Ausblenden nicht-aktivierter Transitionen im Petri-Netz-Werkzeug PIPE

Erweiterung eines Programms zur Modellierung und Analyse von Petri-Netzen

Bachelor-Arbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.) im Studienfach Informatik

vorgelegt von: Kristof Kipp

Matrikelnummer: 3036176

Erstgutachterin: Dr. Sabine Kuske

Zweitgutachterin: Prof. Dr. Ute Bormann

eingereicht in Bremen, am 31. August 2018

Inhaltsverzeichnis

1. Einleitung 1

2. Petri-Netze 2

2.1. Einführung von Netzen . . . 2
2.2. Elementare Netzsysteme . . . 4
2.3. Stellen/Transitionssysteme . . . 6
2.4. Stellen/Transitionssysteme mit Inhibitorkanten . . . 9

3. Modellierung mit Petri-Netzen am Beispiel des Spiels Solitär 10

3.1. Das Spiel Solitär . . . 10
3.2. Formalisierung mit Elementaren Netzsystemen . . . 14

4. Modellierung von Solitär in PIPE 18

4.1. Vorstellung von PIPE . . . 18
4.2. Modellierung von Solitär mit inhibitorischen S/T-Systemen . . . 19
4.3. Algorithmische Generierung der Transitionen und Flusskanten . . . 21

5. Erweiterung von PIPE um das Ausblenden nicht-aktivierter Transitionen 27

5.1. Werkzeuge . . . 27
5.2. Analyse des Quelltextes und der Projektstruktur . . . 27

6. Validierung der Ergebnisse 39

6.1. Einfache Tests . . . 39
6.2. Überprüfung des Petri-Netzes für das Spiel Solitär . . . 41

7. Fazit und Ausblick 48

Literaturverzeichnis IV

Abbildungsverzeichnis VI

Quelltextverzeichnis VII

A. Anhang i

A.1. Beiliegende CD . . . i

1. Einleitung

Mit wachsender Beliebtheit von Petri-Netzen zur Modellierung von Geschäftsprozessen und Spielen steigen auch die Anforderungen an diese Modellierungen. Unter anderem werden Systeme mit einer hohen Anzahl an Kanten unübersichtlich, vor allem, wenn sich die gezeichneten Elemente überschneiden. Dies betrifft analoge Modellierungen mit Stift und Papier, sowie die digitale Modellierung mit Hilfe von Werkzeugen gleichermaßen. Eine mögliche Lösung im Rahmen der Eigenschaften der Petri-Netze ist die Betrachtung nicht-aktivierter Transitionen und das Ausblenden dieser Transitionen. In dieser Arbeit werden daher die folgenden Fragen bearbeitet:

- Erhöht die Ausblendung nicht-aktivierter Transitionen eines Netzes die Übersichtlichkeit der Simulation?
- Ist das Ausblenden nicht-aktivierter Transitionen eines Netzes immer hilfreich?

Zunächst werden die Grundprinzipien eines Petri-Netzes erklärt und das Spiel Solitär vorgestellt. Anschließend wird eine Verbindung zwischen der Spiellogik und den Methoden der Petri-Netze hergestellt, sowie ein den Spielregeln von Solitär entsprechendes Petri-Netz konstruiert. Dieses Petri-Netz wird anschließend im Petri-Netz-Werkzeug PIPE modelliert. Folgend wird die Implementierung der in der Zielsetzung definierten Änderungen am Programm PIPE beschrieben. Dies impliziert die Analyse des Quelltexts, die technische Erweiterung des Quelltextes des Programms, sowie die Auswahl von Methoden, mit der diese Erweiterungen zugänglich machen, ohne die User Experience des bereits bestehenden Programms einzuschränken.

Ziel dieser Arbeit ist zum einen die analoge Erzeugung eines Petri-Netzes für das Brettspiel Solitär, zum anderen die digitale Modellierung dieses Petri-Netz mit Hilfe des Petri-Netz-Werkzeugs PIPE. Dieses digitale Petri-Netz wird zusätzlich mit den Mitteln des Programms PIPE simuliert und anschließend validiert. Erwartet wird ein Petri-Netz mit einer niedrigen Anzahl an Stellen und einer hohen Anzahl an Transitionen, so dass die Darstellung des Netzes unübersichtlich wird. Um dieser Unübersichtlichkeit entgegenzuwirken, wird in diesem Zusammenhang das quelloffene Programm PIPE um die Funktion erweitert, nicht-relevante Elemente des Netzes ausblenden zu können.

2. Petri-Netze

Zunächst werden jene Konzepte von Petri-Netzen vorgestellt, auf die im weiteren Verlauf dieser Arbeit zurückgegriffen wird. Das Prinzip der Petri-Netze beruht auf der Dissertation von Carl Adam Petri (vgl. [Pet62]). Petri-Netze sind mathematisch fundierte Modelle, die primär der Modellierung und Validierung von nebenläufigen Prozessen dienen. Seit ihrer Einführung wurden sie in vielfältigen weiteren Aufgabengebieten wie z.B. der Modellierung von Geschäftsprozessen ([Gru96]) oder der Darstellung von Spiellogik am Beispiel einer Implementierung des Spiels Tic-Tac-Toe ([Sch16]) genutzt. Die hier vorgestellten Konzepte dienen dem Aufbau eines grundlegenden Verständnisses von Petri-Netzen und basieren auf den Ausarbeitungen und Definitionen von [PW08], [Bau97] und [Rei10].

2.1. Einführung von Netzen

Ein Netz besteht aus Stellen, Transitionen und einer Flussrelation. Es lässt sich mit einem gerichteten bipartiten Graphen darstellen, In diesem Graphen repräsentieren Stellen Objekte innerhalb eines Systems und werden durch einen Kreis dargestellt. Transitionen steuern die Inhalte der Objekte, die durch Stellen repräsentiert werden. Sie werden mit Rechtecken dargestellt. Die Flussrelation beschreibt gerichtete Kanten zwischen Stellen und Transitionen. Formal lässt sich ein Netz durch ein Tupel $N = (S, T, F)$ beschreiben, wobei

- S die endliche Menge von *Stellen*,
- T die endliche Menge von *Transitionen* mit $S \cap T = \emptyset$
- F die Flussrelation mit $F \subseteq (S \times T) \cup (T \times S)$ ist.

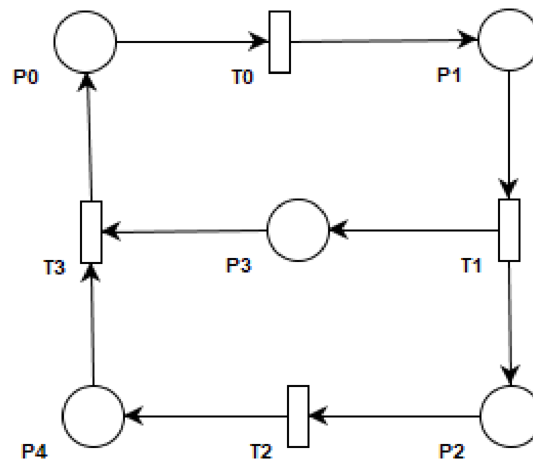


Abb. 2.1.: Einfaches Netzsystem NET_1

In Abbildung 2.1 wird ein einfaches Netzsystem in einer Graphendarstellung gezeigt. Die Komponenten dieses Netzes sind folgende:

- $S = \{P0, P1, P2, P3, P4\}$
- $T = \{T0, T1, T2, T3\}$
- $F = \{(P0, T0), (T0, P1), (P1, T1), (T1, P2), (T1, P3), (P2, T2), (T2, P4), (P4, T3), (P3, T3), (T3, P0)\}$

Für jedes Element in $T \cap S$ gibt es einen sogenannten Vor- und einen Nachbereich, wobei

$$^\bullet x = \{y \in (S \cup T) | (y, x) \in F\} \tag{2.1}$$

den Vorbereich einer Stelle oder Transitionen x und

$$x^\bullet = \{y \in (S \cup T) | (x, y) \in F\} \tag{2.2}$$

den Nachbereich von x angibt. In Abbildung 2.1 kann beispielhaft der Vorbereich der Transition $T3$ bestimmt werden. Dieser wird formal mit $^\bullet T3 = \{P3, P4\}$ beschrieben. Bei Stellen spricht man in der Regel von Vor- und Nachtransitionen einer Stelle $s \in S$, bei Transitionen von Vor- und Nachstellen einer Transition $t \in T$.

2.2. Elementare Netzsysteme

Ein Netz alleine ist unflexibel und beschreibt lediglich den Aufbau eines Systems, jedoch nicht seine Funktionsweise. Um die Dynamik eines Prozesses darstellen zu können, müssen diese Netze zunächst um eine grundlegende Komponente erweitert werden: die Markierung. Durch die Markierung und ihre Eigenschaften ergibt sich eine neue Kategorie von Netzen: die Elementaren Netzsysteme, kurz ENS. Eine Stelle in einem Elementaren Netzsystem kann eine sogenannte Marke (auch Token nach [PW08]) enthalten. Dargestellt wird diese Marke durch einen dicken Punkt in der Stelle. Sammelt man in einem ENS alle Stellen die einen Token enthalten, ergibt dies eine Teilmenge $M \subseteq S$ - diese Teilmenge nennt man Markierung. Ein Elementares Netzsystem ist ein Netz mit einer Markierung M_0, die die initiale Verteilung von Token festlegt. Beschrieben wird es durch das Tupel $ENS = (S, T, F, M_0)$, wobei

- (S, T, F) ein Netz nach Abschnitt 2.1 und
- $M_0 \subseteq S$ die Startmarkierung ist.

Abbildung 2.2 zeigt das bekannte Netz NET_1 mit der Startmarkierung $M_0 = \{P1, P2, P3\}$. Wir nennen dieses Elementare Netzsystem ENS_1.

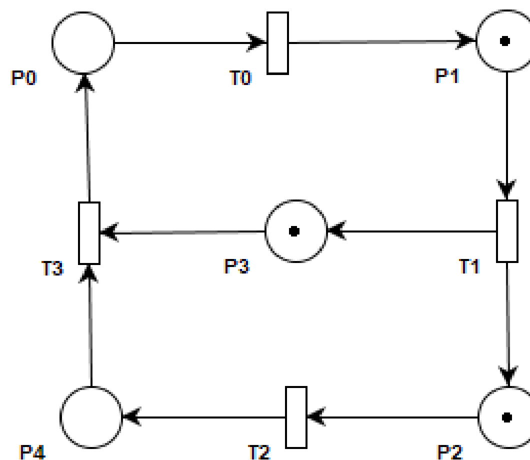


Abb. 2.2.: Elementares Netzsystem ENS_1

Schalten

Um Elementare Netzsysteme nun um Flexibilität und Dynamik zu erweitern, wird das sogenannte *Schalten* definiert. Beim Schalten werden Token von Stellen abgezogen und hinzugefügt. Um dieses Verhalten zu beschreiben, wird Vor- und der Nachbereich einer Transition betrachtet. Sind alle Vorstellen einer Transition mit einem Token belegt und alle Nachstellen derselben Transition nicht mit einem Token belegt, so kann diese Transition schalten. Formal ausgedrückt kann eine Transition $t \in T$ bei einer Markierung $M \in S$ schalten, wenn gilt

$$^\bullet t \subseteq M \text{ und} \tag{2.3}$$

$$t^\bullet \cap M = \emptyset. \tag{2.4}$$

In diesem Fall ist t eine M-aktivierte Transition.

Durch das Schalten (auch *Feuern*, nach [PW08]) einer Transition werden alle Token in den Vorstellen ($s \in {}^\bullet t$) entfernt und alle Nachstellen der Transition ($s \in t^\bullet$) mit einem Token belegt. Notiert wird das Schalten einer Transition mit $M[t\rangle M'$, wobei M' die Markierung nach dem Schalten der Transition t beschreibt, also $M' = (M \setminus {}^\bullet t) \cup t^\bullet$.

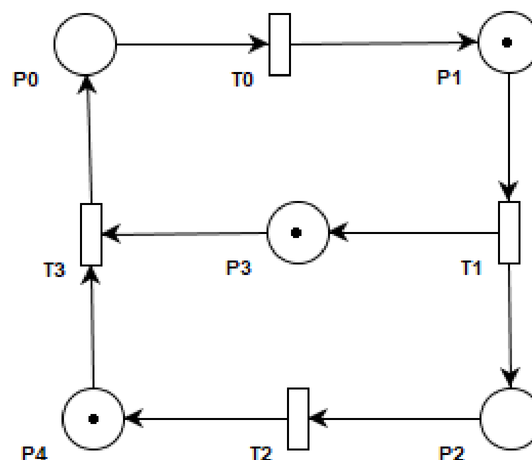


Abb. 2.3.: ENS_1 nach Schalten von T2

Abbildung 2.3 zeigt ENS_1 nach dem Schalten von $T2$. Somit gilt für ENS_1:

$$\{P1, P2, P3\}[T2\rangle\{P1, P3, P4\}$$

Es wurden alle Token aus den Vorstellen von $T2$ entfernt, während jede Nachstelle von $T2$ mit einem Token belegt wurde.

2.3. Stellen/Transitionssysteme

Im vorigen Kapitel führten die Elementare Netzsysteme eine erste Dynamik und Flexibilität in Netzen ein. Diese Netzsysteme werden nun um weitere Eigenschaften und Definitionen erweitert und Stellen/Transitions-Systeme (fortan mit S/T-System oder STS abgekürzt) genannt. Eine dieser Eigenschaften legt fest, dass bei Stellen/Transitions-Systemen die Nachstellen einer Transition nicht auf Leerheit überprüft werden, wodurch einer markierten Stelle weitere Token hinzugefügt werden können. Dadurch ist es möglich, in einem Stellen/Transitions-System eine Stelle mit mehr als einem Token zu belegen. Ein Stellen/Transitions-System besteht aus einem Netz, welches um eine Kantengewichtung und eine Markierung, die mehr als einen Token in einer Stelle erlaubt, erweitert wird. Formal wird ein S/T-System definiert als ein Tupel $STS = (S, T, F, W, M_0)$ mit:

- (S, T, F) ist ein Netz,
- $W : F \to \mathbb{N}_{>0}$ ist eine Abbildung, die jeder Kante eine positive natürliche Zahl als Kantengewichtung zuweist,
- $M_0 : S \to \mathbb{N}$ ist die Startmarkierung, die jeder Stelle eine Anzahl an Marken zuweist.

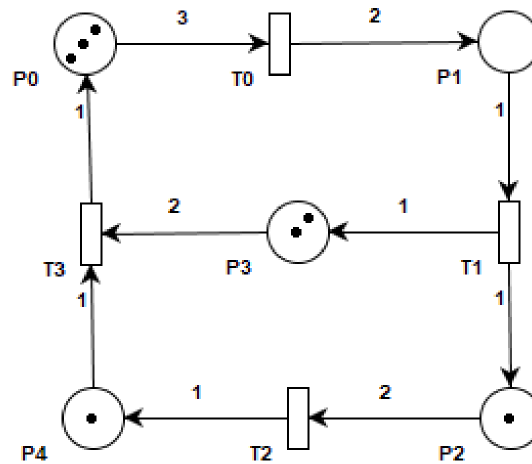


Abb. 2.4.: Stellen-/Transitions-System STS_0

Sind in einem STS die Stellen durchnummeriert, lässt sich jede Markierung auch als Tupel über den natürlichen Zahlen notieren. Die Markierung des Beispiel-Systems aus Abbildung 2.4 kann wie folgt notiert werden: $(3, 0, 1, 2, 1)$, wobei $P0$ die erste Stelle ist; $P1$ die zweite Stelle und alle weiteren nach dem gleichen Schema.

Schalten

Eine Transition t kann geschaltet werden, wenn sie in einer Markierung aktiviert ist. Sei $M : S \to \mathbb{N}$ eine Markierung und $t \in T$ eine Transition. Die Transition t ist M-aktiviert, falls gilt:

$$M(s) \geq W(s,t) \forall s \in {}^\bullet t$$

Ist t M-aktiviert, wird der Schaltvorgang wie bei Elementaren Netzsystemen mit $M[t\rangle M'$ beschrieben. M' ist die Folgemarkierung von M nach dem Schalten von t. Die Anzahl der Token jeder Stelle s in dieser Folgemarkierung M' ist wie folgt definiert:

$$M'(s) = \begin{cases} M(s) - W(s,t) & \text{für} \quad s \in {}^\bullet t \setminus t^\bullet \\ M(s) + W(t,s) & \text{für} \quad s \in t^\bullet \setminus {}^\bullet t \\ M(s) - W(s,t) + W(t,s) & \text{für} \quad s \in {}^\bullet t \cap t^\bullet \\ M(s) & \quad \text{sonst} \end{cases}$$

Wird nun das STS aus Abbildung 2.4 betrachtet und die Transition $T0$ geschaltet, ergibt sich die in Abbildung 2.5 gezeigte Darstellung von STS_0. Die Menge der in Abbildung 2.4 aktivierten Transitionen ist $\{T0, T3\}$.

