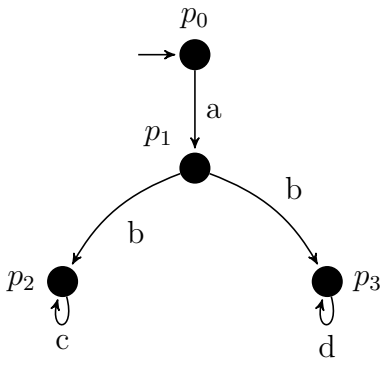


Abbildung 7.12: Transitionssystem 1

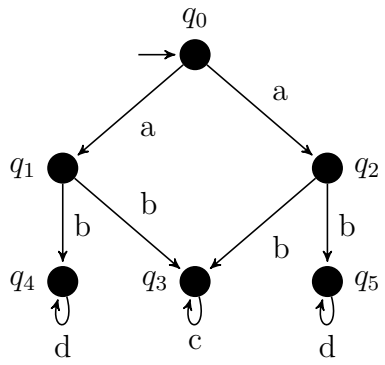


Abbildung 7.13: Transitionssystem 2

Bei jeder Ausführung von `Algorithm3` wird mindestens einmal die `partial_DFS`-Funktion aufgerufen. Nachdem die Variablen initialisiert und die Stacks $\mathtt{St}_1$ und $\mathtt{St}_2$ mit den Werten für das Initialpaar (p_0, q_0) gefüllt worden sind, haben die Variablen nach Zeile (15) folgende Werte:

$\mathtt{V} : \{\}$
$\mathtt{R} : \{\}$
$\mathtt{W} : \{\}$
$\mathtt{St}_1 : \{((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_1)), (\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})\}$
$\mathtt{St}_2 : \{[(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)], [(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}$

Weil $\mathtt{St}_1$ nicht leer ist, wird die Schleife in Zeile (16) ausgeführt. Darin werden die Variablen `((`q_1, q_2`), l)` und `M` wie folgt belegt:

$((\mathtt{q}_1, \mathtt{q}_2), \mathtt{l}) : ((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_1)), (\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})$
$\mathtt{M} : [(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)]$

Da das Paar (p_0, q_0) Nachfolger besitzt, also `l` nicht leer ist, ist die Bedingung in (20) wahr und der zugehörige Code wird ausgeführt. Darin wird zunächst der erste Nachfolger des Paares (p_0, q_0) ausgewählt, also $(q_1', q_2') = (p_1, q_1)$ und aus der Liste der Nachfolger entfernt sowie für diesen überprüft, ob er sich in der Vereinigung von `V` und `W` befindet. Da beide Mengen leer sind, wird noch geprüft, ob sich das Zustandspaar in $\mathtt{St}_1$ befindet. Da dies nicht der Fall ist und (q_1', q_2') *fail* nicht als Nachfolger besitzt, wird (q_1', q_2') mit seinen Nachfolgern in $\mathtt{St}_1$ eingefügt (25) sowie eine ausreichend große Map auf den Stack $\mathtt{St}_2$ (26) gelegt. Die Belegung der Variablen sieht dann wie folgt aus:

$\mathtt{V} : \{\}$
$\mathtt{R} : \{\}$
$\mathtt{W} : \{\}$
$\mathtt{St}_1 : \{((\mathtt{p}_1, \mathtt{q}_1), \{(\mathtt{b}, (\mathtt{p}_2, \mathtt{q}_4)), (\mathtt{b}, (\mathtt{p}_2, \mathtt{q}_3)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_4)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3))\}),$
$\quad ((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})\}$
$\mathtt{St}_2 : \{[(\mathtt{q}_3, 0), (\mathtt{q}_4, 0), (\mathtt{p}_2, 0), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)], [(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}$

Damit ist der erste Durchgang der `while`-Schleife beendet. Im nächsten Durchgang der Schleife ist $(q_1', q_2') = (p_2, q_4)$, dessen Nachfolger *fail* ist, wodurch (24) zu `false` ausgewertet wird und die `while`-Schleife sofort beendet wird. Im nächsten Durchgang

wird $(q_1', q_2') = (p_2, q_3)$ untersucht, die sich selbst als Nachfolger haben und somit (25) und (26) ausgeführt wird. Dann sieht die Belegung der Variablen wie folgt aus:

$\mathtt{V} : \{\}$
$\mathtt{R} : \{\}$
$\mathtt{W} : \{\}$
$\mathtt{St}_1 : \{((\mathtt{p}_2, \mathtt{q}_3), \{(\mathtt{c}, (\mathtt{p}_2, \mathtt{q}_3))\}), ((\mathtt{p}_1, \mathtt{q}_1), \{(\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_4)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3))\}),$
$((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})\}$
$\mathtt{St}_2 : \{[(\mathtt{q}_3, 0), (\mathtt{p}_2, 0)], [(\mathtt{q}_3, 0), (\mathtt{q}_4, 0), (\mathtt{p}_2, 0), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)],$
$[(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}$

Im nächsten Durchgang der `while`-Schleife ist noch einmal $(q_1', q_2') = (p_2, q_3)$, weshalb nicht (24) sondern (28) erfüllt ist. Dadurch wird dieses Paar in R eingefügt und die Werte in M auf 1 gesetzt. Damit wird $\mathtt{M} : \{(\mathtt{q}_3, 0), (\mathtt{p}_2, 0)\}$ zu $\mathtt{M} : \{(\mathtt{q}_3, 1), (\mathtt{p}_2, 1)\}$ und der Durchgang ist beendet. Die Variablen haben nun folgende Werte:

$\mathtt{V} : \{\}$
$\mathtt{R} : \{(\mathtt{p}_2, \mathtt{q}_3)\}$
$\mathtt{W} : \{\}$
$\mathtt{St}_1 : \{((\mathtt{p}_2, \mathtt{q}_3), \{\}), ((\mathtt{p}_1, \mathtt{q}_1), \{(\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_4)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3))\}), ((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})\}$
$\mathtt{St}_2 : \{[(\mathtt{q}_3, 1), (\mathtt{p}_2, 1)], [(\mathtt{q}_3, 0), (\mathtt{q}_4, 0), (\mathtt{p}_2, 0), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)],$
$[(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}$

