Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Christian Hoff

Transformationseinheiten als Kontrollstruktur für rekonfigurierbare Petrinetze in ReConNet

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Hoff

Transformationseinheiten als Kontrollstruktur für rekonfigurierbare Petrinetze in ReConNet

Masterarbeit eingereicht im Rahmen der Masterprüfung

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

Betreuende Prüferin: Prof. Dr. Julia Padberg
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 4. August 2016

Christian Hoff

Thema der Arbeit
Transformationseinheiten als Kontrollstruktur für rekonfigurierbare Petrinetze in ReConNet

Stichworte
Rekonfigurierbare Petrinetze, regelbasierte Rekonfiguration, Kontrollstrukturen, Transformationseinheiten, ReConNet

Kurzzusammenfassung
Rekonfigurierbare Petrinetze erlauben die Modellierung dynamischer Systeme, welche sich zur Laufzeit in ihrer Struktur verändern können. Die Rekonfiguration erfolgt hierbei mit Hilfe von Regeln. Um die Anwendung von Regeln präziser steuern zu können, werden sogenannte Kontrollstrukturen verwendet. Transformationseinheiten bilden eine solche Kontrollstruktur und ermöglichen die Bündelung mehrerer Regeln zu einer zusammengesetzten komplexen Transformation. Im Rahmen dieser Arbeit werden Transformationseinheiten in das Simulationstool ReConNet implementiert und deren Mehrwert für die Modellierung rekonfigurierbarer Petrinetze evaluiert. Die Evaluation der Transformationseinheiten erfolgt anhand einer Fallstudie.

Christian Hoff

Title of the paper
Transformation units as a control structure for reconfigurable Petri nets in ReConNet

Keywords
Reconfigurable Petri nets, rule-based reconfiguration, control structures, transformation units, ReConNet

Abstract
Reconfigurable Petri nets are used to model dynamic systems, which are able to reshape their structure at runtime. The reconfiguration is performed by the application of rules. To control the application of rules in a precise manner, so-called control structures are used. Transformation units belong to the group of control structures. They provide the ability to package multiple rules into a compound complex transformation. This work is about the evaluation of transformation units and how they can be used for modelling systems with reconfigurable Petri nets. The evaluation is based on a case study. As a part of this work, transformation units are implemented in the simulation tool ReConNet.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In Zeiten der Digitalisierung, insbesondere der Digitalisierung der Wirtschaft, werden zunehmend Prozesse mit Hilfe von Informationstechnologien umgesetzt und automatisiert. Häufig handelt es sich hierbei um hochdynamische und nebenläufige Prozesse, welche aufgrund ihrer Komplexität schwer nachzuvollziehen sind. Für die Umsetzung dieser Prozesse werden daher intuitive Techniken benötigt, um die wachsende Komplexität beherrschen zu können. Petrinetze sind ein mächtiges Werkzeug zur Modellierung dynamischer Systeme und können für diese Aufgaben herangezogen werden.

Bei der Modellierung wird das jeweilige System als Petrinetz abgebildet. Ein Petrinetz ist ein gerichteter bipartiter Graph und verfügt über zwei Arten von Knoten: Stellen und Transitionen. Stellen können Token aufweisen und dienen dazu, den Zustand des Systems abzubilden, wohingegen Transitionen für die Zustandsübergänge zuständig sind. Stellen können mit Hilfe von Kanten mit Transitionen verbunden werden und umgekehrt. Die Dynamik von Petrinetzen ergibt sich durch das Schalten von Transitionen und der einhergehenden Veränderung der Stellen-Markierung. Schaltet eine Transition, so nimmt sie Token von Stellen ihres Vorbereichs (Stellen der eingehende Kanten) und legt sie auf die Stellen ihres Nachbereichs (Stellen der ausgehende Kanten). Durch die grafische Darstellung von Petrinetzen kann das Schaltverhalten intuitiv nachvollzogen werden. Des Weiteren sind Petrinetze formal spezifiziert, wodurch die modellierten Systeme auf Eigenschaften, wie zum Beispiel Verklemmungsfreiheit oder Lebendigkeit, analysiert werden können. Petrinetze stellen somit ein mächtiges Werkzeug für die Modellierung dynamischer Systeme dar.

In einer sich schnell verändernden digitalen Welt müssen Prozesse immer flexibler werden und sich gegebenenfalls zur Laufzeit an neue Situationen anpassen. Diese strukturellen Veränderungen verlangen eine neue Art der Dynamik, welche von einem klassischen Petrinetz nicht erfüllt werden kann. Für die Modellierung dieser Systeme ist sowohl eine Dynamik auf

Prozessebene, als auch auf Strukturebene notwendig. Rekonfigurierbare Petrinetze bieten diese Möglichkeit.