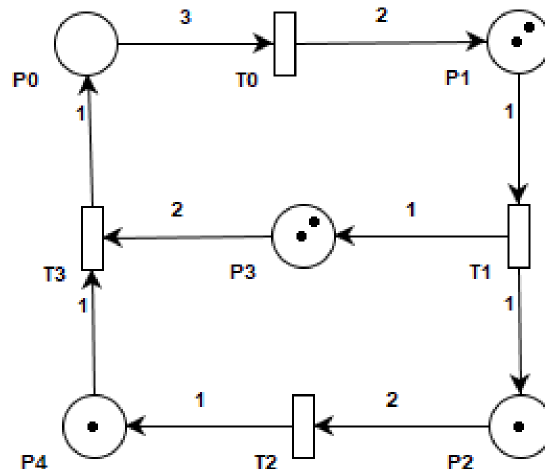


Abb. 2.5.: Stellen/Transitions-System STS_0 nach $M_0[T0\rangle M'$

Wird $T0$ nun geschaltet, befindet sich STS_0 in der in Abbildung 2.5 dargestellten Markierung. Die in M' aktivierten Transitionen sind $\{T1, T3\}$.

2.4. Stellen/Transitionssysteme mit Inhibitorkanten

Bis zu diesem Zeitpunkt wurde bei STS als Voraussetzung für die Schaltbarkeit einer Transition lediglich beschrieben, dass jede Vorstelle der Transition eine bestimmte Anzahl Token, aber mindestens einen Token, trägt. Es ist mit den bekannten Methoden und Eigenschaften nicht möglich, eine Stelle in einem Petri-Netz auf Leerheit zu prüfen.

Durch die Einführung von Inhibitorkanten wird ein Petri-Netz um die Fähigkeit, eine Stelle auf Leerheit zu testen, ergänzt (vgl. [PW08]). Die Menge der Inhibitorkanten wird durch eine Abbildung $inh : T \longrightarrow 2^S$ repräsentiert, die jeder Transition t eine Menge von Stellen zuweist, die vor dem Schalten von t auf Leerheit geprüft werden sollen. Aus der Definition folgt, dass eine Transition $t \in T$ genau dann schaltbar ist, wenn $\forall s \in {}^\bullet t : M(s) \geq W(s,t)$ und $\forall s \in inh(t) : M(s) = 0$.

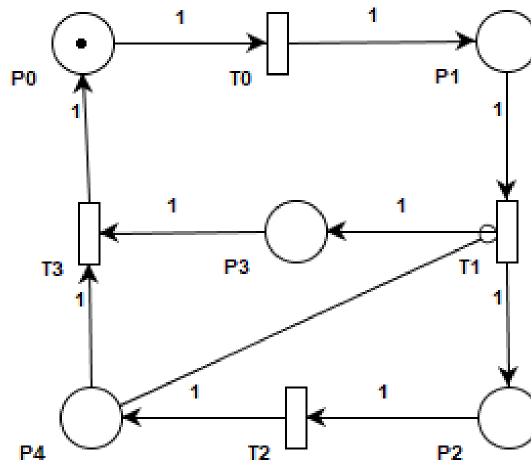


Abb. 2.6.: STS mit Inhibitorkanten

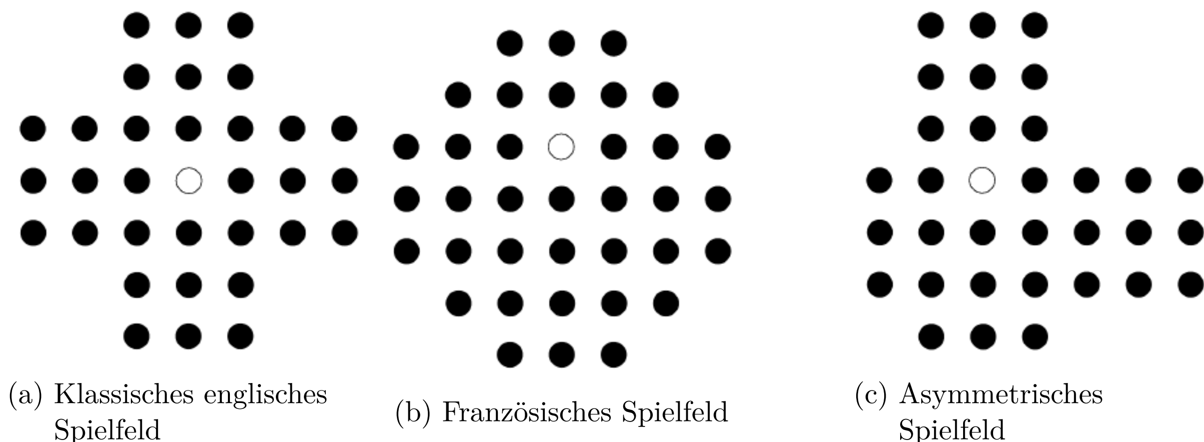
Eine Transition t ist also genau dann aktiviert, wenn alle Vorstellen mit genügend Token gefüllt und alle Stellen in $inh(t)$ leer sind. Abbildung 2.6 zeigt ein inhibitorisches Stellen/Transitionssystem. Die Transition $T1$ kann nur feuern, wenn die Stelle $P4$ keine Token trägt.

3. Modellierung mit Petri-Netzen am Beispiel des Spiels Solitär

In diesem Kapitel wird zunächst das Brettspiel Solitär beschrieben und die Regeln des Spiels eingeführt. Anschließend wird dieses Spiel mit Hilfe der Elementaren Netzsysteme modelliert.

3.1. Das Spiel Solitär

Solitär ist ein Brettspiel, dass für eine Person konzipiert wurde. In der klassischen englischen Variante wird es auf einem kreuzförmigen Spielfeld gespielt (vgl. [BCG04]). Weitere Spielfeldvarianten sind z.B. das in Abbildung 3.1b dargestellte französische Spielfeld und eine asymmetrische Variante, die in Abbildung 3.1c zu sehen ist.



(a) Klassisches englisches Spielfeld

(b) Französisches Spielfeld

(c) Asymmetrisches Spielfeld

Abb. 3.1.: Spielfeldvarianten von Solitär

In der klassischen Variante, die in Abbildung 3.1a dargestellt ist, gibt es insgesamt 33 Felder, von denen 32 initial mit einer Spielfigur besetzt sind. Das übrige Feld bleibt frei. In der für diese Arbeit genutzte Darstellung stellt der leere Kreis in einem Spielfeld eine nicht-besetzte Position eines Felds dar. Ein schwarzer Punkt ist eine Figur in einer Position im Spielfeld.

Die meisten Varianten von Solitär sind sogenannte Umkehr-Spiele, bei denen alle initial mit Figuren versehenen Felder von den Figuren befreit werden müssen. Das in diesen Fällen einzige freie Feld muss bei Umkehr-Spielen mit einer Figur versehen werden. Es gibt zusätzlich Varianten, bei denen die Position der letzten Figur irrelevant ist (vgl. [JMMT06]). In dieser Arbeit wird lediglich das englische Standard-Spielfeld berücksichtigt.

Spielablauf von Solitär

Um ein Spielfeld umzukehren, müssen Spielzüge durchgeführt werden. Diese Spielzüge werden in Solitär Sprung genannt. Bei einem Sprung wird eine Spielfigur über eine zweite Spielfigur in ein leeres Feld bewegt. Dabei wird diese zweite Spielfigur, die übersprungen wird, aus dem Spiel genommen.

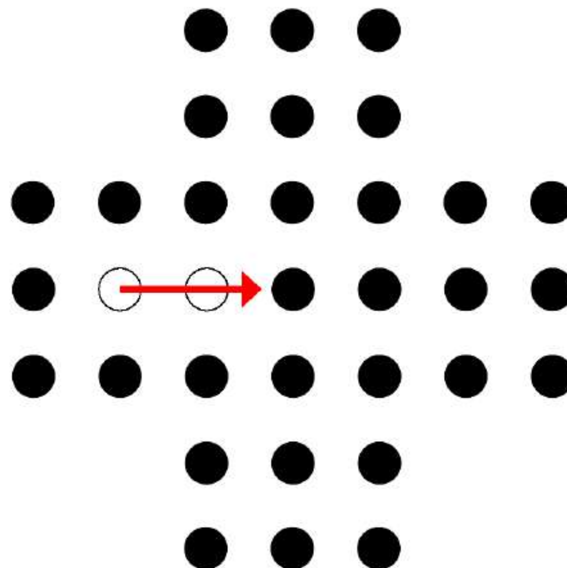


Abb. 3.2.: Solitär Spielzug

In Abbildung 3.2 ist das initial leere Feld in der Mitte des Spielfelds mit der Spielfigur aus dem Feld, aus dem der rote Pfeil stammt, besetzt. Das Feld, über das die Spielfigur *gesprungen* ist, wurde geleert.

Zur systematischen Analyse lässt sich das Spielfeld in einer Matrix darstellen. Dafür wird über das bestehende Spielfeld aus Abbildung 3.1a ein Gitter gelegt und die Zeilen und Spalten werden nummeriert.

Abb. 3.3.: Solitär Spielfeld in einem Grid

Mit der in Abbildung 3.3 gezeigten Nummerierung der Spielfelder lässt sich der Zug aus Abbildung 3.2 mit einer einfachen Übergangsform darstellen:

$$(x_1, y_3) \xrightarrow{(x_2,y_3)} (x_3, y_3)$$

Diese Form liest sich wie folgt: Die Figur auf (x_1, y_3) wird über (x_2, y_3) auf (x_3, y_3) gesetzt. Dabei wird das Feld (x_2, y_3) geleert. Der Pfeil indiziert die Richtung des Zugs. Ein Zug in Solitär darf horizontal oder vertikal vollzogen werden (vgl. [BCG04]).

Es gibt für jedes Feld pro Dimension (horizontal und vertikal) jeweils zwei Sprungvarianten. Diese sind auch mit der vorgestellten Übergangsform beschreibbar:

$$(x_2, y_3) \xrightarrow{(x_3,y_3)} (x_4, y_3) \qquad \text{Sprung nach } \textit{rechts}$$

$$(x_4, y_3) \xrightarrow{(x_3,y_3)} (x_2, y_3) \qquad \text{Sprung nach } \textit{links}$$

$$(x_3, y_2) \xrightarrow{(x_3,y_3)} (x_3, y_4) \qquad \text{Sprung nach } \textit{unten}$$

$$(x_3, y_4) \xrightarrow{(x_3,y_3)} (x_3, y_2) \qquad \text{Sprung nach } \textit{oben}$$

Die Anzahl und Auflistung aller Sprung-Möglichkeiten lässt sich aus Tabelle 3.1 entnehmen. Die in dieser Tabelle vorgestellte, von Jefferson et al. in ihrer Arbeit (vgl. [JMMT06]) genutzte Darstellung ähnelt der in diesem Kapitel vorgestellten Übergangsform von Sprüngen in Solitär.

No.	Trans.	No.	Trans.	No.	Trans.	No.	Trans.	No.	Trans.
0	2,0 → 4,0	16	2,2 → 2,4	32	2,3 → 0,3	48	1,4 → 3,4	64	6,4 → 6,2
1	2,0 → 2,2	17	2,2 → 0,2	33	2,3 → 2,1	49	1,4 → 1,2	65	6,4 → 4,4
2	3,0 → 3,2	18	3,2 → 3,0	34	2,3 → 2,5	50	2,4 → 0,4	66	2,5 → 4,5
3	4,0 → 2,0	19	3,2 → 5,2	35	2,3 → 4,3	51	2,4 → 2,2	67	2,5 → 2,3
4	4,0 → 4,2	20	3,2 → 3,4	36	3,3 → 1,3	52	2,4 → 4,4	68	3,5 → 3,3
5	2,1 → 4,1	21	3,2 → 1,2	37	3,3 → 3,1	53	2,4 → 2,6	69	4,5 → 2,5
6	2,1 → 2,3	22	4,2 → 4,0	38	3,3 → 3,5	54	3,4 → 1,4	70	4,5 → 4,3
7	3,1 → 3,3	23	4,2 → 6,2	39	3,3 → 5,3	55	3,4 → 3,2	71	2,6 → 4,6
8	4,1 → 2,1	24	4,2 → 4,4	40	4,3 → 2,3	56	3,4 → 5,4	72	2,6 → 2,4
9	4,1 → 4,3	25	4,2 → 2,2	41	4,3 → 4,1	57	3,4 → 3,6	73	3,6 → 3,4
10	0,2 → 0,4	26	5,2 → 3,2	42	4,3 → 4,5	58	4,4 → 2,4	74	4,6 → 2,6
11	0,2 → 2,2	27	5,2 → 5,4	43	4,3 → 6,3	59	4,4 → 4,2	75	4,6 → 4,4
12	1,2 → 3,2	28	6,2 → 6,4	44	5,3 → 3,3	60	4,4 → 6,4		
13	1,2 → 1,4	29	6,2 → 4,2	45	6,3 → 4,3	61	4,4 → 4,6		
14	2,2 → 2,0	30	0,3 → 2,3	46	0,4 → 0,2	62	5,4 → 3,4		
15	2,2 → 4,2	31	1,3 → 3,3	47	0,4 → 2,4	63	5,4 → 5,2		

Tab. 3.1.: Tabelle aller Transition nach [JMMT06]

Diese Tabelle beschreibt alle Sprünge im klassischen englischen Spielfeld. Die Spalten mit der Überschrift *No.* dienen der Nummerierung aller Sprünge, während die Spalten *Trans.* die Notation des Sprunges, also alle Felder, die von einem Sprung betroffen sind, bezeichnen.

3.2. Formalisierung mit Elementaren Netzsystemen

An dieser Stelle wird unmittelbar eine Verbindung und die Abbildbarkeit der Spiellogik durch Elementare Netzsysteme offensichtlich, da jedes Feld (x, y) des Spielfelds durch eine Stelle xy und jede Spielfigur durch einen Token dargestellt werden können. Da jedes Feld stets mit höchstens einer Figur belegt sein kann, eignen sich die in Abschnitt 2.2 vorgestellten Elementaren Netzsysteme zur Modellierung von Solitär, dessen Stellen mit maximal einem Token belegt sein können.

Aus Abschnitt 3.1 ist bekannt, dass für einen Sprung jeweils drei horizontal oder vertikal benachbarte Spielfelder betrachtet werden müssen. Da jedes Spielfeld durch eine Stelle und jeder Sprung mit Hilfe einer Transition dargestellt wird, muss für jeden Sprung, der in Tabelle 3.1 zu sehen ist, eine Transition erzeugt werden.

Um eine Transition zu erzeugen, wird eine Stelle mit ihren Nachbarn auf der x-Achse oder auf der y-Achse betrachtet. Daraus ergibt sich ein Tripel aus Stellen. Wir nennen ein solches Tripel K_{sol}. Das Tripel $K_{sol} = (x_0y_2, x_0y_3, x_0y_4)$ ist auf der x-Achse fest, während auf der y-Achse drei benachbarte Stellen betrachtet werden. Unabhängig, in welche Richtung ein Sprung stattfindet, muss die Stelle an zweiter Position im Tripel einen Token enthalten, da diese Position übersprungen und die Figur des Feldes aus dem Spiel genommen wird. Also muss $M(mid\{K_{sol}\}) = 1$[1] sein.

Jeder der Sprünge aus Tabelle 3.1 kann durch eine Transition dargestellt werden. Die Transition für das Beispiel $(x_2, y_3) \xrightarrow{(x_3,y_3)} (x_4, y_3)$ ist in Abbildung 3.4 dargestellt.

[1] *mid* ist eine Funktion, die den zweiten Wert des Tripels extrahiert, d.h. *mid* ist die Projektion auf die mittlere Komponente.

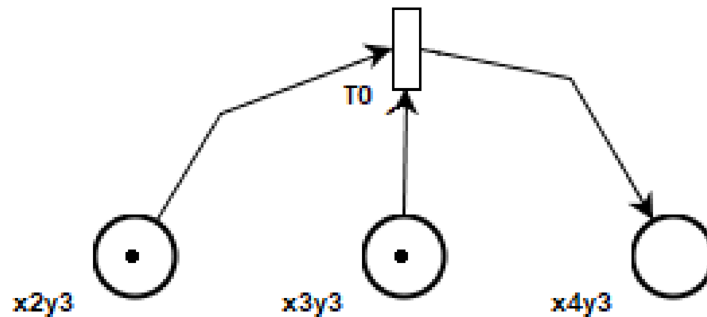


Abb. 3.4.: Transition von $K_{sol} = (x_2y_3, x_3y_3, x_4y_3)$

Die erzeugten Transitionen, ihre Benennung und auch ihre Positionierung, folgen einem bestimmten Schema. Der Name einer Transition besteht zum einen aus der Aktion *jmp__*, dem Namen der Stelle, die übersprungen wird (z.B. $x4y4$) und der Richtung, in die gesprungen wird. Die Richtung ist der erste Buchstabe des englischen Wortes für die Richtung, in die gesprungen wird (z.B. l für left, also links).