Im nächsten Durchgang ist `l` nun leer, wodurch statt Zeile (20) Zeile (37) ausgeführt wird. Dazu wird die oberste Map aus $\mathtt{St}_2$ gelöscht und (p_2, q_3) in die Menge der besuchten Zustände, V, eingefügt. $\mathtt{M}'$ stellt nun die oberste Map von $\mathtt{St}_2$ dar, also $\mathtt{M}' : \{(\mathtt{q}_3, 0), (\mathtt{q}_4, 0), (\mathtt{p}_2, 0), (\mathtt{p}_3, 0)\}$, während M weiterhin so aussieht, wie oben beschrieben ($\mathtt{M} : \{(\mathtt{q}_3, 1), (\mathtt{p}_2, 1)\}$). Da beide Werte in M gleich 1 sind, ist die Bedingung von (41) wahr und diese Werte können in M' übertragen werden. Die Variablen haben nach diesem Durchgang der `while`-Schleife nun folgende Werte:

$\mathtt{V} : \{(\mathtt{p}_2, \mathtt{q}_3)\}$
$\mathtt{R} : \{(\mathtt{p}_2, \mathtt{q}_3)\}$
$\mathtt{W} : \{\}$
$\mathtt{St}_1 : \{((\mathtt{p}_1, \mathtt{q}_1), \{(\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_4)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3))\}), ((\mathtt{p}_0, \mathtt{q}_0), \{(\mathtt{a}, (\mathtt{p}_1, \mathtt{q}_2))\})\}$
$\mathtt{St}_2 : \{[(\mathtt{q}_3, 1), (\mathtt{q}_4, 0), (\mathtt{p}_2, 1), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 0), (\mathtt{p}_1, 0)], [(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}$

Dieser Verlauf wird nun fortgesetzt für die nächsten Zustandspaare. Interessant wird es erst wieder, wenn an folgendem Punkt angelangt wurde:

$$\begin{aligned}
&\mathtt{V}: \{(\mathtt{p}_1, \mathtt{q}_1), (\mathtt{p}_2, \mathtt{q}_3), (\mathtt{p}_3, \mathtt{q}_4)\} \\
&\mathtt{R}: \{(\mathtt{p}_2, \mathtt{q}_3), (\mathtt{p}_3, \mathtt{q}_4)\} \\
&\mathtt{W}: \{\} \\
&\mathtt{St}_1: \{((\mathtt{p}_1, \mathtt{q}_2), \{(\mathtt{b}, (\mathtt{p}_2, \mathtt{q}_3)), (\mathtt{b}, (\mathtt{p}_2, \mathtt{q}_5)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_5))\}), ((\mathtt{p}_0, \mathtt{q}_0), \{\})\} \\
&\mathtt{St}_2: \{[(\mathtt{q}_3, 0), (\mathtt{q}_5, 0), (\mathtt{p}_2, 0), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 1), (\mathtt{p}_1, 1)], [(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}
\end{aligned}$$

Zu diesem Zeitpunkt ist der Algorithmus an der Stelle angelangt, dass in der folgenden `while`-Schleife die Nachfolger von (p_1, q_2) untersucht werden. Der erste Nachfolger ist (p_2, q_3), der schon in einem der obigen Schritte untersucht worden ist. Aus diesem Grund wird statt (22) auch Zeile (32) zu `true` ausgewertet und, da (p_2, q_3) aber keinen unsicheren Status hat, also nicht in `W` ist, wird auch Zeile (34) ausgeführt. Die Belegung der Variablen ist danach wie folgt:

$$\begin{aligned}
&\mathtt{V}: \{(\mathtt{p}_1, \mathtt{q}_1), (\mathtt{p}_2, \mathtt{q}_3), (\mathtt{p}_3, \mathtt{q}_4)\} \\
&\mathtt{R}: \{(\mathtt{p}_2, \mathtt{q}_3), (\mathtt{p}_3, \mathtt{q}_4)\} \\
&\mathtt{W}: \{\} \\
&\mathtt{St}_1: \{((\mathtt{p}_1, \mathtt{q}_2), \{(\mathtt{b}, (\mathtt{p}_2, \mathtt{q}_5)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_3)), (\mathtt{b}, (\mathtt{p}_3, \mathtt{q}_5))\}), ((\mathtt{p}_0, \mathtt{q}_0), \{\})\} \\
&\mathtt{St}_2: \{[(\mathtt{q}_3, 1), (\mathtt{q}_5, 0), (\mathtt{p}_2, 1), (\mathtt{p}_3, 0)], [(\mathtt{q}_2, 0), (\mathtt{q}_1, 1), (\mathtt{p}_1, 1)], [(\mathtt{p}_0, 0), (\mathtt{q}_0, 0)]\}
\end{aligned}$$

Die weitere Ausführung entspricht den vorgestellten Schritten bis alle Elemente des Stacks $\mathtt{St}_1$ untersucht worden sind. Dann wird die `while`-Schleife nicht nochmal ausgeführt, sondern zu Zeile (64) gesprungen. Dort wird `M` die oberste Map von $\mathtt{St}_2$ zugewiesen, wobei $\mathtt{St}_2$ auch lediglich nur noch diese Map enthält. Es ergibt sich somit $\mathtt{M}: \{(\mathtt{p}_0, 1), (\mathtt{q}_0, 1)\}$. Da beide Werte gleich 1 sind und sich die Variable `stable` während des ganzen Verlaufes der Ausführung nicht verändert hat, wird als Rückgabewert `TRUE` (69) zurückgeliefert.

Leider konnte kein Beispiel gefunden werden, welches ein unsicheren Status beinhaltet, wodurch auch die Zeilen (43) bis (47) ausgeführt worden wären. Für diesen Fall besteht weiterhin Testbedarf.