Es werden nun anhand von Tabelle 3.1 alle 76 möglichen Sprung-Transitionen erzeugt.

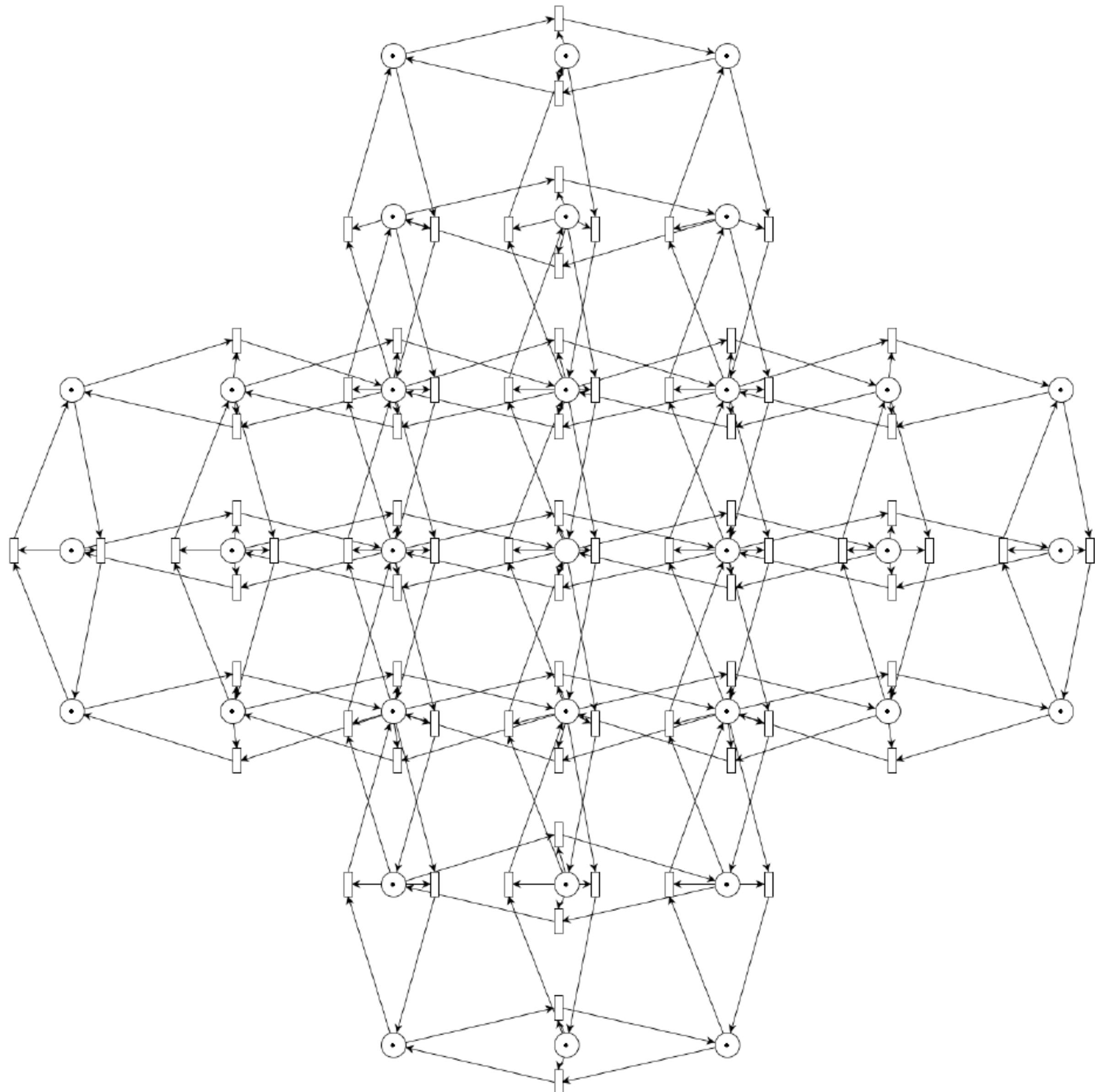


Abb. 3.5.: Solitär in initialer Markierung als Elementares Netzsystem ENS_{sol}

Das Elementare Netzsystem ENS_{sol}, das das Spiel Solitär darstellt, ist in Abbildung 3.5 zu sehen. Ein übersichtliches Spielen ist in dieser Form erschwert, da zum einen die Anzahl der Transitionen und zum anderen die Menge der Kanten zwischen Transitionen und Stellen das gesamte Netz unübersichtlich machen.

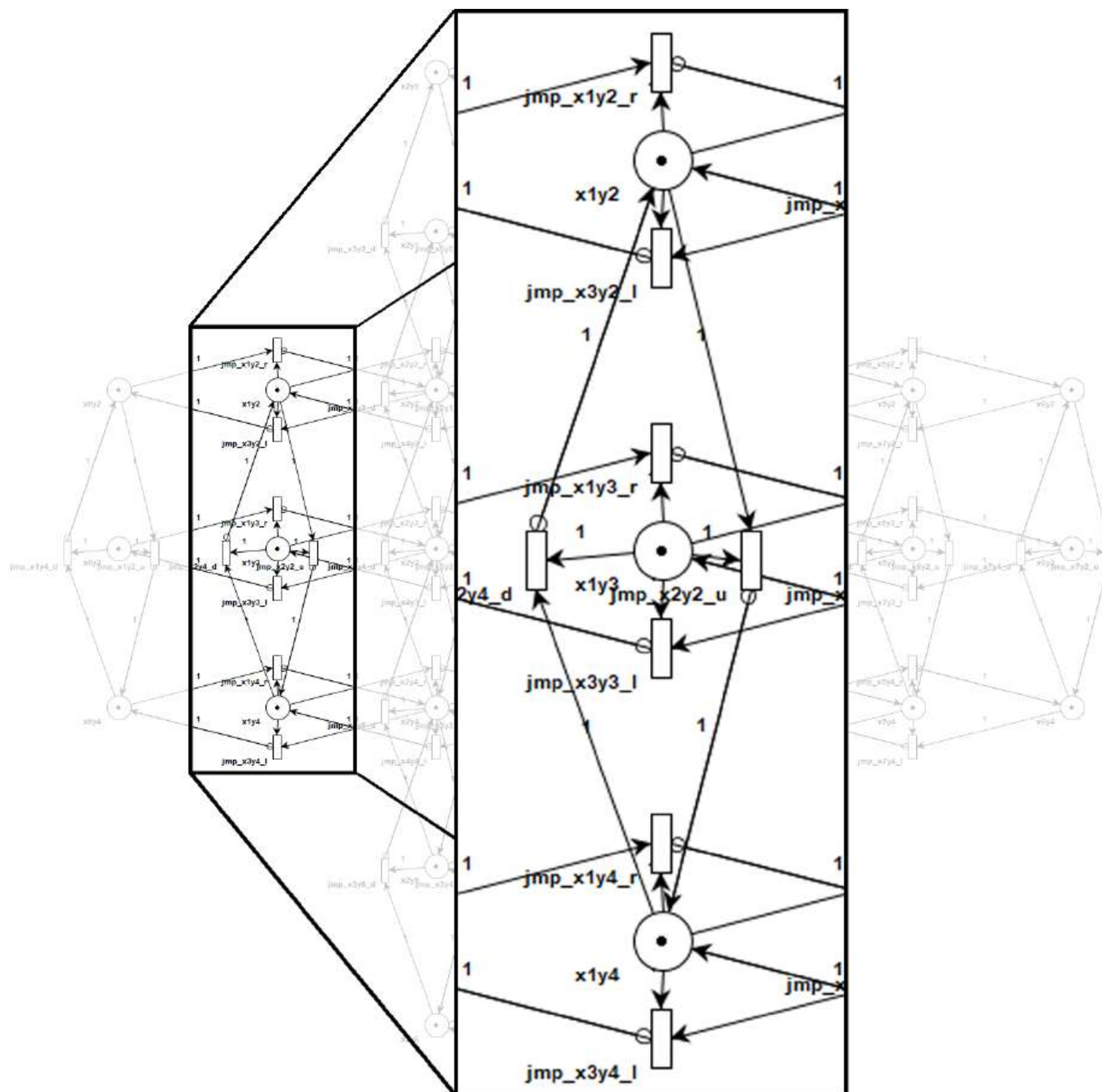


Abb. 3.6.: Detailansicht eines Übergangstripel

Insbesondere hat jede Stelle von ENS_{sol} genau $4 \leq x \leq 8$ ein- oder ausgehende Kanten. Abbildung 3.6 zeigt einen Ausschnitt von ENS_{sol} mit einem Übergangstripel, seinen ein- und ausgehenden Kanten und beiden vertikalen Sprung-Transitionen. Es zeigt sich, dass selbst in diesem Ausschnitt nicht auf Anhieb erkennbar ist, welche der gezeigten Transitionen feuerbar ist.

4. Modellierung von Solitär in PIPE

Das Programm PIPE wurde ursprünglich im Jahre 2002 als studentisches Projekt (vgl. [JDJKS09]) des Imperial College London[2] geschrieben und anschließend von unterschiedlichen weiteren Projektgruppen derselben Hochschule weitergeführt. Mit Hilfe von PIPE lassen sich über eine grafische Oberfläche inhibitorische Stellen/Transitionssysteme modellieren. Im Rahmen dieser Arbeit wurde die Version 5.0.2 genutzt.

Im Anschluss an die Vorstellung des Programms PIPE wird das in dieser Arbeit erzeugte ENS_{sol} in ein inhibitorisches S/T-System übersetzt, da PIPE die in Kapitel 3 genutzten Elementaren Netzsysteme nicht unterstützt.

4.1. Vorstellung von PIPE

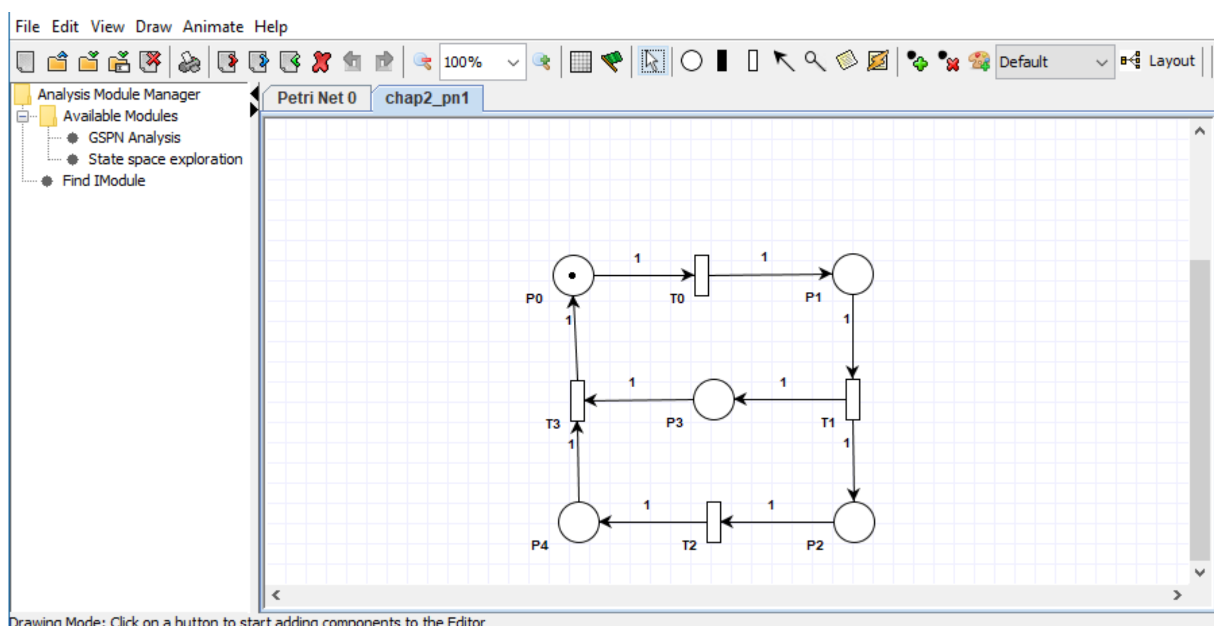


Abb. 4.1.: Hauptbildschirm von PIPE in der Version 5.0.2

Nach dem Start des Programms befindet sich der Fokus auf dem Hauptbildschirm (vgl. Abbildung 4.1). Dieses Hauptfenster wird folgend in vier Teilen erklärt: Der Menübereich

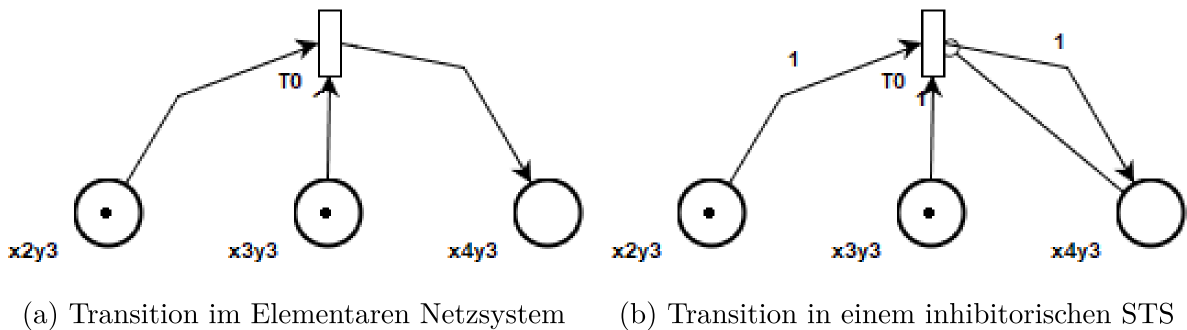
[2]http://www.imperial.ac.uk/computing

dient dem Dateimanagement und stellt eine Übersicht über alle Funktionen bereit. Über die Icon-Leiste, die sich unter dem Menübereich befindet, sind häufig benötigte Funktionen erreichbar, zum Beispiel Werkzeuge zur Modellierung von Petri-Netzen (Stellen, Transitionen, Kanten und so weiter). Unter dieser Icon-Leiste befindet sich der Arbeitsbereich, in dem die Petri-Netze erstellt und bearbeitet werden können. Auf der linken Seite des Bildschirms sind Modulen zur Analyse und Simulation aufgelistet.

Über den Menüpunkt *Animation* → *Animation mode* im Menübereich, beziehungsweise über das grüne Flaggen-Icon in der Icon-Leiste wird der sogenannte Animationsmodus gestartet. In diesem kann das sukzessive Schalten eines Petri-Netzes simuliert werden. Von der initialen Markierung ausgehend können aktivierte Transitionen geschaltet werden und so einem Petri-Netz seine Dynamik verliehen werden. Wird der Animationsmodus gestartet, verändert sich ein Teil der Icon-Leiste: Die Modellierungswerkzeuge werden ausgeblendet durch Animationswerkzeuge ersetzt. Diese bieten unter anderem die Möglichkeit, eine zufällige Transition zu feuern. Dies kann entweder pro Klick (*randomly fire a transition*) oder über eine festgelegte Anzahl von Feuerungen (*randomly fire a number of transitions*) geschehen. Neben diesem Vorgang zum Schalten von Transitionen gibt es zusätzlich die Möglichkeit, aktivierte Transitionen anzuklicken. Aktivierte Transitionen sind mit einem roten Rahmen gekennzeichnet, nicht-aktivierte Transitionen hingegen mit einem Schwarzen.

4.2. Modellierung von Solitär mit inhibitorischen S/T-Systemen

Der Aufbau des Elementaren Netzsystems für Solitär ENS_{sol} ohne Flusskanten kann direkt aus Abbildung 3.3 übernommen werden. Die Kanten von ENS_{sol} werden in PIPE durch eine Kombination aus Kanten mit einer Gewichtung von eins und einer Inhibitorkante ersetzt. Mit Hilfe dieser Kombination wird ein zu den Elementaren Netzsystemen identisches Verhalten für das inhibitorische S/T-System erzeugt. Ein Beispiel einer solchen Kombination ist Abbildung 4.2 zu entnehmen.



(a) Transition im Elementaren Netzsystem (b) Transition in einem inhibitorischen STS

Abb. 4.2.: Abbildung einer Sprung-Transitionen

Insbesondere wird an dieser Stelle die Abfrage ob der Leerheit der Nachstelle mit einer Inhibitorkante modelliert. Wie in Kapitel 3 bereits dargestellt wurde, lassen sich die Transitionen und Flusskanten systematisch aus Tabelle 3.1 erzeugen. Dieses Vorgehen erübrigt durch seine Systematik eine aufwendige Generierung der Transitionen und Flusskanten per Hand. Im Rahmen dieser Arbeit werden die Transitionen und Flusskanten algorithmisch von einem Programm erzeugt.