Tests

Zu dem `Bisimulation`-Modul gibt es insgesamt 15 Tests, wovon 6 auf Bisimulation und 9 auf Nicht-Bisimulation testen. Letztere Tests sind dabei weitaus interessanter, denn dort wird die Korrektheit des ausgegeben Fehlerpfades getestet. Unter den Tests befinden sich auch welche, die untersuchen, ob alle Eingabekombinationen auch das richtige Ergebnis liefern. Ebenfalls werden die Spezialfälle getestet, dass ein eingegebenes Petri-Netz leer ist oder dass ein Transitionssystem nur eine Stelle besitzt. Dieses soll sicher stellen, dass die oben genannte Modifikation der `getSuccessor`-Funktion auch das richtige Ergebnis liefert. Ebenso werden beide Algorithmen getestet, wobei der Code zu der determinitischen Variante komplett durchlaufen wird, welches bei der generellen Variante nicht der Fall ist. Dort wird nämlich nicht der Fall getestet, dass die `partial_DFS`-Funktion als Ergebnis `UNRELIABLE` zurückgibt und damit eine weitere Tiefensuche durchgeführt werden muss. Dies liegt daran, dass es keine ausreichend komplexen Test-Netze gab. Desweiteren werden alle Hilfsfunktionen durch die Tests durchlaufen, wodurch diese optimal getestet werden.

8 Benutzungsschnittstelle

In APT stellt die Benutzungsschnittstelle dem Benutzer den Zugriff auf die verfügbaren Module bereit. Dazu bedient sie sich direkt des im Abschnitt 6.2 vorstellten Modulsystems. Ein großer Teil der Benutzungsschnittstelle wird automatisch aus den verfügbaren Modulen erzeugt.

Um dem Benutzer die Arbeit mit dem System zu erleichtern, basiert die Benutzungsschnittstelle auf der Idee, dass alle Funktionalitäten intuitiv bedient werden können; dies bezieht sich vor allem auf die verfügbaren Module und deren Nutzung.

Die Benutzung von APT basiert auf folgendem Befehlsschema.

```
apt [module_name] [arguments] [optional_arguments] [file_arguments]
```

Der Aufruf eines Moduls erfolgt über den Namen des Moduls (dargestellt durch `module_name`). Die Angabe des Namens ist optional. Wird dieser ausgelassen, zeigt das APT eine Übersicht aller verfügbaren Module inklusive ihrer Namen und Beschreibungen an. Jedes Modul ist einer oder mehreren Kategorien zugeordnet. Diese werden genutzt, um die Module übersichtlich und sortiert nach Kategorien anzuzeigen.

Ein Modul kann zudem über einen eindeutigen Präfix des Namens aufgerufen werden (z. B. `coverab` statt `coverability`).

Das Auslassen von Parametern beim Aufruf des Programms stellt im Allgemeinen keinen Fehler dar. Stattdessen wird der Benutzer darüber informiert, welche Parameter fehlen bzw. zulässig sind. Dadurch bietet das System eine eingebaute Hilfe, die

den Benutzer mit weiteren Informationen versorgt.

Ähnlich wie beim Auslassen des Namens eines Moduls, zeigt das Auslassen der Argumente (dargestellt durch `arguments`) die Beschreibung des entsprechenden Moduls an. Die Beschreibung enthält die benötigten sowie optionalen Parameter des Moduls. Mithilfe dieser Beschreibung sollte es dem Benutzer möglich sein, das Modul korrekt zu nutzen.

Anders als die benötigten Parameter können die optionalen Parameter (dargestellt durch `optional_parameters`) entfallen; das angegebene Modul wird dennoch aufgerufen und die optionalen Parameter bekommen implizit, wie in Kapitel 6.2 beschriebene, Standardwerte. Optionale Parameter verändern üblicherweise die Einstellungen des genutzten Algorithmus oder verwenden gar einen alternativen Algorithmus.

Die Datei-Parameter (dargestellt durch `file_arguments`) ermöglichen es, bestimmte Rückgabewerte der Ausgabe eines Moduls in eine Datei zu schreiben, anstatt diese dem Benutzer anzuzeigen; dies geschieht in APT insbesondere bei Rückgabewerten vom Typ *Petri-Netz* und *beschriftetem Transititionssystem*, da diese häufig als Eingabe für weitere Aufrufe von Modulen genutzt werden.

Da der Benutzer die verschiedenen Parameter über die Kommandozeile eingibt, liegen diese lediglich als Zeichenketten vor. Diese Zeichenketten müssen zunächst in die erwarteten Typen umgewandelt werden, bevor diese schließlich an das aufgerufene Modul weitergereicht werden können. Ähnliches gilt für die Rückgabewerte eines Moduls, welche für die Ausgabe auf der Kommandozeile in Zeichenketten umgewandelt werden.

Für die verschiedenen Typen wurden daher möglichst intuitive, von der mathematischen Notation inspirierte Darstellungen bestimmt.

Eine Auswahl häufig genutzter Darstellungen von Parametern und Rückgabewerten ist Tabelle 8.1 zu entnehmen. Alle Parameter, die Leerzeichen oder spezielle Zeichen wie etwa das Semikolon enthalten, müssen mit Anführungszeichen umschlossen werden.

Typ	Darstellung	Beispiel
Zeichen(-kette)	`"string"`	`"Beispiel"`
Wort	$\texttt{label}_1\texttt{,...,label}_n$	`a,b,c`
Zahl	`n`	7
Parikh-Vektor	$\texttt{\{label}_1\texttt{=i}_1\texttt{,...,label}_n\texttt{=i}_n\texttt{\}}$	`{t3=1, t2=0, t1=1}`
Wahrheitswert	`Yes` für wahr, `No`, für falsch	–
Zustand	`state`	`s1`
Stelle	`place`	`p1`
Transition	`transition`	`t1`
Markierung	$\texttt{place}_1\texttt{:i}_1\ \texttt{...}\ \texttt{place}_n\texttt{:i}_n$	`p1:2 p2:1`
Knoten	`node`	`n1`
Schaltfolge	$\texttt{transition}_1\texttt{;...;transition}_n$	`t1;t2;t3`
	$\texttt{transition}_1\ \texttt{...}\ \texttt{transition}_n$	`t1 t2 t3`
Liste	$\texttt{[item}_1\texttt{, ..., item}_n\texttt{]}$	`[id1, id2, id3]`

Tabelle 8.1: Eine Auswahl häufig genutzter Darstellungen von Typen

Komplexere Typen wie Petri-Netze und beschriftete Transitionssysteme werden nicht über die Kommandozeile eingegeben, sondern in einem dafür entwickelten Dateiformat beschrieben; dieses wird in Abschnitt 6.3 erklärt. Lediglich der Name einer solchen Datei wird als Parameter an APT übergeben. Gibt der Benutzer als Namen „-“ ein, wird der Parameter statt aus einer Datei von der Standard-Eingabe gelesen.