4.3. Algorithmische Generierung der Transitionen und Flusskanten

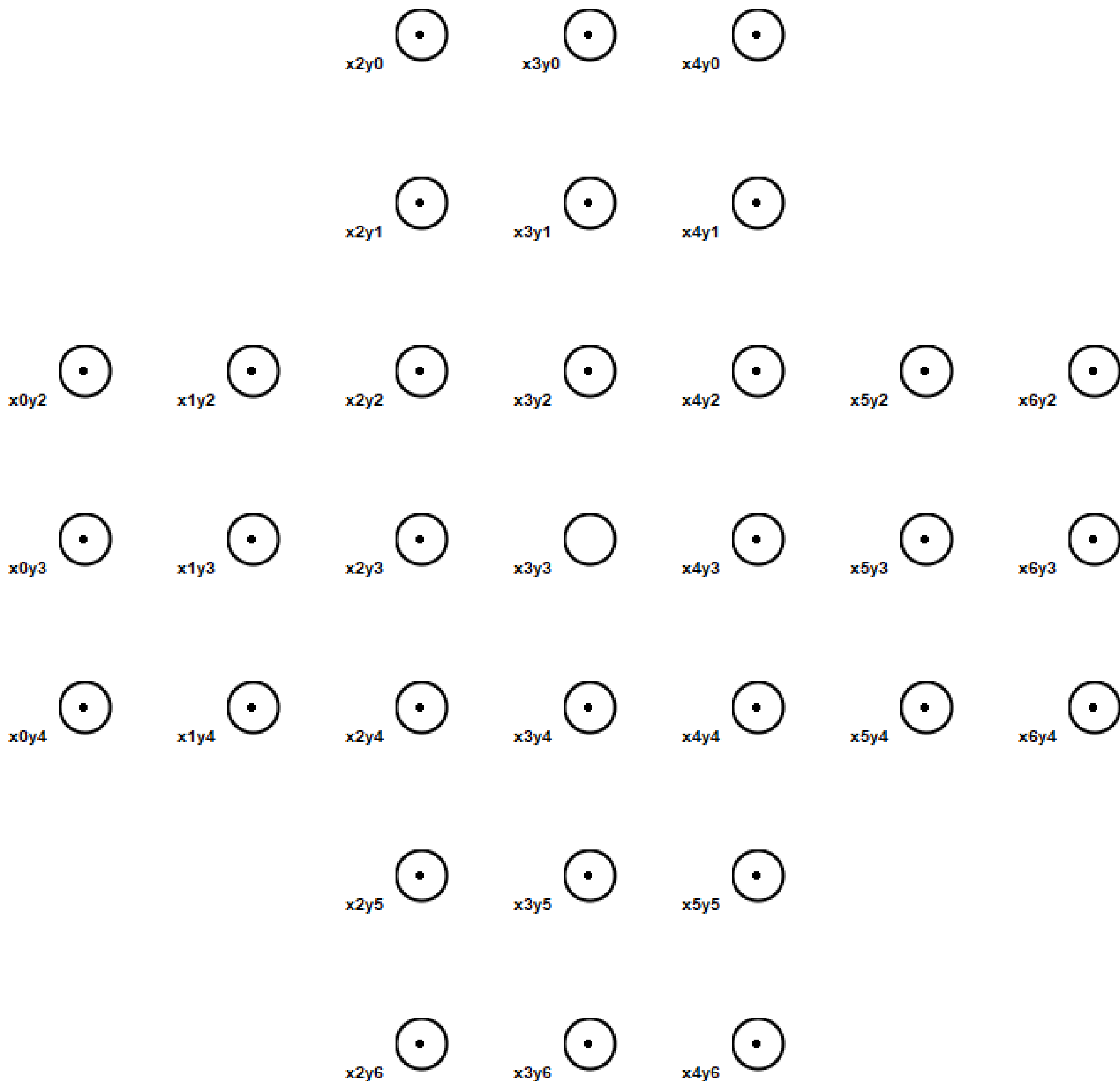


Abb. 4.3.: Solitär-Spielfeld als Petri-Netz

Zur Generierung des Petri-Netzes für Solitär wurde zunächst ein Netz in PIPE erzeugt, das nur aus den in Abbildung 4.3 gezeigten Stellen und der Startmarkierung besteht.

Anschließend wurde dieses Netz in einer XML-Datei gespeichert (vgl. [BLPJK18]). Diese XML-Datei entspricht schematisch der Petri-Net Markup Language (PNML), die in [WK03] als zentrales Dateiformat zur Beschreibung von Petri-Netzen vorgestellt wird. Die PNML-Datei wurde anschließend mit einem Texteditor analysiert. Daraus folgend wurde mit der Programmiersprache *go*[3] ein Programm erstellt, mit dem die XML-Datei um die Transitionen und die Flusskanten zwischen Stellen und Transitionen, analog zu Abbildung 3.5, erweitert wurden.

Quelltext 4.1: Ausschnitt des Generator-Programms

```
func addTransitions(net *Cpnml) error {
    xpos := []func(int) int{[...]}
    ypos := []func(int) int{[...]}

    for _, place := range net.Cnet.Cplace {
        // iterates through places
        // four possible transitions, lets go clockwise: up, right, down, left
        x, e := strconv.Atoi(string(place.Attrid[1]))
        y, e := strconv.Atoi(string(place.Attrid[3]))
        xs := []int{x + 1, x + 2, x - 1, x - 2, x, x, x, x}
        ys := []int{y, y, y, y, y + 1, y + 2, y - 1, y - 2}
        dirs := []string{"r", "r", "l", "l", "u", "u", "d", "d"}

        for i := 0; i < len(xpos); i += 2 {
            if placeExists(xs[i], ys[i], net.Cnet) && placeExists(xs[i+1], ys[i+1], net.Cnet) {
                cnt := getPlace(xs[i], ys[i], net.Cnet) // stores the central transition
                name := fmt.Sprintf("jmp_x%dy%d_%s", x+1, y, dirs[i]) // the desired name
                trans := &Ctransition{}
                [...]
                addArcs(getPlace(x, y, net.Cnet), getPlace(xs[i], ys[i], net.Cnet), getPlace(xs[
                    i+1], ys[i+1], net.Cnet), trans, net.Cnet) // adds all corresponding arcs to
                     the triple
                net.Cnet.Ctransition = append(net.Cnet.Ctransition, trans) // adds transition to
                     the petri-net
            }
        }
    }
    return nil
}
```

Quelltext 4.1 zeigt den Teil des Programms, das die Transitionen und Flusskanten für das S/T-System für Solitär erzeugt. Der Algorithmus iteriert zunächst durch alle Stellen des

[3] http://www.golang.org

Netzes, wie in Quelltext 4.1 ab Zeile 5 beschrieben. Bei den Stellen handelt es sich um alle Spielfelder des klassischen englischen Spielbretts. In der Zeile 15 wird für jede Richtung die beiden nächsten Nachbarn einer einzelnen Stelle betrachtet. Existieren beide Nachbarn für eine Stelle, entspricht diese Kombination einem Tripel, wie es in Abschnitt 3.2 vorgestellt wurde und es wird eine Transition erzeugt.

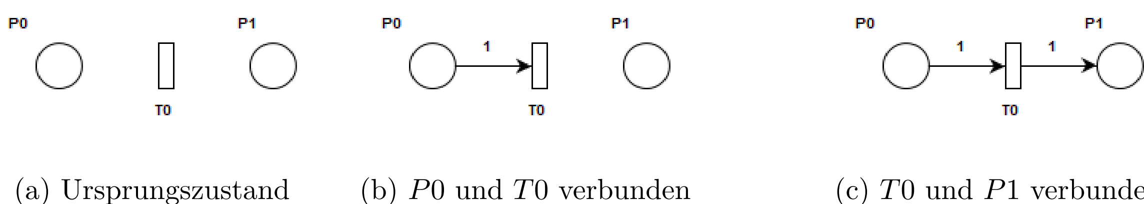
Um diese Transition mit den passenden Stellen zu verbinden, wird der Algorithmus um die Möglichkeit erweitert, auch die Flusskanten zu generieren. Zu diesem Zweck wurde die Funktion `addArcs()` erstellt, die in Quelltext 4.2 dargestellt ist. Diese Funktion ruft viermal eine andere Funktion mit dem Namen `addArc()` auf, die mit jedem Aufruf eine Kante zwischen zwei Komponenten des Petri-Netzes erzeugt.

Quelltext 4.2: Ausschnitt der addArc-Funktion

```
func addArc(source, target string, inhib bool, src, trg *Cposition) *Carc {
    arc := &Carc{
        Attrid:    fmt.Sprintf("%s TO %s", source, target),
        Attrsource: source,
        Attrtarget: target,
    }
    [...]

    if inhib {
        arc.Ctype.Attrvalue = "inhibitor"
        arc.Cinscription.Cvalue.string = ""
    } else {
        arc.Ctype.Attrvalue = "normal"
        arc.Cinscription.Cvalue.string = "Default,1"
    }

    makeArcPath := func(pos *Cposition) *Carcpath {
        return &Carcpath{Attrid: "", Attrx: pos.Attrx, Attry: pos.Attry, AttrcurvePoint: "false
            "}
    }
    arc.Carcpath = []*Carcpath{makeArcPath(src), makeArcPath(trg)}

    return arc
}
```

Die Funktion `addArc()` erhält als Argumente die Namen des Ursprungs und des Ziels der zu erzeugenden Kante, einen booleschen Wert und die Positionen (x- und y-Koordinaten) des Ursprungs und des Ziels der zu erzeugenden Kante. Der boolesche Wert ist wahr, wenn

es sich bei der zu erzeugenden Kante um eine Inhibitorkante handelt. Es wird folgend beispielhaft ein Petri-Netz mit zwei Stellen ($S = \{P0, P1\}$), einer einzelnen Transition $T0$ (also ist $T = \{T0\}$) und einer leeren Kantenmenge ($F = \emptyset$) betrachtet. Dieses nicht weiter spezifizierte Petri-Netz soll ausschließlich mit Aufrufen der `addArc()` Funktion um die Flusskanten $(P0, T0)$ und $(T0, P1)$ erweitert werden.



(a) Ursprungszustand (b) $P0$ und $T0$ verbunden (c) $T0$ und $P1$ verbunden

Abb. 4.4.: Aufbau von Verbindungen mit der addArc() Funktion

Im ersten Schritt, der in Abbildung 4.4a gezeigt ist, existiert ausschließlich die Transition $T0$ und die beiden Stellen $P0$ und $P1$. In der Anwendung des Algorithmus für das inhibitorische S/T-System für Solitär wird zunächst eine Transition erzeugt. Auf die Darstellung dieser Erzeugung wurde in diesem Beispiel bewusst verzichtet. In Abbildung 4.4b ist nun das Petri-Netz nach dem ersten Aufruf von `addArc()` zu sehen. Die Parameter, die die Funktion erhalten hat, sind die Namen des Ursprungs ($P0$) und des Ziels ($T0$), ein `false`, da es sich nicht um eine Inhibitorkante handelt und jeweils die graphischen Koordinaten der beiden Komponenten (Quelle und Ziel), um den graphischen Ursprung und das graphische Ziel der Kante zu definieren. Im Fall von Abbildung 4.4b wurde der Befehl wie folgt ausgeführt: `addArc(P0, T0, false, &Cposition{0,50}, &Cposition{50,50})`. Der nächste Schritt (wie in Abbildung 4.4b dargestellt) verläuft analog mit dem Aufruf `addArc(T0, P1, false, &Cposition{50,50}, &Cposition{100,50})`.

Wird das Programm mit Daten der XML-Datei des nicht-verbundenen Netzes für Solitär genutzt, werden die graphischen Koordinaten aus den graphischen Positionen der zu verbindenden Komponenten des Petri-Netzes berechnet.

Im Folgenden wird das in Abbildung 4.3 gezeigte Netz, das ausschließlich aus Stellen besteht, mit Hilfe dieses Algorithmus zunächst um Transitionen erweitert. Die bereits vorhandenen Stellen und die erzeugten Transitionen werden anschließend miteinander

verbunden. Dies erweitert das in Abbildung 4.3 gezeigte Netz um die Spiellogik des Spiels Solitär.

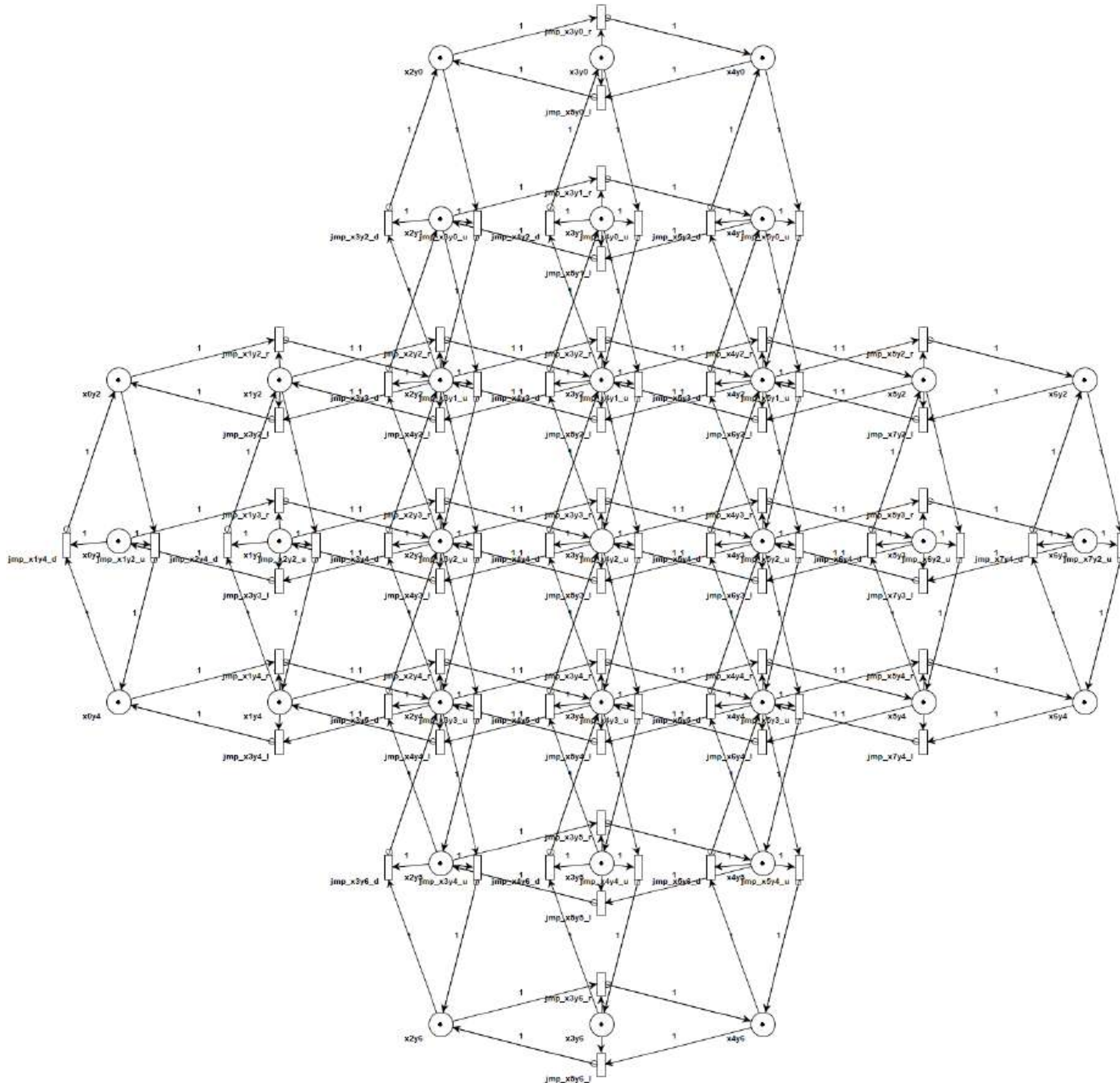


Abb. 4.5.: Solitär als inhibitorisches S/T-System

In Abbildung 4.5 ist das vollständig verbundene inhibitorische S/T-System des Spiels Solitär in seiner *spielbaren* Version in PIPE sichtbar. Wird nun der Simulations-Modus gestartet, färben sich die Rahmen aktivierter Transitionen rot. Ist eine Transition nicht aktiviert, wird sie weiterhin mit einem schwarzen Rahmen dargestellt. Gerade bei sehr großen Netzen, die eine Vielzahl an Transitionen und Flusskanten haben, ist nicht sofort

erkennbar, wann eine Transition aktiviert ist. Ebenfalls sind die Auswirkungen auf das gezeigte Petri-Netz nicht direkt ersichtlich. Dies ist insbesondere dann der Fall, wenn sich die Flusskanten zwischen Transitionen und Stellen überlappen.

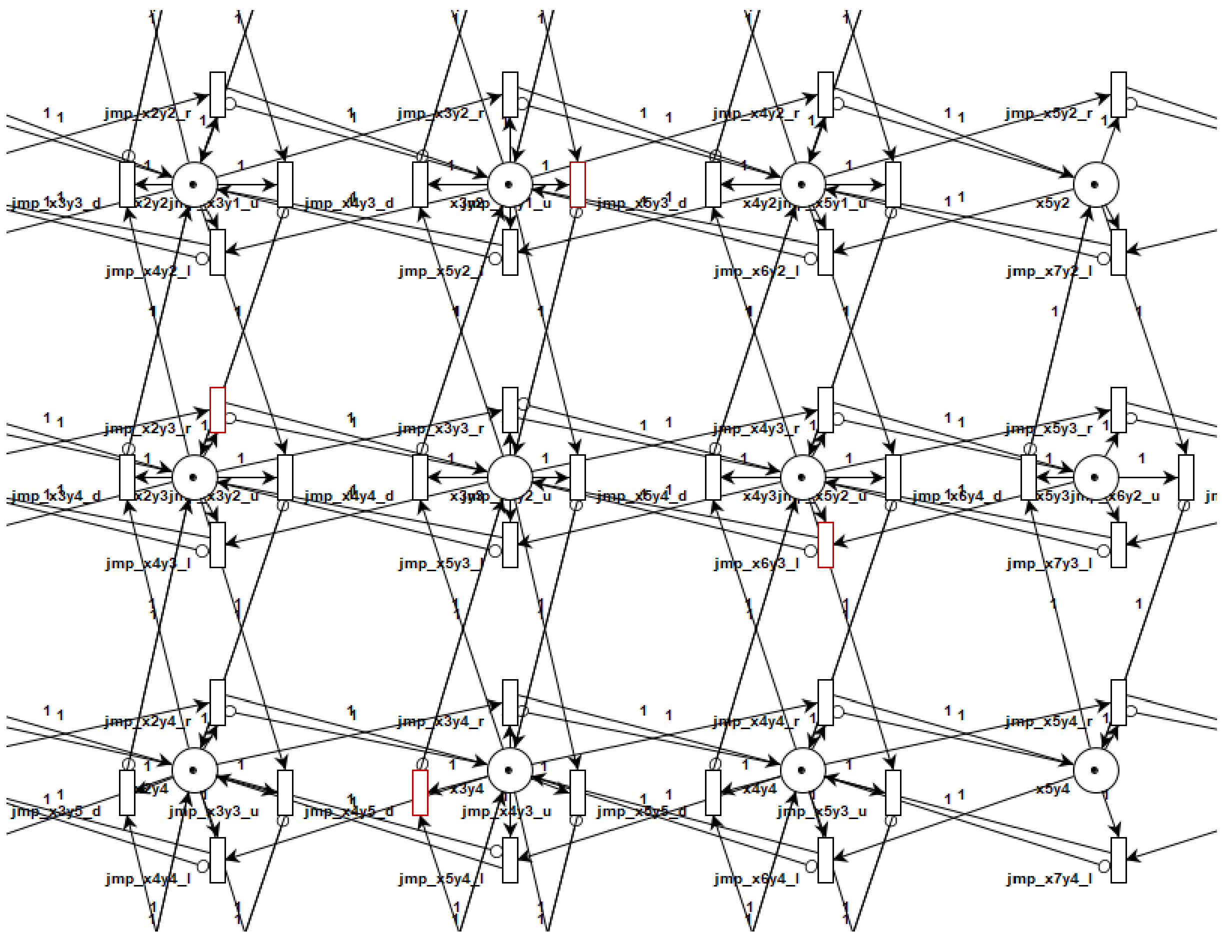


Abb. 4.6.: Aktivierte Transition in PIPE 5.0.2

Abbildung 4.6 zeigt einen Ausschnitt des in Abbildung 4.5 dargestellten inhibitorischen S/T-System des Spiels Solitär. In diesem Ausschnitt wird deutlich, dass bei komplexen Petri-Netzen, die aus einer Vielzahl an Transitionen und Flusskanten bestehen, die möglichen Auswirkungen des Schaltens einer aktivierten Transitionen nicht direkt ersichtlich sind. Dies kann zum einen das Verständnis für den Ablauf eines bestimmten Petri-Netzes, als auch die Spielbarkeit eines mit Petri-Netzen modellierten Spiels beeinträchtigen. Um diese Beeinträchtigungen zu eliminieren, wird PIPE um die Möglichkeit erweitert, nicht-aktivierte Transitionen auszublenden.

5. Erweiterung von PIPE um das Ausblenden nicht-aktivierter Transitionen

Um die Probleme mit der Übersichtlichkeit des in Abbildung 3.5 gezeigten Petri-Netz für Solitär zu lösen, muss das Programm PIPE erweitert werden. Zunächst wird der Inhalt des Quelltexts des PIPE-Projekts sowie die bestehende Architektur und Struktur in einer Analyse vorgestellt. Anschließend wird unter Berücksichtigung der analysierten Architektur ein Konzept erstellt, mit dem die Erweiterung des Programms um das Ausblenden nicht-aktivierter Transitionen umgesetzt werden. Während der Analyse und Konzeptionsphase wird primär darauf geachtet, dass die Anzahl der zu modifizierenden Quelltextzeilen auf ein Minimum reduziert werden, um die Abwärtskompatibilität zu Petri-Netzen älterer PIPE Versionen zu gewährleisten.

5.1. Werkzeuge

Zur Analyse und Implementierung der Änderungen an der Software wurde die Integrierte Entwicklungsumgebung IntelliJ Idea (Version 2018.1.2) in der kostenlosen *Community Version* genutzt. Die Klassendiagramme, die im Verlauf dieses Kapitels gezeigt werden, wurden mit Hilfe des Intellij Idea-Plugins *Code Iris*[4] generiert. Die Erstellung der Grafiken fand mit dem GNU Image Manipulation Program, kurz GIMP[5] statt. Die gesamte Entwicklung fand in der Quelloffenen Java Entwicklungs- und Laufzeitumgebung OpenJDK[6] in Version 1.7.0 statt.

5.2. Analyse des Quelltextes und der Projektstruktur

Das PIPE-Projekt ist in zwei Teilprojekte unterteilt. Zum einen sind in *PIPECore* die Klassen und die Logik des Programms zu finden. Zum anderen sind im Teilprojekt *PIPE*

[4]https://plugins.jetbrains.com/plugin/7324-code-iris
[5]https://www.gimp.org/
[6]http://openjdk.java.net/

die GUI und diverse Zusatzmodule (zum Beispiel Module zur automatischen Klassifizierung von Petri-Netzen) verankert. Um eine Übersicht über die genauere Architektur dieser Teilprojekte zu erhalten, wurde für jedes dieser Teilprojekte ein UML-Klassendiagramm erzeugt.[7]

5.2.1. PIPECore

Abb. 5.1.: Klassendiagramm des PIPECore Teilprojekts

In Abbildung 5.1 ist das UML-Diagramm des Teilprojekts PIPECore zu sehen. Der im Rahmen dieser Arbeit betrachtete Teil der Anwendung ist in Abbildung 5.1 im oberen

[7]Die vollständigen Klassendiagramme sind auf der beigelegten CD zu finden.

linken Teil und in Abbildung 5.2 im Detail sichtbar. Dieser Teil befindet sich im Java-Namespace `uk.ac.imperial.pipe.models.petrinet`. In diesem Namespace befinden sich die Klassen für alle Petri-Netz relevanten Elemente: Stellen, Transitionen, Token und Kanten. Jedes dieser Elemente hat eine abstrakte Definition und mindestens eine spezifische Implementierung.

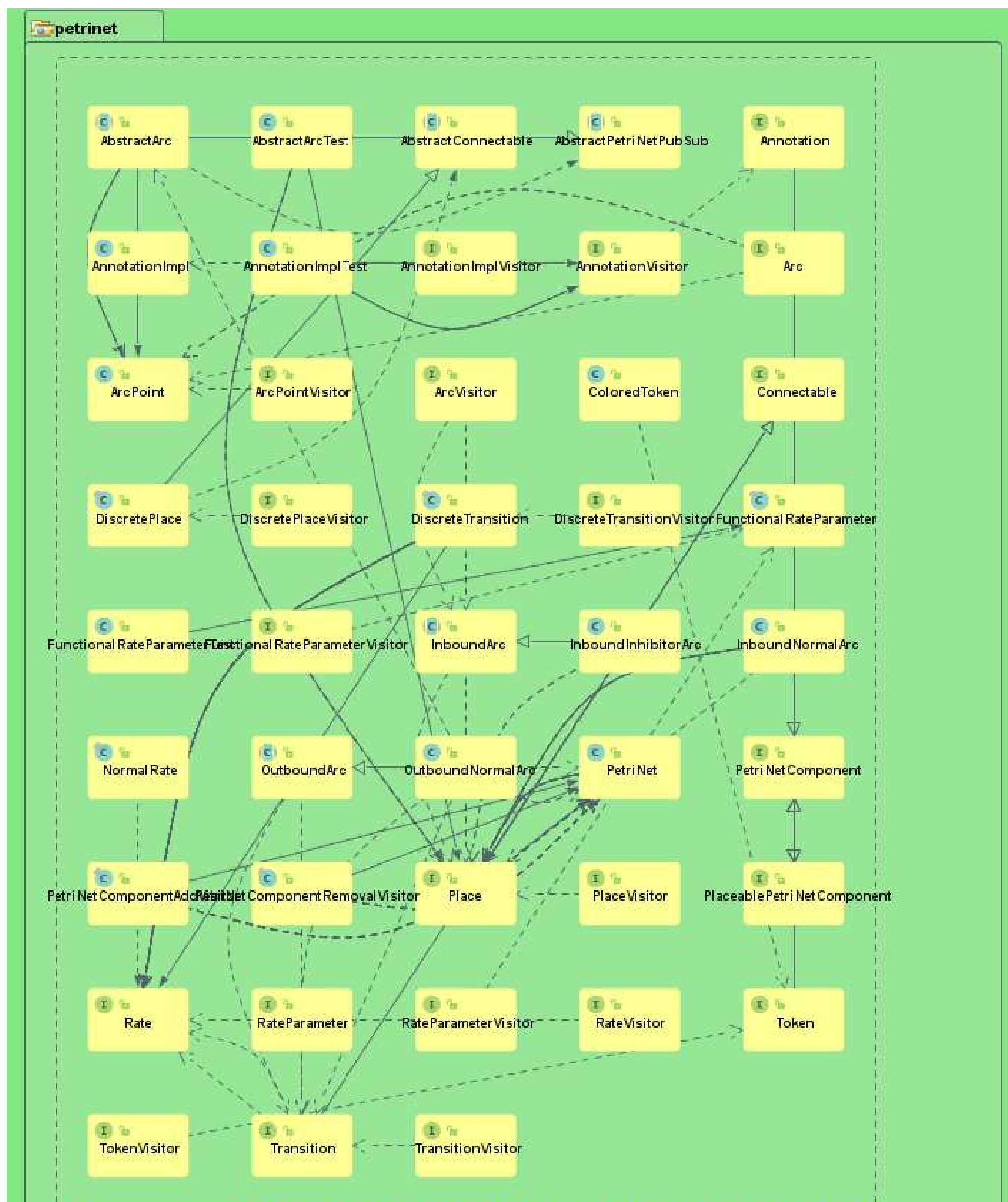


Abb. 5.2.: Detailansicht des PIPECore Teilprojekts

Die Implementierung der Stellen findet über das Interface `Place` statt und wird logisch von der `DiscretePlace` Klasse abgebildet. Transitionen werden über das Interface `Transition` definiert, die in `DiscreteTransition` implementiert wird. Die Kanten sind im Interface `Arc` beschrieben und werden in der Klasse `AbstractArc` implementiert. Die Stellen und Transitionen haben jeweils nur eine einzelne Implementierung. Die Klasse der Kanten (`AbstractArc`) ist in diesem Fall eine Besonderheit, da sie mehr als eine Implementierung im Projekt hat: `InboundArc` und `OutboundArc`. Diese entsprechen den eingehenden und den ausgehenden Kanten der Elemente eines Petri-Netzes. Unter Berücksichtigung der in Abschnitt 2.4 vorgestellten inhibitorischen Petri-Netze wird nun offensichtlich, dass diese Klassen noch weiter spezifiziert werden müssen: In normale Kanten (`InboundNormalArc` und `OutboundNormalArc`) und Inhibitorkanten (`InboundInhibitorArc`).

Um die gewünschten Erweiterungen am Programm zu vollziehen, müssen alle Transitionen, die zu einem Zeitpunkt t nicht aktiviert sind, ausgeblendet werden. Zusätzlich müssen alle Kanten, die mit diesen Transitionen verbunden sind (sowohl eingehend, als auch ausgehend) ausgeblendet werden. Gesucht wird demnach eine Verbindung zwischen den Transitionen und den Kanten innerhalb von PIPECore. Im Folgenden werden die Deklarationen der Interfaces `Transition` und `Arc` sowie die implementierenden Klassen dieser Interfaces miteinander verglichen und ein gemeinsames Interface gesucht.

Quelltext 5.1: Erweiterungen des PetriNetComponent Interface

```java
1 public interface Transition extends Connectable { [...] }
2 public interface Connectable extends PlaceablePetriNetComponent { [...] }
3 public interface PlaceablePetriNetComponent extends PetriNetComponent { [...] }
4
5 public interface Arc<S extends Connectable, T extends Connectable> extends PetriNetComponent {
      [...] }
```

Aus Quelltext 5.1 lassen sich die Implementierungsketten

`Transition` $\rightarrow$ `Connectable` $\rightarrow$ `PlaceablePetriNetComponent` $\rightarrow$ `PetriNetComponent`

und

`Arc` $\rightarrow$ `PetriNetComponent`

erzeugen. Durch diese Implementierungsketten wird deutlich, dass das gemeinsame Interface der Transitionen und Kanten mit dem Interface `PetriNetComponent` gefunden wurde.

In diesem Interface müssen die Definitionen, die zum Ausblenden dieser Elemente nötig sind, hinterlegt werden. Die `PetriNetComponent` Klasse wurde daher um drei Methoden erweitert: Jeweils eine Anweisung zum Ausblenden der Komponente (`Hide()`), eine zum Anzeigen der Komponente (`Show()`) und eine Abfrage, ob die Komponente gezeichnet werden soll.

Quelltext 5.2: Erweiterungen des PetriNetComponent Interface

```
void Hide(); // weist ein objekt an, nicht mehr gezeichnet zu werden

void Show(); // weist ein objekt an, gezeichnet zu werden

boolean IsHidden(); // gibt einen boolschen wert zurück, der besagt, dass das element nicht
    gezeichnet werden soll
```

Mit den in Quelltext 5.2 beschriebenen Erweiterungen sind die nötigen Vorkehrungen getroffen, um Komponenten ausblenden zu können. Diese Vorkehrungen werden in den Implementierungen des Interfaces um ihre Programmlogik ergänzt. An dieser Stelle ist zu beachten, dass Dummy-Implementierungen dieser Methoden in jeder Klasse, die von `PetriNetComponent` erbt, hinzefügt wurden. Diese Dummy-Implementierungen sind in Quelltext 5.3 gezeigt.

Quelltext 5.3: Dummy-Implementierung

```
@Override
public void Hide() {
    throw new UnsupportedOperationException();
}

@Override
public void Show() {
    throw new UnsupportedOperationException();
}

@Override
public boolean IsHidden() {
    throw new UnsupportedOperationException();
}
```

Es wird zunächst bei allen Elementen eine Ausnahme ausgelöst, mit der einer Aufrufenden Stelle mitgeteilt wird, dass die Methode nicht unterstützt wird. Es werden nun alle Komponenten, die dieses Interface implementieren, differenziert betrachtet: Zum einen gibt es implementierende Klassen, die von nicht-aktivierte Transitionen betroffen sind (betroffen heißt in diesem Zusammenhang, dass sie ausgeblendet werden müssen.). Zum anderen gibt es Implementierungen, denen keine weitere Logik hinzugefügt werden, da sie vom Ausblenden nicht-aktivierter Transitionen nicht betroffen sind.

Unter allen Implementierungen des `PetriNetComponent`-Interfaces gibt es also implementierende Klassen, die die Logik zum Ausblenden beinhalten müssen und solche, die dieser Logik nicht folgen. An dieser Stelle folgt nun eine Liste der Klassen, die auch nach Fertigstellung aller Erweiterungen die Dummy-Implementierung beinhalten:

- `ColoredToken` → `Token` → `PetriNetComponent`
- `RateParameter` → `PetriNetComponent`
- `PlaceablePetriNetComponent` → `PetriNetComponent`

Für alle anderen Klassen, die das Interface `PetriNetComponent` implementieren, gibt es eine konkrete Implementierung der Logik. Diese ist für jede spezifische Klasse, die diese Logik tatsächlich verwendet (Transitionen, Kanten), identisch:

Quelltext 5.4: Implementierung der Methoden

```
1  boolean _hide;
2  @Override
3  public void Hide() {
4
5      _hide = true;
6  }
7
8  @Override
9  public void Show() {
10
11     _hide = false;
12 }
13 @Override
14 public boolean IsHidden() {
15
16     return _hide;
17 }
```

Es wird eine boolesche Variable `_hide` eingeführt, die den Sichtbarkeits-Status des Elements speichert. Diese Variable kann über die Methoden `Hide()` und `Show()` modifiziert und über die Methode `IsHidden()` abgefragt werden.