Nachdem das aufgerufene Modul alle vom Benutzer übergebenen Parameter als korrekte Typen erhalten hat, wird es ausgeführt. Ist die Ausführung des Moduls beendet, werden dem Benutzer die Rückgabewerte des Moduls auf der Kommandozeile in geeigneter Darstellung angezeigt. Die Ausgabe besteht üblicherweise aus den Namen und den Darstellungen aller Rückgabewerte.

Der Aufruf des Moduls `BoundedModule` verdeutlicht, wie die Ausgabe von Modulen aussehen kann.

```
bounded: No
witness_place: s3
witness_firing_sequence: "t1;t2;t3"
```

Die Ausgabe des Moduls `BoundedModule` auf der Kommandozeile zeigt drei verschiedene Rückgabewerte. Der Rückgabewert `bounded`, der einen Wahrheitswert darstellt, zeigt hier den Wert `No` und damit, dass die Beschränktheit des übergebenen Petri-

Netzes nicht gegeben ist. Zudem wird mit `witness_place` vom Typ Stelle sowie `witness_firing_sequence` vom Typ Schaltfolge zusätzliche Information zur Beschränktheit bereitgestellt.

Falls ein Rückgabewert für die Speicherung in einer Datei vorgesehen ist und der dazugehörige Dateiname als Teil der `file_arguments` übergeben wurde, wird dieser dem Benutzer nicht auf der Kommandozeile angezeigt, sondern direkt in eine Datei mit übergebenem Namen geschrieben. Gibt der Benutzer als Namen „-“ ein, wird der Rückgabewert stattdessen auf der Standard-Ausgabe ausgegeben.

Während das anfängliche Ziel der Benutzungsschnittstelle die intuitive Nutzung war, zeigte sich, dass der modulare Aufbau von APT genutzt werden könnte, Aufgaben zu automatisieren, die über die einzelnen Module hinaus gehen. Als weiteren Ziel galt somit die Benutzungsschnittstelle auch ohne interne Kenntnisse von APT durch den Benutzer programmierbar zu machen.

Die Programmierbarkeit von APT wird im Wesentlichen durch das Lesen von der Standard-Eingabe, dem Schreiben auf der Standard-Ausgabe sowie dem Rückgabewert von APT, der abhängig vom Erfolg eines Modules gesetzt wird, ermöglicht.

Aufrufe von APT können durch das Lesen von der Standard-Eingabe und Schreiben auf der Standard-Ausgabe beliebig ohne Zwischenschritte verkettet werden. Dies bietet sich an, wenn das Ergebnis eines Moduls nur benötigt wird, um ein weiteres Modul mit diesem aufzurufen. Eine typische Anwendung ist das Zeichnen von Petri-Netzen und beschrifteten Transitionssystemen.

Das Modul `CoverabilityModule` berechnet den Erreichbarkeitsgraphen eines Petri-Netzes. Statt den Erreichbarkeitsgraphen zunächst in einer Datei zu speichern, um ihn dann mithilfe des Moduls `DrawModule` zu zeichnen, kann dieser Zwischenschritt entfallen, wenn die Aufrufe der Module verkettet werden.

Die Verkettung von Aufrufen von Modulen ist beispielsweise in der Programmiersprache der Bash-Shell folgendermaßen möglich.

```
apt coverability_graph Petri-Netz.apt - | apt draw - LTS.dot
```

Die Verkettung ruft zunächst das Modul `CoverabilityModule` für das Petri-Netz in der Datei „Petri-Netz.apt“ im APT-Format auf. Die Ausgabe des Moduls ist ein beschriftetes Transitionssystem im APT-Format, das nicht in eine Datei gespeichert wird, sondern unter Angabe von „-“ auf der Standard-Ausgabe ausgegeben wird.

Dann wird das Modul `DrawModule` erneut unter Angabe von „-“ diesmal auf das beschriftete Transitionssystem, das von der Standard-Eingabe gelesen wird, aufgerufen. Das Modul zeichnet das übergebene Transitionssystem und speichert es im Dot-Format in der Datei „LTS.dot“. Durch die Verkettung konnte auf die Speicherung des LTS im APT-Format verzichtet werden.

Der Rückgabewert von APT kann genutzt werden, um APT in andere Programme zu integrieren. Ein Aufruf kann beispielsweise als Bedingung für eine Verzweigung innerhalb eines Programms genutzt werden. In der Programmiersprache der Bash-Shell könnte APT in Bedingungen folgendermaßen genutzt werden.

```
if apt bounded Petri-Netz.apt > /dev/null; then
  echo "Das Petri-Netz ist beschränkt"
else;
  echo "Das Petri-Netz ist nicht beschränkt"
fi;
```

Das Programm ruft APT und damit das Modul `BoundedModule` auf, welches das Petri-Netz in der übergebenen Datei auf Beschränktheit prüft. Ist das Petri-Netz beschränkt, wird der Satz „Das Petri-Netz ist beschränkt“ ausgegeben, andernfalls der Satz „Das Petri-Netz ist nicht beschränkt“. Die Angabe von „/dev/null“ sorgt dafür, dass dem Benutzer die Ausgabe von APT nicht gezeigt wird, sondern die Ausgabe des Programms in dem APT verwendet wird.

9 Qualitätssicherung

Bei einem Projekt zur Entwicklung einer Software ist es erforderlich, darauf zu achten, dass die entstehende Software die durch die Anforderungen gegebenen Qualitätskriterien erfüllt, da ansonsten bereits in der Entwicklung und auch später Probleme auftreten können. Dieses Kapitel beschreibt daher, wie die Qualität von APT gesichert wurde.