5.2.2. PIPE-GUI

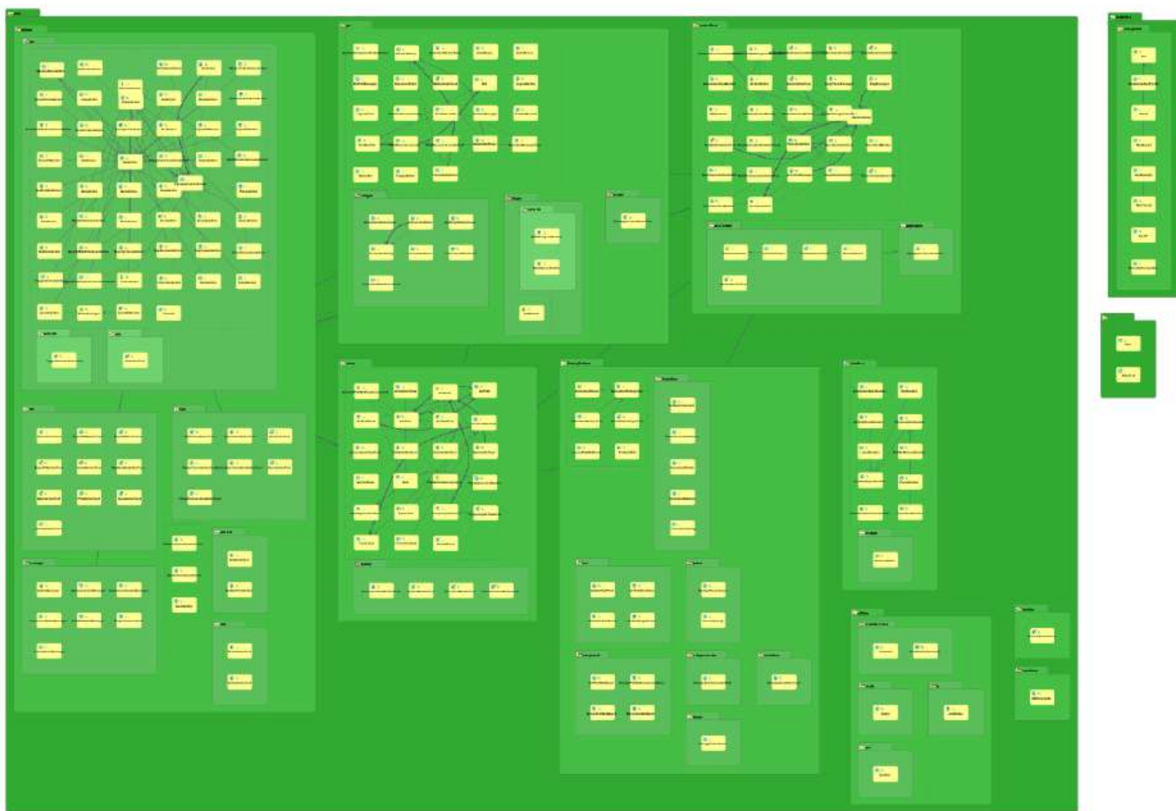


Abb. 5.3.: Klassendiagramm des PIPE-GUI Teilprojekts

Im PIPE-GUI-Projekt werden die Elemente der gezeichneten Petri-Netzen in der grafischen Benutzerschnittstelle (GUI) von PIPE gesammelt. Zunächst wird das Ausblenden der Elemente innerhalb des angezeigten Petri-Netzes implementiert. Anschließend wird der GUI ein Bedienelement hinzugefügt, das den Wechsel zwischen dem Ein- und Ausblenden nicht-aktivierter Transitionen ermöglicht. Das zugehörige Bedienelement wird der Icon-Leiste des Programms im Animationsmodus in Form eines weiteren Icons, das im Rahmen dieser Arbeit erstellt wurde, hinzugefügt.

5.2.3. Modifikation der angezeigten Petri-Netz Elemente

Nachdem die logischen Vorkehrungen für das Ausblenden von Komponenten in Petri-Netzen getroffen wurden, müssen nun die Klassen, die die Transitionen und Kanten im Programm PIPE zeichnen, modifiziert werden.

Diesem Schritt vorausgehend muss dem Programm mitgeteilt werden, ob nicht-aktivierte Transitionen gezeichnet oder ausgeblendet werden sollen. Dies betrifft nur den Animationsmodus der Anwendung, der sich in der Klasse `GUIAnimator` befindet. Diese wurde um die Eigenschaft `hideDisabledTransitions` erweitert, die programmweit festlegt, ob nicht-aktivierte Elemente gezeichnet werden sollen.

Quelltext 5.5: Änderungen an der GuiAnimator-Klasse

```
1    public void ToggleHideDisabledTransitions() {
2        this.hideDisabledTransitions = !this.hideDisabledTransitions; // oldest trick in the
             books to toggle boolean variables
3        startAnimation(); // resets the drawing
4    }
5
6    public boolean shouldHideDisabledTransitions() {
7        return this.hideDisabledTransitions;
8    }
```

Die Klasse `GUIAnimator` wird nach dem Start des PIPE-Programms einmalig instanziiert und ist in Folge über die gesamte Anwendung erreichbar. Demnach ist die öffentliche Methode `shouldHideDisabledTransitions` auch für jede Klasse der Petri-Netz-Elemente erreichbar. Durch Quelltext 5.1 ist bereits das Interface `Connectable` bekannt. Ein ähnliches Interface findet sich auch auf der Zeichnungsebene der einzelnen Elemente. Das Interface nennt sich `ConnectableView`, mit dem direkt die Nomenklatur der Zeichnungsebene eingefügt wird.

Die einzelnen Elemente (Transitionen, Stellen und Kanten) werden durch sogenannte Views dargestellt. Da nur Transitionen und Kanten ausgeblendet werden, müssen die diesen Elementen entsprechenden Views angepasst werden. Dabei handelt es sich bei den Transitionen um die Klasse `TransitionView` und bei den Kanten um die Klasse `ArcView`.

TransitionView

Die Klasse `TransitionView` ist die darstellende Klasse aller Transitionen eines Petri-Netzes.

Quelltext 5.6: paintComponent-Funktion der Klasse TransitionView

```
1  @Override
2  public void paintComponent(Graphics g) {
3      if (petriNetController.isInAnimationMode() && petriNetController.getAnimator().
           shouldHideDisabledTransitions() && !model.isEnabled()) {
4          this.textLabel.setVisible(false);
5          return;
6      }
7  ...
8  }
```

Die Methode `paintComponent` wurde um eine Abfrage erweitert, die das Zeichnen unterbindet (`return;` in Zeile 6), falls die Abfrage, die in Zeile 4 zu sehen ist, mit wahr ausgewertet wird. In dieser Abfrage wird überprüft, ob sich die Anwendung im Animationsmodus befindet (`petriNetController.isInAnimationMode()`), die nicht-aktivierten Transitionen ausgeblendet werden sollen (analog zur Einführung aus dem vorigen Kapitel) und die aktuell betrachtete Transition nicht aktiviert ist (`!model.isEnabled()`). In dieser Abfrage stellt `!` die Negierung einer Aussage dar. Das Objekt `model` ist die logische Repräsentation einer Komponente (In der `TransitionView` ist dies zum Beispiel ein Objekt der Klasse `DiscreteTransition`).

ArcView

Es gibt analog zu den Erkenntnissen aus dem vorigen Kapitel zwei Klassen für die Kanten: `NormalArcView` und `InhibitorArcView`. Diese obliegen zwar den gleichen Anpassungen, müssen dennoch unabhängig voneinander betrachtet werden.

Quelltext 5.7: paintComponent Funktion der Klasse ArcView

```
@Override
    public void paintComponent(Graphics g) {
        if (petriNetController.isInAnimationMode() && petriNetController.getAnimator().
            shouldHideDisabledTransitions() && model.IsHidden()) {
            for (TextLabel lbl : this.weightLabel) {
                lbl.setVisible(false);
            }
            return;
        }
    ...
    }
```

Die Abfrage aus Quelltext 5.7 ist identisch zu der, die der Klasse `TransitionView` in Quelltext 5.6 hinzugefügt wurde. Zusätzlich werden an dieser Stelle die `TextLabel` der Kanten über die Methode `setVisible(false);` ausgeblendet. Diese Label sind zum Beispiel die Anzeige der Gewichtung der Kanten eines Petri-Netzes, die bei entsprechender Einstellung ebenfalls ausgeblendet werden sollen.

An dieser Stelle sind die Voraussetzungen zum Ausblenden nicht-aktivierter Transitionen und der dazugehörigen Kanten erfüllt. In der Vorstellung von PIPE aus Kapitel 3 ist bereits bekannt, dass aktivierte Transitionen durch einen roten Rahmen gekennzeichnet werden. Dies impliziert, dass bereits eine Möglichkeit zur Identifikation von aktivierten Transitionen innerhalb des Programmcodes existiert. Diese Methode befindet sich ebenfalls in der `GuiAnimator` Klasse. Sie wurde um die Möglichkeit erweitert, dass nach einem jeden Schritt der Simulation, also nach dem Feuern einer Transition, das Petri-Netz neu gezeichnet wird. So wird pro Simulationsschritt jede Transition betrachtet, die mit Hilfe der Methode `Hide()` ausgeblendet oder per `Show()` eingeblendet wird. Anschließend werden alle ein- und ausgehenden Kanten dieser Transitionen aufgezählt und ebenfalls mit der Methode `Hide()` aus- oder mit der Methode `Show()` eingeblendet.

Mit diesen Änderungen kann nun gesteuert werden, ob in der Simulation in PIPE nicht-aktivierte Transitionen ein- oder ausgeblendet werden sollen.

5.2.4. Erweiterung der Icon-Leiste um einen Button

Die Icon-Leiste wird über eine Instanz der Klasse `PipeApplicationBuilder` gesteuert, der zum Start von PIPE eine Menge an Unterklassen hinzugefügt wird. Diese stellen die einzelnen Icon-Gruppen dar (zum Beispiel alle Icons des Animationsmodus oder alle Icons zum Zeichnen eines Petri-Netzes). Die Klasse, die die Icons für den Animationsmodus bereitstellt heißt `AnimateActionManager`. Die Methode `getAnimationToolbar()`, die während des Programmstarts aufgerufen wird, benutzt eine Methode der Klasse `AnimateActionManager` , die die Liste aller Icons und ihre Funktionen, die im Animationsmodus aufrufbar sein sollen, bereitstellt.

Abb. 5.4.: Klassendiagramm AnimateActionManager

Abbildung 5.4 zeigt einen Ausschnitt der Klassendefinition der `AnimateActionManager`-Klasse. In diesem Diagramm sind die Funktionen des `AnimateActionManager` dargestellt. Es lässt sich jeder Variable eine Aktion zuordnen: Die Variable `toggleAnimationAction` ist der Hauptschalter für den Animationsmodus, der als Wechselschalter den Animationsmodus startet beziehungsweise stoppt. `stepbackwardAction` und `stepforwardAction` steuern den historischen Verlauf einer Animationsfolge.

`randomAction` dient der einfachen, zufallsbasierten Feuerung einer aktivierten Transition. `multipleRandomAction` führt die Aktion von `randomAction` endlich mal aus, die genaue Anzahl der Feuerungen werden vom Benutzer über einen Dialog festgelegt.

Zur Steuerung der in den vorigen Kapiteln erstellten und vorgestellten Modifikation wurde der `AnimateActionManager` um die Variable `toggleHideDisabledTransitionsAction` erweitert.

Quelltext 5.8: Quelltextauszug der Klasse AnimateActionManager

```
public final class AnimateActionManager implements ActionManager {
    public AnimateActionManager(PipeApplicationModel applicationModel,
        PipeApplicationController applicationController) {
        [...]
        toggleHideDisabledTransitionsAction = new ToggleHideDisabledTransitionsAction("Toggle",
            "toggles the visibility of non-enabled transitions", "Ctrl T",
            applicationController);
    }
}
```

Die dazugehörige Klasse `ToggleHideDisabledTransitionsAction` steuert im Animationsmodus die Eigenschaft, die nicht-aktivierten Transitionen ein- oder auszublenden. Der Name der Grafik, die in der Leiste als Icon für eine Funktion angezeigt wird, ergibt sich aus dem ersten Parameter des Konstruktors. In der Klasse aus Quelltext 5.8 ist dies der Name *Toggle*. Das Icon wird im PNG-Format im Ressourcen-Verzeichnis des Java Projekts für PIPE hinterlegt.

Quelltext 5.9: Quelltextauszug der Klasse ToggleHideDisabledTransitionsAction

```
public class ToggleHideDisabledTransitionsAction extends AnimateAction {
    [...]
    public void actionPerformed(ActionEvent e) {
        applicationController.getActivePetriNetController().getAnimator().
            ToggleHideDisabledTransitions();
    }
    [...]
}
```

Quelltext 5.9 zeigt den Teil des Quelltextes für den Button, mit dem das ein- und ausblenden nicht-aktivierter Transitionen gesteuert wird. Die Methode `actionPerformed (ActionEvent e)` wird automatisch mit Betätigung des Buttons aufgerufen. Über den `PipeApplicationController` wird auf die aktuelle Instanz der `GuiAnimator`-Klasse zugegriffen und dessen Methode `ToggleHideDisabledTransitions()` aufgerufen, die den Anzeigemodus für nicht-aktivierte Transitionen im angezeigten Petri-Netz umschaltet. Die erweiterte Icon-Leiste im Animationsmodus ist in Abbildung 5.5 sichtbar.

100%

Abb. 5.5.: Erweiterte Icon-Leiste in PIPE

6. Validierung der Ergebnisse

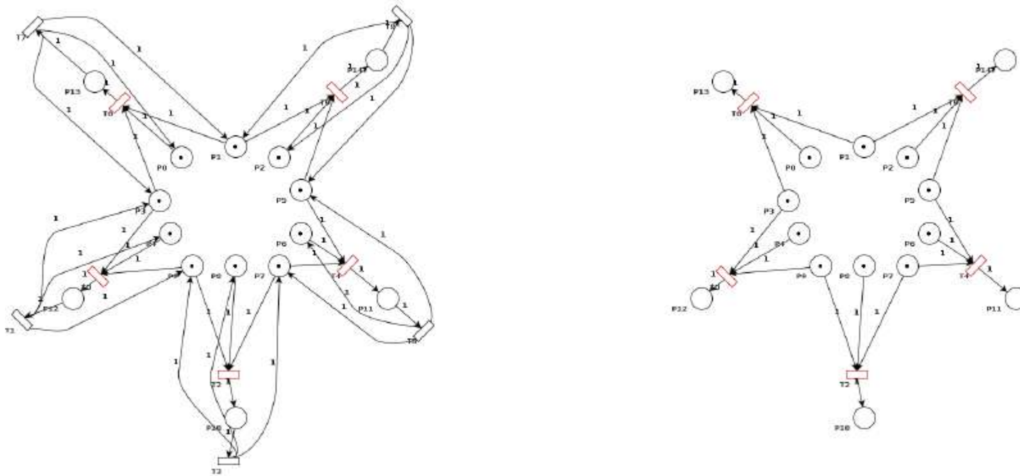
Zur Validierung der Änderungen am Programm wird das Verhalten von PIPE betrachtet. Anschließend werden die Beobachtungen des Petri-Netzes für Solitär im Programm PIPE mit den Regeln von Solitär verglichen. Neben einzelnen Schritten wird auch eine kürzeste Gewinnstrategie für Solitär, wie sie in [BCG04] zu finden ist, überprüft. Diese Lösung bedient sich einer abweichenden Notation, die Lösungsschritte sind jedoch nachvollziehbar und auf die in Kapitel 3 eingeführte Notation abbildbar. Zunächst wird die Software jedoch an einem einfachen Beispielnetz, wie es aus Abschnitt 2.2 bekannt ist, validiert.

6.1. Einfache Tests

Mit diesen Tests werden die generellen Funktionen des Programms überprüft. Sie dienen der Bereitstellung einer fehlerfreien Basis für die komplexeren Testfälle, die in den noch folgenden Kapiteln durchgeführt werden.

6.1.1. Laden eines Petri-Netzes

In diesem Test soll ein Petri-Netz geladen werden, das mit einer alten Version von PIPE erzeugt wurde. Als Akzeptanzkriterium wird das erfolgreiche Laden und Anzeigen des Petri-Netzes definiert. Das Programm PIPE stellt nach seiner Installation eine Auswahl an Beispielnetzen bereit. Dass diese Netze nicht im Rahmen dieser Arbeit erzeugt wurden und daher auf einer älteren Version des Programms PIPE basieren, ist obligatorisch und demnach ein optimales Beispiel für das Laden eines Petri-Netzes, das mit einer alten Version von PIPE erzeugt wurde. Für den Test wurde eines dieser Beispielnetze geladen.