9.1 Tests

Zur Prüfung auf Korrektheit der Implementierung ist es wichtig, wiederholbare und automatisch ausführbare Testfälle zur Hand zu haben, deren Ergebnis bekannt ist. Wenn diese vor der Implementierung der eigentlichen Funktionalität eines Moduls geschrieben werden, ist es möglich, zu prüfen, ob die Implementierung sich korrekt verhält. Auch später implementierte Tests haben in vielen Fällen Fehler in der Implementierung aufgezeigt, die ansonsten möglicherweise überhaupt nicht oder erst sehr viel später aufgefallen wären.

Ein anderer wichtiger Aspekt dieser Tests ist es, sie aufgrund ihrer Wiederholbarkeit jederzeit ausführen zu können und so erkennen zu können, ob durch Änderungen an der Software Fehler aufgetreten sind, die vorher nicht vorhanden waren.

Um die Korrektheit eines Moduls systematisch mit Tests überprüfen zu können, ist es wichtig, eine ausreichende Testabdeckung vorweisen zu können. Dieses bedeutet, dass sowohl hinreichend viele typische, als auch insbesondere spezielle Testfälle geschrieben werden müssen.

Während der Umsetzungsphase wurde zunächst nicht für alle Teile von APT eine ausreichende Bandbreite an Tests geschrieben. Da allerdings häufig durch manuelle Tests der Teilnehmer und Betreuer Gegenbeispiele und Sonderfälle für einige Module gefunden wurden, wurden die Tests stetig erweitert. Leider konnte nicht mit Tools geprüft werden, wie gut die Test die verschiedenen Teile des Programmcodes abdecken, da die Code-Coverage-Tools, die man hierfür verwenden müsste, noch kein Java 7 unterstützen.

Wie bereits im vorherigen Abschnitt erwähnt, wurden neben den automatisierten Tests auch manuelle Tests durchgeführt. Hierzu haben sowohl die Betreuer als auch die Mitglieder der Projektgruppe das Werkzeug direkt aufgerufen und angesehen, wie es sich für bestimmte Beispiele verhielt. Hierbei konnte sehr gut das Prinizp beobachtet werden, dass es sinnvoll ist, wenn eine andere Person als derjenige der ein bestimmtes Modul implementiert hat, dieses mit manuellen Tests überprüft. Auf diesem Weg wurden viele Sonderfälle und Gegenbeispiele entdeckt, die bisher bei den automatisierten Tests übersehen wurden. Durch diese weiteren Beispiele wurde die Bandbreite der automatisierten Testfälle stetig erweitert, so dass auch die Funktionalität des Werkzeuges zunehmend stabiler wurde. Weitere Informationen zur Testabdeckung können direkt in den Abschnitten, in welchen die Analyse-Algorithmen beschrieben werden, nachgelesen werden (siehe Seite 99 ff.).

9.2 Kontinuierliche Integration

Um sicherzustellen, dass die für alle Mitglieder der Projektgruppe in Git verfügbare aktuelle Entwicklungsversion von APT zu jedem Zeitpunkt kompilierbar ist, wird das Continuous-Integration-Tool *Jenkins* [1] in der Version 1.483 genutzt. Dieses startet nach jeder Aktualisierung im Git-Repository einen Versuch das gesamte Projekt zu kompilieren, und bietet über ein Webinterface den Zugriff auf dabei auftretende Fehler beziehungsweise Warnungen.

Außerdem wurden auch die Tests automatisch ausgeführt, damit Fehler schneller gefunden werden können und einfach über das Jenkins-Webinterface sichtbar ist, wie

[1] `http://jenkins-ci.org/`, zuletzt aufgerufen am 26. 03. 2013

lange einzelne Fehler schon auftreten. Ebenso werden die Javadoc-Kommentare ausgewertet und es wird eine Webseite generiert, die die Javadoc-Dokumentation enthält.

9.3 Statische Code-Analyse

Weitere Qualitätssicherungsmaßnahmen sind verschiedene Formen der statischen Code-Analyse, also des Betrachten des Programmcodes, ohne ihn auszuführen. Hierfür sind in Jenkins die Tools *Checkstyle*[2] in der Version 5.6 und *Findbugs*[3] in der Version 2.0.1 eingebunden. Checkstyle prüft den Code auf Übereinstimmung mit einer von der Projektgruppe zu Beginn definierten Coding-Style-Richtlinie, die im folgenden Abschnitt erläutert wird.

9.3.1 Coding-Style

Der Coding-Style in diesem Projekt entspricht im Wesentlichen dem allgemein üblichem Java Coding Style, unter anderen sind die folgenden Punkte :

- Zeilen dürfen bis zu 120 Zeichen lang sein.
- Die Einrückung muss mit Tabulatoren erfolgen.
- Zeilenenden müssen dem Unix-Standard entsprechen (kein Leerzeichen am Ende einer Zeile)

Durch diesen Coding Style ist sichergestellt, dass der Code einheitlich aussieht und so jeder Betrachter des Codes jeden Teil des Programmcodes lesen kann, ohne sich ständig in verschiedene Coding-Styles einlesen zu müssen. Findbugs prüft hingegen den Programmcode auf typische Fehler, die zu unerwartetem und unerwünschtem Verhalten führen, wenn der Programmcode ausgeführt wird.

[2] `http://checkstyle.sourceforge.net/`, zuletzt aufgerufen am 26. 03. 2013
[3] `http://findbugs.sourceforge.net/`, zuletzt aufgerufen am 26. 03. 2013

9.3.2 Code-Review

Eine andere Form der statischen Code-Analyse sind Code-Reviews, bei denen sich Projektgruppenmitglieder Code-Fragmente ansehen, die sie nicht selber geschrieben haben. Falls die Code-Dokumentation nicht gut verständlich ist, oder andere Qualitätskriterien nicht erfüllt werden, übermittelt das lesende Projektgruppenmitglied entsprechende Anmerkungen an die Autoren des Codes.

Zu Beginn der Implementierungsphase wurde ein Code-Review durchgeführt, bei dem jedes Projektgruppemitglied involviert sein sollte. Die Anmerkungen zum Code wurden zur möglichst übersichtlichen Darstellung im Projektmanagement-Tool notiert. Ein Review mit allen Teilnehmern macht zum einen aus Gründen der Arbeitsteilung Sinn und hat weiterhin den Vorteil, dass jeder Teilnehmer sich aktiv mit den Coding-Styles auseinander setzt. Dieser Code-Review wurde mit Absicht sehr früh in der Implementierungsphase durchgeführt, da es sinnvoll ist, fehlerhaften Coding-Style früh zu entdecken, damit im weiteren Verlauf der Implementierung auf Anhieb weniger Fehler auftreten.

Bei diesem Review wurde festgestellt, dass die Projektgruppenmitglieder zum einen mit einer verschieden starken Intensität über den Code geschaut haben und dass zum anderen die Anmerkungen verschieden detailliert weitergegeben wurden. Auch war der Zeitraum des Code-Reviews relativ groß, da jeder Teilnehmer ein individuelles Tempo beim Gegenlesen hatte und zudem von Seiten der jeweiligen Code-Autoren einige Wochen nötig waren, um die Code-Anmerkungen einzuarbeiten, oder zu Begründen, warum die Anmerkungen nicht umsetzbar sind.

Um den Code-Review einheitlicher durchzuführen, wurde später in der Implementierungsphase ein Code-Review von einer kleineren Gruppe, den Mitgliedern der Gruppe Qualitätssicherung, durchgeführt. Die Anmerkungen aus diesem Code-Review wurden direkt an die Autoren des jeweiligen Codes weitergegeben. Der Vorteil hierbei war, dass eine kleinere Experten-Gruppe gezielter und organisierter Vorgehen kann.

Insgesamt ist festzustellen, dass der gemeinsame Code-Review sehr gut als Lerneffekt für alle Teillnehmer war, da sich jeder zu Beginn der Programmierphase aktiv mit den Coding-Style-Regeln auseinandersetzen musste. Produktiver war allerdings der

Code-Review der Qualitätsmanager, weswegen für zukünftige Projekte diese Form des Code-Reviews zu empfehlen ist. Eine weitere Erkenntnis ist, dass ein Code-Review so früh wie möglich durchgeführt werden sollte, da ein ausführlicher Review viel Zeit in Anspruch nimmt und es in einem Projekt – mit vielen involvierten Personen – wichtig ist, dass von Anfang an auf Einhaltung der Coding-Styles geachtet wird.

10 Fazit, Reflexion und Ausblick

In diesem Kapitel soll zunächst reflektiert und bewertet werden, was die Projektgruppe APT erreicht hat und welche Probleme während der Entwicklung des Analysewerkzeugs aufgetreten sind. Anschließend wird ein kleiner Ausblick darauf gewagt, welche Weiterentwicklungs- und Nutzungsmöglichkeiten in der Zukunft denkbar sind.

10.1 Reflexion und Fazit

Blick zurück: Im vorliegenden Endbericht wurden die Projektarbeit sowie das entwickelte Analysewerkzeug der Projektgruppe „APT“ beleuchtet. Neben der Beschreibung des Projektablaufes wurden zu Anfang die Grundlagen in Bezug auf beschriftete Transitionssysteme und beschriftete und unbeschriftete Petri-Netze geschaffen. Danach wurden die funktionalen und nichtfunktionalen Anforderungen vorgestellt, die das Analysewerkzeug umsetzen soll, und Fremdsysteme in Bezug auf Verwendbarkeit analysiert. Im Folgenden wurde auf Implementierungsdetails und auf die eingesetzten Analyse-Algorithmen eingegangen, bevor die Benutzungsschnittstelle erläutert und schließlich die Techniken zur Qualitätssicherung vorgestellt wurden.

Grundlegende Entscheidungen: Die Projektgruppe hat sich gegen die Weiterentwicklung eines bestehenden Systems und für eine eigene Entwicklung entschieden, da die Betrachtung der Fremdsysteme zeigte, dass sich die verschiedenen Systeme nicht direkt zur Weiterentwicklung eigneten. Entweder konnten diese nur wenige der geforderten Eigenschaften überprüfen oder sie waren nicht funktionstüchtig. Zwei der Fremdsysteme, die Tools Synet und Petrify, konnten aber zumindest eingebunden werden und das Tool PEP sowie die studentischen Arbeiten von Robert Bleiker und

Florian Hinz konnten als Ideensammlung für die Umsetzung einiger Algorithmen verwendet werden.

Ein Vorteil dadurch, dass wir – angefangen bei einem eigenen Dateiformat, bis hin zu eigenen Datenstrukturen – ein komplett neues Analysewerkzeug für Petri-Netze und beschriftete Transitionssysteme geschrieben haben, ist, dass alle Teilnehmer einen Einblick in die Planung, Strukturierung und Entwicklung eines autarken Software-Systems bekommen haben.

Zeitkomplexität vs. Speicherplatz: Bei der Umsetzung der einzelnen funktionalen Anforderungen hat sich die Projektgruppe bemüht, die entsprechenden Algorithmen möglichst platz- und zeitsparend zu implementieren, welches bei einigen Algorithmen allerdings nur unzureichend möglich war.

In vielen Fällen haben wir Speicherplatz- vor Zeitkomplexität gesetzt, da es unter Umständen einfacher ist, etwas mehr Zeit zu benötigen, als wegen zu vieler gespeicherter Zwischendaten den Speicher zu sprengen. Begründet werden kann dieses auch damit, dass das Werkzeug auch auf „normalen“ Heimrechnern funktionieren soll, die nicht über einen überdurchschnittlich großen Arbeitsspeicher verfügen.

Eigenes Modulsystem: Als besonders gut stellte sich die Design-Entscheidung für das Modulsystem heraus. Damit war es möglich, dass die verschiedenen Algorithmen bis auf wenige Ausnahmen parallel entwickelt werden konnten, ohne dass ein noch nicht funktionierender Algorithmus die Funktionalität eines anderen Algorithmus beeinträchtigte. Zudem konnten zu jedem Zeitpunkt des Projektes weitere Algorithmen einfach implementiert und eingebunden werden.