(a) STS_{dine} mit eingeblendeten nicht-aktivierten Transitionen

(b) STS_{dine} mit ausgeblendeten nicht-aktivierten Transitionen

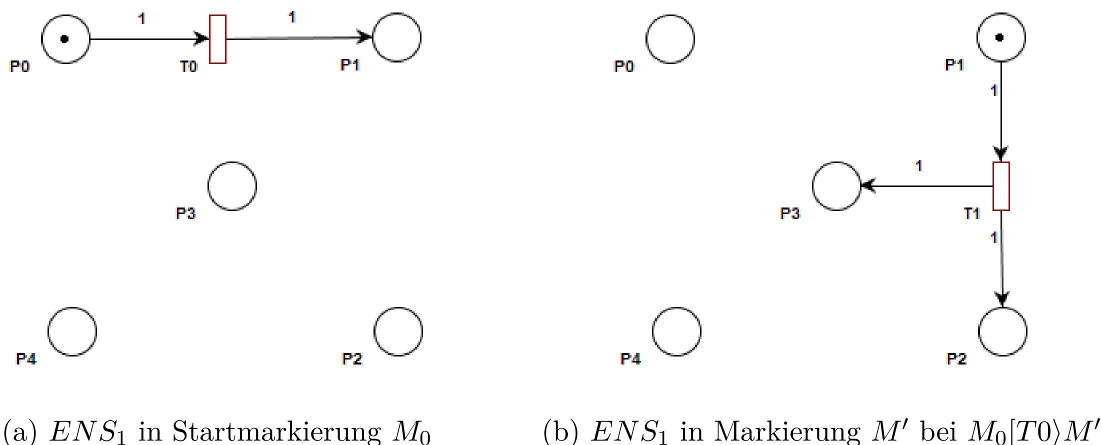
Abb. 6.1.: Beispielnetz *Dining philosophers.xml* STS_{dine}

6.1.2. Ausblenden nicht-aktivierter Transitionen

Dieses Beispiel bedient sich am Petri-Netz ENS_1 aus Abbildung 2.3. Zur besseren Übersicht findet sich das ENS_1 ein zweites Mal in Abbildung 6.2. Es wird für jede Transition $t \in T$ für ENS_1 der Vor- und den Nachbereich betrachtet:

- ${}^{\bullet}T0 = \{P0\}$
- $T0^{\bullet} = \{P1\}$
- ${}^{\bullet}T1 = \{P1\}$
- $T1^{\bullet} = \{P2, P3\}$
- ${}^{\bullet}T2 = \{P2\}$
- $T2^{\bullet} = \{P4\}$
- ${}^{\bullet}T3 = \{P3, P4\}$
- $T3^{\bullet} = \{P0\}$

Für $M_0 = P0$ werden die Transitionen, deren Vorbereich nur die Stelle $P0$ enthält, betrachtet. Dies ist genau die Transition $T0$. Also muss nach dem Ausblenden aller nicht-aktivierten Transitionen für ENS_1 in der Markierung M_0 lediglich die Transition $T0$ sichtbar sein. Nach dem Feuern der Transition $T0$ befindet sich ENS_1 in einer Folgemarkierung von M_0, die an dieser Stelle mit M' bezeichnet wird. Nach Definition des Feuerbegriffs in Abschnitt 2.2 sollte die Folgemarkierung wie folgt definiert sein: $M' = \{P1\}$. Dieses Verhalten wird folgend in der modifizierten Version von PIPE nachgestellt.



(a) ENS_1 in Startmarkierung M_0

(b) ENS_1 in Markierung M' bei $M_0[T0\rangle M'$

Abb. 6.2.: ENS_1 mit ausgeblendeten nicht-aktivierten Transitionen

Wie in Abbildung 6.2 zu sehen ist, entspricht auch die Simulation in der Erweiterung von PIPE der formalen Beschreibung von M'. Hiermit ist gezeigt, dass zumindest für diesen einfachen Fall die Simulation mit der Berechnung übereinstimmt.

6.2. Überprüfung des Petri-Netzes für das Spiel Solitär

Nachdem das modifizierte Programm über einfache Testfälle validiert wurde, wird folgend das aus Kapitel 3 bekannte Petri-Netz für Solitär verwendet um diverse Spielsituationen nachzustellen. Es werden im Rahmen dieser Arbeit drei Spielsituationen betrachtet: Zunächst muss der Spielstart korrekt modelliert sein, damit induktiv jede Folgesituation aus einem validen Zustand heraus erreicht wird. Dies bildet die Grundlage der Validierung. Anschließend wird über die von PIPE entsprechend bereitgestellte Funktion eine

endliche Anzahl von zufällig gewählten aktivierten Transitionen gefeuert. Zum Abschluss dieses Kapitels werden jeweils die Niederlage- als auch die Gewinnsituation des Spiels validiert.

6.2.1. Spielstart

Zu Beginn des Spiels befindet sich das in PIPE modellierte inhibitorische S/T-System in der Startmarkierung. Diese ist mit angezeigten nicht-aktivierten Transitionen bereits in Abbildung 4.5 zu sehen. Das gleiche System mit ausgeblendeten nicht-aktivierten Transitionen, also nach der Betätigung des Buttons in der Animationsleiste, sollte ausschließlich die Transitionen einblenden, die auch aktiviert sind.

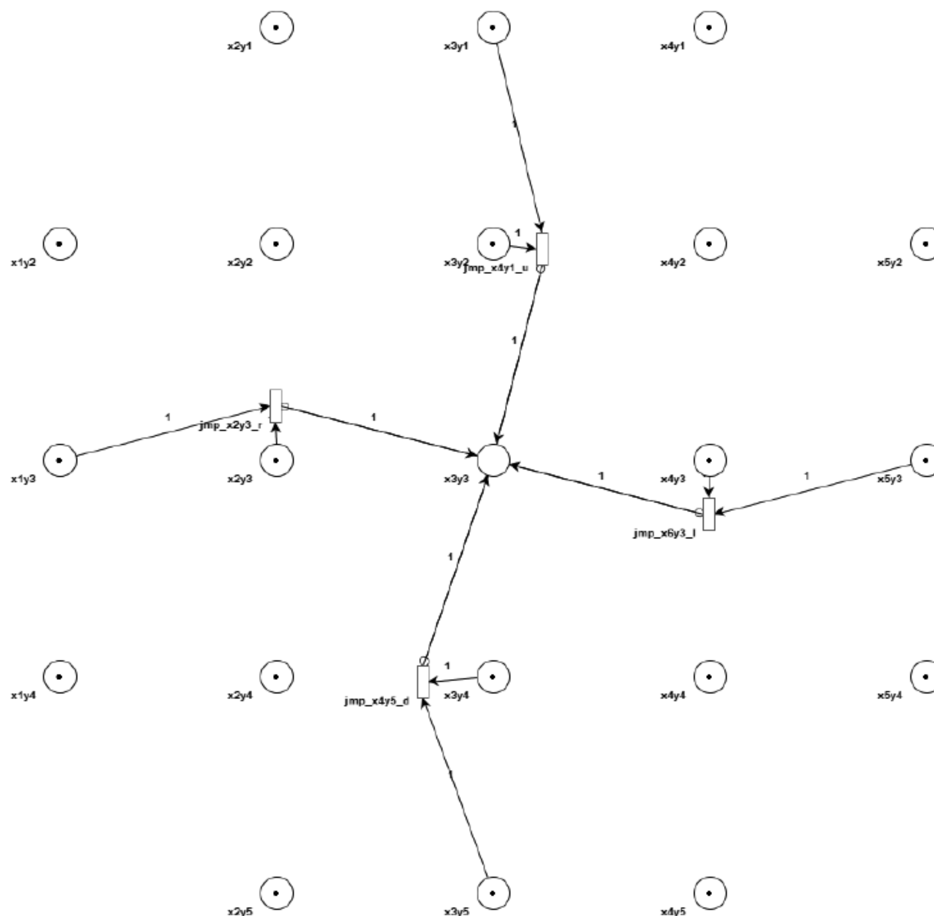


Abb. 6.3.: Inhibitorisches S/T-System mit ausgeblendete nicht-aktivierten Transitionen

Abbildung 6.3 zeigt das S/T-System für Solitär in PIPE, nachdem der Button zum Ausblenden nicht-aktivierter Transitionen gedrückt wurde.

6.2.2. Zufällige Züge

In diesem Validierungsfall wird ausschließlich darauf geachtet, dass das in PIPE angezeigte System auf das Feuern von zufälligen Transitionen reagiert. Diese Validierung findet ausschließlich über die Simulation innerhalb von PIPE statt und bedient sich am inhibitorischen S/T-System für Solitär, wie es bereits in Abbildung 4.5 beziehungsweise Abbildung 6.3 gezeigt wurde.

6.2.3. Niederlage

Die Niederlage ist definiert als Spielsituation, in der kein Zug mehr möglich ist, also keine Transition mehr erreichbar ist, es aber mehr als einen Token im Netz selbst gibt und die Stelle $x3y3$ nicht mit einem Token belegt ist. In dieser Markierung darf bei ausgeblendeten nicht-aktivierten Transitionen demnach keine Transition und keine Kante gezeichnet werden. Eine Markierung dieser Art wird zum Beispiel über das Schalten in folgender Reihenfolge erreicht:

1. jmp_x2y3_r
2. jmp_x3y5_d
3. jmp_x5y5_l
4. jmp_x3y2_u
5. jmp_x1y2_r
6. jmp_x4y3_u
7. jmp_x1y4_d
8. jmp_x3y4_l
9. jmp_x4y6_d
10. jmp_x3y6_d
11. jmp_x4y2_l
12. jmp_x3y0_u
13. jmp_x6y2_l
14. jmp_x5y0_u
15. jmp_x4y2_r
16. jmp_x6y3_l
17. jmp_x2y2_r
18. jmp_x7y2_l
19. jmp_x7y4_d
20. jmp_x5y2_l
21. jmp_x4y0_u
22. jmp_x3y2_r
23. jmp_x4y3_u
24. jmp_x5y4_r

Diese Schaltfolge wurde mit der *randomly fire transition* Funktion von PIPE erzeugt, die so lange betätigt wurde, bis die oben genannte Niederlage-Situation eingetreten ist. Die Zahl vor dem Namen der Transition bezieht sich auf die Reihenfolge der Feuerung der Transitionen.

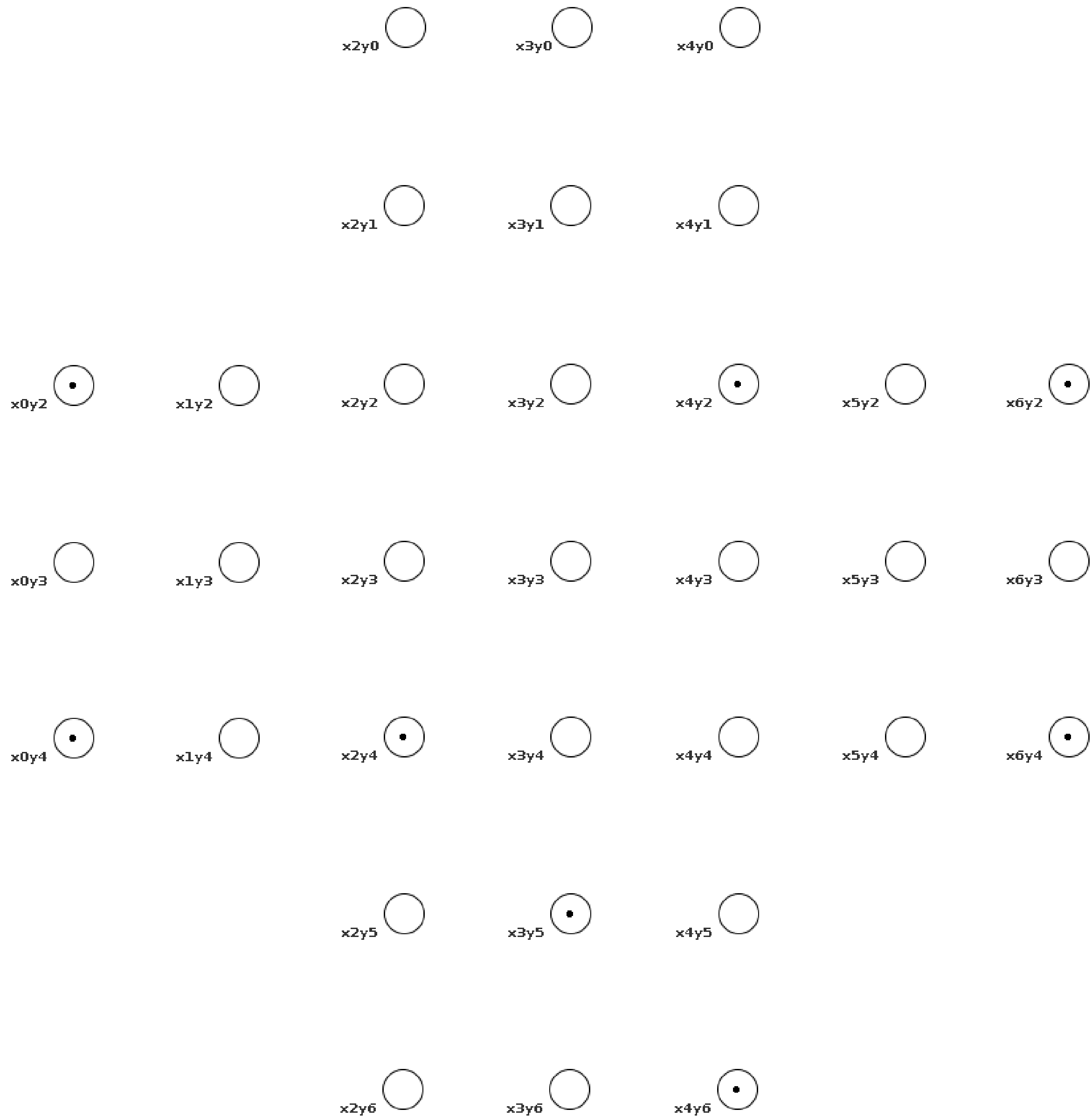


Abb. 6.4.: Niederlage-Situation im Petri-Netz für Solitär

Abbildung 6.4 zeigt das inhibitorische S/T-System von Solitär nach dem sukzessiven Schalten der Transitionen zum Erreichen einer Niederlage-Situation. Die Voraussetzungen für eine Niederlage des Spiels sind erfüllt: Es sind keine Transitionen, keine Kanten und mehr als ein Token im System zu sehen.

6.2.4. Sieg

Eine Sieg-Situation hingegen ist erreicht, wenn kein Zug mehr möglich ist, keine Transition mehr erreichbar ist und nur die Stelle $x3y3$ mit einem Token belegt ist. Zur Validierung einer solchen Situation muss zunächst eine Transitionsfolge gefunden werden, mit der all diese Bedingungen erfüllt sind. Als Vorlage für diese Folge wird die Lösung von James Yates (vgl. [yat]) verwendet.

1. jmp_x2y3_r
2. jmp_x3y5_d
3. jmp_x1y4_r
4. jmp_x4y4_l
5. jmp_x6y4_l
6. jmp_x5y6_d
7. jmp_x5y3_u
8. jmp_x3y6_r
9. jmp_x5y6_d
10. jmp_x3y2_u
11. jmp_x3y0_u
12. jmp_x5y1_u
13. jmp_x5y3_u
14. jmp_x5y5_l
15. jmp_x3y5_d
16. jmp_x3y3_d
17. jmp_x1y2_u
18. jmp_x1y4_r
19. jmp_x3y4_r
20. jmp_x7y2_l
21. jmp_x4y2_r
22. jmp_x7y4_d
23. jmp_x7y2_l
24. jmp_x5y0_l
25. jmp_x3y0_u
26. jmp_x2y2_r
27. jmp_x4y2_r
28. jmp_x6y2_u
29. jmp_x6y4_l
30. jmp_x4y4_d
31. jmp_x4y1_u

Wenn man diese Transitionsfolge in der genannten Reihenfolge feuert, befindet sich das inhibitorische S/T-System in einer Markierung, in der ausschließlich die Stelle $x3y3$ mit einem Token markiert ist.