Qualitätsmanagement: Die nichtfunktionalen Anforderungen zur Qualitätssicherung konnten erfolgreich umgesetzt werden (beispielsweise korrekter Coding-Style, Kommentare mit JavaDoc). Hierdurch konnte nicht nur eine optimale Handhabung des Analysewerkzeugs durch die Anwender sichergestellt werden, sondern auch die langfristige Wartung und Weiterentwicklung des Produkts vereinfacht werden.

Die qualitätssichernde Maßnahme des Testens erwies sich als positiv, da sich während der Implementierungsphase beispielsweise zeigte, dass bei der Benutzung der

bis dahin entwickelten Datenstrukturen durch eine zu große Anzahl an Interfaces Missverständnisse entstanden. So konnten die Datenstrukturen noch während der Entwicklungsphase vereinfacht werden, wodurch die Benutzung verbessert wurde.

Fairerweise muss an dieser Stelle erwähnt werden, dass zu Beginn der Implementierung das Testen zum Teil unterschätzt wurde. Schnell konnte aber in vielen Modulen festgestellt werden, dass durch die vielen Unit-Tests und manuellen Tests durch Betreuer und Teilnehmer viele Fehler sehr früh entdeckt und behoben werden konnten. Dadurch wurden immer mehr Tests geschrieben und es konnte letztendlich eine größtenteils gute Testabdeckung erreicht werden, die sicherstellt, dass auch weiterhin alle Bestandteile korrekt funktionieren.

Ablauf der Projektarbeit: Für die Projektarbeit halfen die regelmäßigen Treffen, einen Überblick über das ganze Projekt zu behalten, da die einzelnen Implementierungs- und Dokumentationsaufgaben in Kleingruppen bearbeitet wurden. Außerdem konnten so Fragen und Wünsche von den Betreuern im Plenum besser beantwortet werden. Zudem konnten in den Sitzungen wichtige Projekt-Entscheidungen diskutiert und beschlossen werden und es gab Raum für gesellige Spieleabende und PG-Essen.

Abseits der regelmäßigen Treffen fiel anfangs auf, dass Unklarheiten über Absprachen vorherrschten (welche Version der Anforderungen die zu nutzende ist, wie einzelne Abschnitte des Programms strukturiert werden sollen, etc.). Wir haben allerdings recht schnell angefangen, das durch das Projektmanagement-Tool bereitgestellte Wiki zu nutzen, um Protokolle und Aufteilungen in Arbeitsgruppen dort zu notieren und auch wichtige Dokumente sowie sehr schöne Tutorials – wie beispielsweise eine detaillierte Beschreibung wie das Modulsystem angewendet wird – dort zu hinterlegen.

Für eine zukünftige Projektarbeit bleibt festzustellen, dass es wichtig ist, verbindliche Absprachen aber auch nützliche Hilfen früh an einem für alle Teilnehmer leicht zugänglichen Ort zu notieren. Die Entscheidung, dieses über das Wiki zu regeln, war in unserem Fall sehr gut. Darüber hinaus stellte es sich als gut heraus, für die einzelnen Arbeitsgruppen und Aufgaben einen direkten Ansprechpartner zu haben, um sich bei Fragen, Problemen und Wünschen direkt an die zuständige Person wenden zu können.

Fazit: Insgesamt ist durch eine erfolgreiche Zusammenarbeit ein funktionstüchtiges Analysewerkzeug entstanden. Dieses kann nun in Zukunft beliebig erweitert werden und sowohl in der Lehre als auch in der Praxis zur Erstellung von Transitionssystemen und Petri-Netzen und zur Untersuchung von Eigenschaften dieser eingesetzt werden.

10.2 Ausblick

Eine interessante Weiterentwicklungsmöglichkeit des Analysewerkzeugs könnten Aufgabenstellungen sein, die den Anforderungen E1 bis G2 ähneln. Damit ist gemeint, dass neue Module entwickelt werden könnten, die mit vorhandenen Analyse-Algorithmen von APT arbeiten und versuchen, Petri-Netze oder Transitionssysteme mit bestimmten Eigenschaften automatisch zu ermitteln.

In der Reflexion im vorherigen Abschnitt wurde bemerkt, dass einige Algorithmen über eine schlechte Speicherplatz- und/ oder Zeitkomplexität verfügen. Eine sinnvolle Weiterentwicklungsidee könnte daher auch sein, Optimierungsmöglichkeiten für vorhandene Algorithmen zu suchen, da es möglich ist, dass zu den genutzten Algorithmen in ein paar Monaten oder Jahren verbesserte Versionen oder andere Algorithmen mit besseren Komplexitätsklassen vorhanden sind.

Damit APT in der andauernd stattfindenden rasanten Weiterentwicklung nahezu aller Teilbereiche der Informatik auch in einigen Jahren noch aktuell ist, ist eine oben beschriebene Wartung der in APT eingebetteten Algorithmen empfehlenswert, damit es zu einem Werkzeug werden kann, was stets mit dem aktuellen Forschungsstand im Gebiet der Petri-Netze und Transitionssysteme mithalten kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

Abkürzungsverzeichnis

BCF	behaviourally conflict-free
BiCF	binary conflict-free
DEA	Deterministischer endlicher Automat
LPN	Beschriftetes Petri-Netz
LTS	Beschriftetes Transitionssystem
NEA	Nichtdeterministischer endlicher Automat
TS	Transitionssystem

Übersetzungsliste Deutsch-Englisch

beschriftetes Petri-Netz	labelled Petri net
beschriftetes Transitionssystem	labelled transition system
beschränkt	bounded
deterministisch	deterministic
einfach lebendig	simply live
Erreichbarkeitsgraph	reachability graph
Erreichbarkeitsmenge	reachability set
Falle	trap
Feuersequenz	firing sequence
Inzidenzmatrix	incidence matrix
isoliert	isolated
isomorph	isomorphic
k-Markierung	k-marking
k-Netz	k-net
konfliktfrei	conflict-free
lebendig	live
Nebenbedingung	side condition
Parikh-Vektor	Parikh vector
Parikh-äquivalent	Parikh-equivalent

Petri-Netz	Petri net
pur	pure
reversibel	reversible
Rückwärtsmatrix	backward matrix
S-Invariante	S-invariant
S-Netz	S-net
Schaltfolge	execution sequence
schlicht	plain
schwach k-separierbar	weakly k-separable sort
schwach lebendig	weakly live
schwach zusammenhängend	weakly connected
sicher	safe
sprachäquivalent	language-equivalent
stark k-separierbar	strongly k-separable
stark lebendig	strongly live
stark zusammenhängend	strongly connected
T-Invariante	T-invariant
T-Netz	T-net
tot	dead
total erreichbar	totally reachable
Vorwärtsmatrix	forward matrix
Überdeckungsgraph	coverability graph

Literaturverzeichnis

[B+12a] Best, Eike u. a.: *Grundlegende Spezifikationen für APT / Teil 1.* E-Mail vom 26. 7. 2012, 2012

[B+12b] Best, Eike u. a.: *Grundlegende Spezifikationen für APT / Teil 2.* E-Mail vom 6. 9. 2012, 2012

[BCC+03] Bloom, James ; Clark, Clare ; Clifford, Camilla ; Duncan, Alex ; Khan, Haroun ; Papantonious, Manos: Final Report. In: *Platform Independent Petri-net Editor*, 2003, S. 17–21

[BD11] Best, Eike ; Darondeau, Philippe: Separability in Persistent Petri Nets. In: *Fundamenta Informaticae* Bd. 112, IOS Press, 2011, S. 1–25

[Bes12] Best, Eike: *Transitionsbeschriftete Petri-Netze (LPN) und beschriftete Transitionssysteme (LTS)*, Carl von Ossietzky Universität Oldenburg, Foliensatz für die PG APT am 24. April und 8. Mai, 2012

[Ble09] Bleiker, Robert: *Das Wortproblem für Petri-Netze*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 2009

[BW12] Best, Eike ; Wimmel, Harro: *Skriptum zum Modul Petri-Netze*, Carl von Ossietzky Universität Oldenburg, Vorlesungsskript, 2012

[CFSV99] Cordella, L. P. ; Foggia, P. ; Sansone, C. ; Vento, M.: Performance Evaluation of the VF Graph Matching Algorithm. In: *Image Analysis and Processing, 1999. Proceedings.* Venice, 1999, S. 1172–1177

[CFSV01] Cordella, L. P. ; Foggia, P. ; Sansone, C. ; Vento, M.: An Improved Algorithm for Matching Large Graphs. In: *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen.* Cuen, 2001, S. 149–159

[DE95] Desel, Jörg ; Esparza, Javier: *Free Choice Petri Nets.* Cambridge : Cambridge University Press, 1995

[Dij71] Dijkstra, Edsger W.: Hierarchical Ordering of Sequential Processes. In: *Acta Informatica* Bd. 1, Springer Verlag, 1971, S. 115–138

[DT88] D'Anna, Mario ; Trigila, Sebastiano: Concurrent system analysis using Petri nets – an optimised algorithm for finding net invariants. In: *Computer Communications* Bd. 11, 1988

[Far02] Farkas, Julius: Theorie der einfachen Ungleichungen. In: *Journal für die reine und angewandte Mathematik* Bd. 124, 1902, S. 1–27

[FM90] Fernandez, Jean-Claude ; Mounier, Laurent: Verifying Bisimulations On the Fly. In: *Formal Description Techniques, III, Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE 90, Madrid, Spain, 5-8 November 1990*, North-Holland, 1990, S. 95–110

[FSV01] Foggia, P. ; Sansone, C. ; Vento, M.: A Performance Comparison of Five Algorithms for Graph Isomorphism. In: *in Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001, S. 188–199

[Hin12] Hinz, Florian: *Untersuchung und Implementierung von Algorithmen zur Isomorphie bei Petri-Netzen*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 2012

[Las09] Lasserre, Jean B.: Duality and a Farkas lemma for integer programs. In:

Optimization – Structure and Applications Bd. 32. New York : Springer, 2009, S. 15–39

[LBP10] Le Berre, Daniel ; Parrain, Anne: The Sat4j Library, release 2.2. In: *Journal on Satisfiability, Boolean Modeling and Computation* Bd. 7, 2010, S. 59–64

[MBS91] Marinescu, D.C. ; Beaven, M. ; Stansifer, R.: A parallel algorithm for computing invariants of Petri net models. In: *Petri Nets and Performance Models, Proceedings of the Fourth International Workshop on*, 1991, S. 136–143

[MS81] Martínez, Javier ; Silva, Manuel: A Simple and Fast Algorithm to Obtain All Invariants of a Generalized Petri Net. In: *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets.* London : Springer-Verlag, 1981, S. 301–310

[NFMS12] Nabli, Faten ; Fages, François ; Martinez, Thierry ; Soliman, Sylvain: A Boolean Model for Enumerating Minimal Siphons and Traps in Petri Nets. In: *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming.* Berlin, Heidelberg : Springer-Verlag, 2012, S. 798–814

[Por97] Portinale, Luigi: Modelling and Solving Constraint Satisfaction Problems through Petri Nets. In: *Lecture Notes in Computer Science* Bd. 1248. Berlin, Heidelberg : Springer, 1997, S. 348–366

[PW98] Priese, Lutz ; Wimmel, Harro: A Uniform Approach to True-Concurrency and Interleaving Semantics for Petri Nets. In: *Theoretical computer science* Bd. 206. Essex : Elsevier Science Publishers Ltd., 1998, S. 219–256

[Sta90] Starke, Peter H.: *Analyse von Petri-Netz-Modellen. Leitfäden und Monographien der Informatik.* Stuttgart : Teubner, 1990

[Wil12] WILKEIT, Elke: *Vorlesungsfolien der Veranstaltung „Algorithmische Graphentheorie“*, Carl von Ossietzky Universität Oldenburg, Vorlesungsskript, 2012

[Zha10] ZHANG, Yangzi: *Algorithmische Überprüfung struktureller Eigenschaften von Petrinetzen und Transitionssystemen*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 2010

Schlagwortverzeichnis