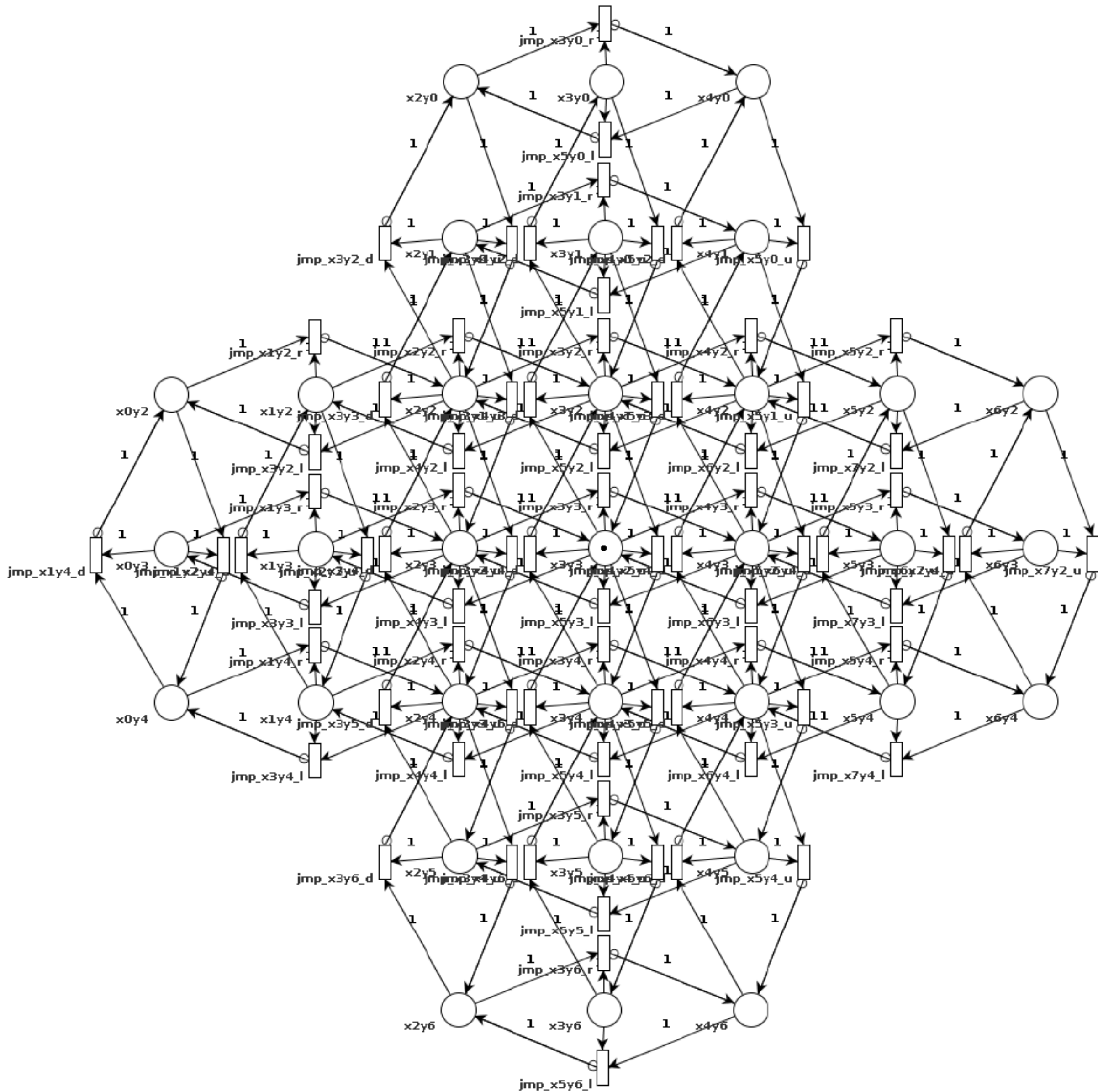


Abb. 6.5.: Solitär gewonnen ohne Ausblenden nicht-aktivierter Transitionen

In Abbildung 6.5 wird gezeigt, wie das inhibitorische S/T-System nach dem sukzessiven Feuern aller Transitionen, die zum Gewinn des Spiels führen, in PIPE angezeigt wird. An dieser Stelle werden nicht-aktivierte Transitionen und ihre ein- und ausgehenden Kanten angezeigt. Das identische Netz, nur mit ausgeblendeten nicht-aktivierten Transitionen, ist in Abbildung 6.6 zu sehen.

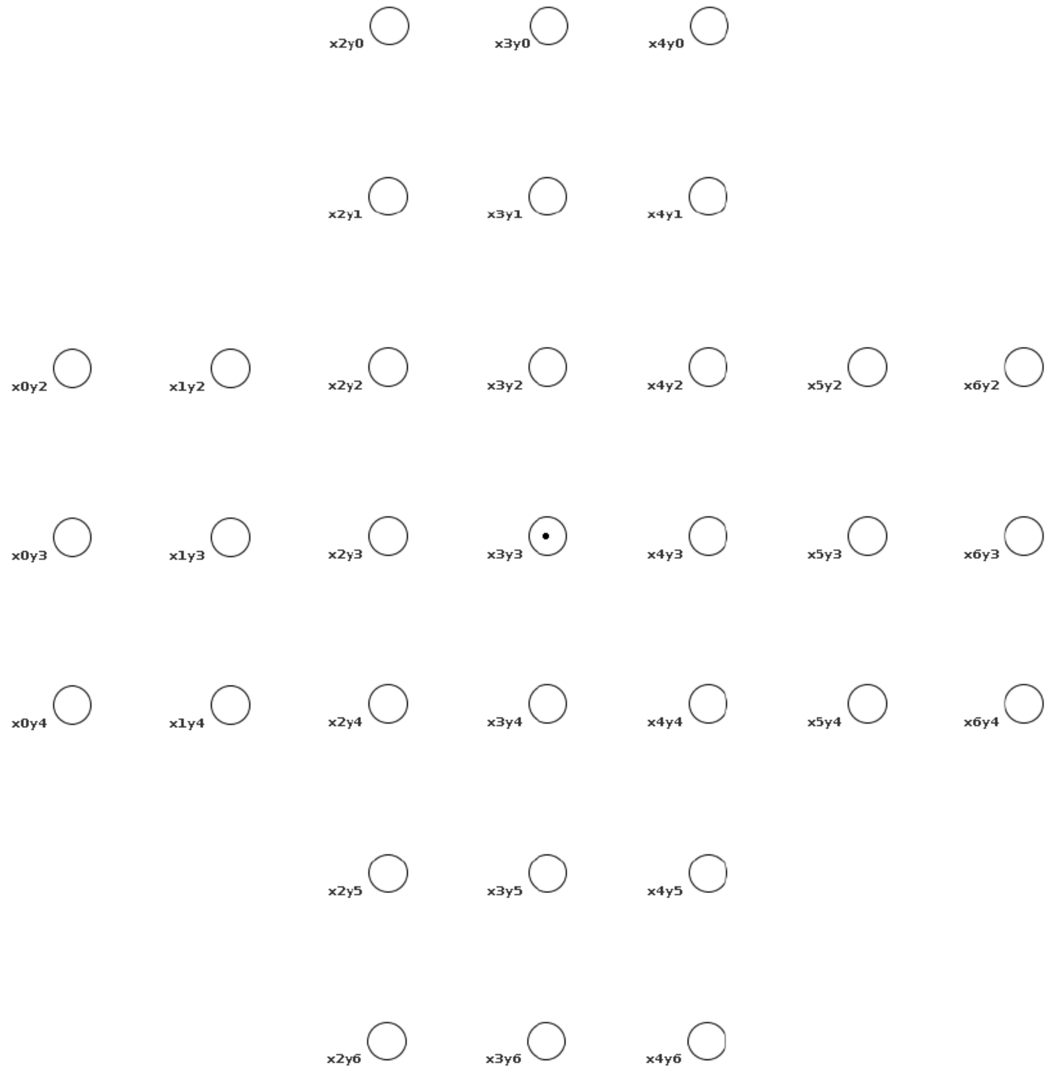


Abb. 6.6.: Solitär gewonnen mit Ausblenden nicht-aktivierter Transitionen

Die einzige Stelle, die in Abbildung 6.6 mit einem Token belegt ist, ist $x3y3$, dies entspricht der in Abschnitt 3.1 definierten Gewinnsituation. Es ist ebenfalls gut zu erkennen, dass das Spiel nicht weitergeführt werden kann, da es keine aktivierten Transitionen mehr gibt.

7. Fazit und Ausblick

Das Ausblenden nicht-aktivierter Transitionen ist bei komplexen Netzen wie zum Beispiel Modellnetzen von Brettspielen eine Methodik, die die Übersichtlichkeit eines Petri-Netzes stark erhöhen kann. In dem Beispielnetz für Solitär, das während dieser Arbeit konstruiert wurde, wird die Auswirkung direkt sichtbar. Mit erhöhter Komplexität eines Petri-Netzes steigt auch die Nützlichkeit der in dieser Arbeit programmierten Funktion in PIPE. Ein weiterer Vorteil, der im Rahmen dieser Arbeit nicht vorgestellt wurde, ist, dass in einem System mit ausgeblendeten nicht-aktivierten Transitionen sogenannte Deadlocks (Markierungen, in denen es keine aktivierte Transitionen gibt) direkt visualisiert werden, da im Fall eines Deadlocks keinerlei Kanten sichtbar sind. Eine solche Situation ist zum Beispiel die Niederlage- bzw. Sieg-Situation im Petri-Netz für das Spiel Solitär.

Es gibt jedoch Systeme, bei denen auch die nicht-aktivierten Transitionen angezeigt werden sollten. In diesen Systemen erhöht das Ausblenden dieser Transitionen die Übersichtlichkeit nicht. Wird zum Beispiel eine Transitionsfolge gesucht, um eine bestimmte Transition t aktivierbar zu machen, müssen auch nicht-aktivierte Transitionen angezeigt werden. Ebenfalls würde, besonders bei kleinen Petri-Netzen wie zum Beispiel dem Erzeuger/Verbraucher-System das Ausblenden nicht-aktivierter Transitionen die Übersichtlichkeit nicht erhöhen. Ein ähnliches Beispiel sind Petri-Netze, die zwar eine hohe Komplexität, allerdings auch für eine beliebige Markierung eine hohe Anzahl an aktivierten Transitionen aufweisen. Bei diese Netzen erhöht das Ausblenden der nicht-aktivierten Transitionen die Übersichtlichkeit nicht, da es keine hohe Anzahl an Elementen gibt, die nicht mehr gezeichnet werden sollen.

Die in dieser Arbeit erzeugten Änderungen sind ausschließlich für das Programm PIPE getätigt. Da PIPE nur einen Teil aller möglichen Petri-Netze modellieren kann, konnten weitere Netztypen wie zum Beispiel gefärbte Petri-Netze nach [Jen91] nicht validiert werden. Da gefärbte Petri-Netze selbst schon die Übersichtlichkeit der Modellierungen erhöhen und versuchen, die Anzahl der Kanten zu reduzieren, hat das Ausblenden nicht-aktivierter Transitionen bei diesen Netztypen eine geringere Auswirkung auf die Übersichtlichkeit der Netze. In der Vergangenheit getätigte Aussagen wie zum Beispiel der Nichteignung von S/T-Systemen für Brettspielen (vgl. [Sch16]) können durch die Ergebnisse dieser Arbeit neu betrachtet werden, da durch das Ausblenden nicht-aktivierter Transitionen die Übersichtlichkeit komplexer Netze erhöht wird.

Der im Rahmen dieser Arbeit erzeugte Quelltext und die Änderungen am bereits bestehenden Quelltextes des Programms PIPE werden im Anschluss an die Arbeit an die Betreuer des Programm übergeben und für mögliche weitere Arbeiten der Öffentlichkeit frei verfügbar gemacht.

Abkürzungsverzeichnis

Abb. Abbildung

Tab. Tabelle

vgl. Vergleiche

bzw. Beziehungsweise

XML Extensible Markup Language

PNML Petri Net Markup Language

GIMP GNU Image Manipulation Program

UML Unified Modeling Language

PNG Portable Network Graphics

PIPE Platform Independent Petri Net Editor

STS Stellen/Transitionssystem

ENS Elementares Netzsystem

Literaturverzeichnis

[Bau97] Bernd Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* Spektrum Akademischer Verlag, 1997. 2

[BCG04] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays, Volume 4.* A K Peters/CRC Press, 2004. 3.1, 3.1, 6

[BLPJK18] Pere Bonet, Catalina Llado, Ramon Puigjaner, and William J. Knottenbelt. Pipe v2.5: a petri net tool for performance modeling. 04 2018. 4.3

[Gru96] Volker Gruhn. Geschäftsprozeß-management als grundlage der softwareentwicklung. *Informatik Forschung und Entwicklung*, 11(2):94–101, May 1996. 2

[JDJKS09] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. Pipe2: A tool for the performance evaluation of generalised stochastic petri nets. 36:34–39, 01 2009. 4

[Jen91] Kurt Jensen. Coloured petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 342–416, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. 7

[JMMT06] Christopher Jefferson, Angela Miguel, Ian Miguel, and S. Armagan Tarim. Modelling and solving english peg solitaire. *Computers and Operations Research*, 33(10):2935 – 2959, 2006. Part Special Issue: Constraint Programming. 3.1, 3.1, 3.1

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Hamburg, 1962. 2

[PW08] L. Priese and H. Wimmel. *Petri-Netze.* eXamen.press. Springer Berlin Heidelberg, 2008. 2, 2.2, 2.2, 2.4

[Rei10] Wolfgang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien (Leitfäden der Informatik).* Vieweg+Teubner Verlag, 2010. 2

[Sch16] Björn Scheetz. Werkzeugbasierte modellierung von spielen mit petrinetzen. 2016. 2, 7

[WK03] Michael Weber and Ekkart Kindler. *The Petri Net Markup Language*, pages 124–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 4.3

[yat] Peg solitaire solution. http://www.chessandpoker.com/peg-solitaire-solution.html. 6.2.4

Abbildungsverzeichnis

2.1. Einfaches Netzsystem NET_1 3
2.2. Elementares Netzsystem ENS_1 4
2.3. ENS_1 nach Schalten von T2 5
2.4. Stellen-/Transitions-System STS_0 7
2.5. Stellen/Transitions-System STS_0 nach $M_0[T0\rangle M'$ 8
2.6. STS mit Inhibitorkanten 9

3.1. Spielfeldvarianten von Solitär 10
3.2. Solitär Spielzug 11
3.3. Solitär Spielfeld in einem Grid 12
3.4. Transition von $K_{sol} = (x_2y_3, x_3y_3, x_4y_3)$ 15
3.5. Solitär in initialer Markierung als Elementares Netzsystem ENS_{sol} 16
3.6. Detailansicht eines Übergangstripel 17

4.1. Hauptbildschirm von PIPE in der Version 5.0.2 18
4.2. Abbildung einer Sprung-Transitionen 20
4.3. Solitär-Spielfeld als Petri-Netz 21
4.4. Aufbau von Verbindungen mit der addArc() Funktion 24
4.5. Solitär als inhibitorisches S/T-System 25
4.6. Aktivierte Transition in PIPE 5.0.2 26

5.1. Klassendiagramm des PIPECore Teilprojekts 28
5.2. Detailansicht des PIPECore Teilprojekts 29
5.3. Klassendiagramm des PIPE-GUI Teilprojekts 33
5.4. Klassendiagramm AnimateActionManager 37
5.5. Erweiterte Icon-Leiste in PIPE 38

6.1. Beispielnetz *Dining philosophers.xml* STS_{dine} 40
6.2. ENS_1 mit ausgeblendeten nicht-aktivierten Transitionen 41
6.3. Inhibitorisches S/T-System mit ausgeblendete nicht-aktivierten Transitionen 42
6.4. Niederlage-Situation im Petri-Netz für Solitär 44
6.5. Solitär gewonnen ohne Ausblenden nicht-aktivierter Transitionen 46
6.6. Solitär gewonnen mit Ausblenden nicht-aktivierter Transitionen 47

Quelltextverzeichnis

4.1. Ausschnitt des Generator-Programms 22
4.2. Ausschnitt der addArc-Funktion 23

5.1. Erweiterungen des PetriNetComponent Interface 30
5.2. Erweiterungen des PetriNetComponent Interface 31
5.3. Dummy-Implementierung 31
5.4. Implementierung der Methoden 32
5.5. Änderungen an der GuiAnimator-Klasse 34
5.6. paintComponent-Funktion der Klasse TransitionView 35
5.7. paintComponent Funktion der Klasse ArcView 36
5.8. Quelltextauszug der Klasse AnimateActionManager 38
5.9. Quelltextauszug der Klasse ToggleHideDisabledTransitionsAction 38

Eidesstattliche Versicherung

Ich, Kristof Kipp, Matrikel-Nr. 3036176, versichere hiermit, dass ich die vorgelegte Bachelor-Arbeit mit dem Thema

Ausblenden nicht-aktivierter Transitionen im Petri-Netz-Werkzeug PIPE

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als Solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bremen, 30. August 2018

____

Kristof Kipp

A. Anhang

A.1. Beiliegende CD

A.1.1. Inhaltsverzeichnis der CD

1. Die Bachelorarbeit als PDF-Datei
2. Alle verwendeten Online-Quellen als PDF-Dateien
3. Der Quelltext des Erzeugerprogramms aus Abschnitt 4.3
4. Der Quelltext des modifizierten PIPE-Programms
5. Eine Ausführbare Version der modifizierten PIPE-Programms
6. Sämtliche Diagramme, Graphiken und Screenshots der Bachelorarbeit