Rekonfgurierbare Petrinetze erlauben die Modellierung dynamischer Systeme, welche sich zur Laufzeit in ihrer Struktur verändern können. Auf diese Weise können sich verändernde Prozesse abgebildet werden. Beispiele hierfür wären unter anderem dynamische Infrastrukturen, biochemische Vorgänge oder evolvierende Geschäftsprozesse. Die Rekonfiguration von Petrinetzen erfolgt mit Hilfe von Regeln. Eine Regel legt dabei fest, welche Vorbedingungen das zu transformierende Netz erfüllen muss und inwieweit das Netz letztendlich transformiert wird. Um die Dynamik von rekonfigurierbaren Petrinetzen auf Prozess- und Strukturebene zu steuern, werden sogenannte Kontrollstrukturen eingesetzt. Sie werden dazu verwendet, das Schalt- und Transformationsverhalten zu beeinflussen und erweitern somit die Ausdrucksmächtigkeit bei der Modellierung rekonfigurierbarer Petrinetze. Beispiele für solche Kontrollstrukturen sind etwa Stellen- und Transitionsnamen, Transitionslabels, Inhibitor-Kanten sowie negative Anwendungsbedingungen.

Die Abbildung realer Probleme mit rekonfigurierbaren Petrinetzen führt häufig zu großen Netzen mit vielen Regeln. Komplexe Prozesse können hierbei größtenteils nur mit komplexen Regeln abgebildet werden. Um die Komplexität beherrschen zu können ist es sinnvoll, Teilprozesse zu identifizieren und diese anschließend zu komplexen Prozessen zusammenzufügen. Transformationseinheiten ermöglichen diesen Vorgang. Hierbei werden Regeln mit Hilfe eines Kontrollausdrucks zu Einheiten zusammengefügt, welche anschließend auf das Netz angewendet werden können. Durch die Identifizierung von Teilprozessen wird die Wiederverwendbarkeit von Regeln deutlich erhöht. Transformationseinheiten helfen somit bei der Modellierung komplexer Probleme mit rekonfigurierbaren Petrinetze.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist eine umfassende Auseinandersetzung mit Kontrollstrukturen für rekonfigurierbare Petrinetze, insbesondere den Transformationseinheiten. An der HAW Hamburg wurde das Tool ReConNet entwickelt, welches die Modellierung und Simulation rekonfigurierbarer Petrinetze ermöglicht. ReConNet unterstützt hierbei bereits diverse Kontrollstrukturen. Im Rahmen dieser Arbeit soll ReConNet um Transformationseinheiten erweitert werden. Abschließend soll evaluiert werden, welchen Mehrwert Transformationseinheiten bei der Modellierung rekonfigurierbarer Petrinetze bieten.

1.3 Aufbau der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. Nach dem einleitenden Kapitel 1, folgt in Kapitel 2 die Einführung in die Grundlagen. Hier werden die theoretischen Grundlagen sowie etwaig verwendete Techniken erläutert, welche im Kontext dieser Arbeit relevant sind.
In Kapitel 3 werden Kontrollstrukturen für rekonfigurierbare Petrinetze erläutert. Dies umfasst unter anderem diverse Kontrollstrukturen auf Basis dekorierter Petrinetze. Des Weiteren werden negative Anwendungsbedingungen sowie Transformationseinheiten als Kontrollstrukturen für rekonfigurierbare Petrinetze erläutert. Weiterhin erfolgt ein Überblick über die in ReConNet verfügbaren Kontrollstrukturen.
Kapitel 4 befasst sich mit der Konzeption von Transformationseinheiten im Kontext von ReConNet. Es wird unter anderem der Algorithmus zur Durchführung von Transformationseinheiten erläutert. Des Weiteren wird das Konzept zur grafischen Oberfläche vorgestellt.
In Kapitel 5 erfolgt die Erläuterung der Implementierung von Transformationseinheiten in ReConNet. Dies umfasst die technische Abbildung der Transformationseinheiten, sowie die letztendliche Umsetzung des konzipierten Algorithmus.
In Kapitel 6 erfolgt der Test der implementierten Funktionen. Hierbei werden Testfälle zur Durchführung von Transformationseinheiten angeführt und deren Testergebnisse dargelegt.
Kapitel 7 befasst sich mit der Evaluation von Transformationseinheiten. Anhand einer Fallstudie wird evaluiert, inwieweit Transformationseinheiten als Kontrollstruktur für die Modellierung rekonfigurierbarer Petrinetze herangezogen werden können und welchen Mehrwert sie hierbei bieten.
Im letzten Kapitel 8 erfolgt eine Zusammenfassung dieser Arbeit sowie ein Ausblick auf mögliche aufbauende Arbeiten.

2 Grundlagen

2.1 Petrinetze

Petrinetze, basierend auf der Arbeit von Carl Adam Petri [Pet62], stellen ein Werkzeug für die Modellierung dynamischer, nebenläufiger und verteilter Systeme dar.

Ein Petrinetz ist ein gerichteter bipartiter Graph und verfügt über zwei Arten von Knoten: Stellen und Transitionen. Die Knoten eines Petrinetzes sind mittels Kanten verbunden. Eine Kante kann hierbei entweder eine Stelle mit einer Transition, oder eine Transition mit einer Stelle verbinden und verfügt über ein Gewicht. Auf den Stellen des Netzes können sogenannte *Token* liegen. Die maximale Anzahl von Token auf einer Stelle kann mit einer *Kapazität* bestimmt werden. Eine bestimmte Verteilung von Token im Netz wird auch *Markierung* genannt. Die klassische Form des Petrinetzes, das sogenannte **S**tellen/**T**ransitions-Netz, kann durch das Tupel $N = (P, T, pre, post, m, cap)$ ausgedrückt werden (Matrixdarstellung).

Definition 1 (Stellen/Transitions-Netz)**.** *Ein Stellen/Transitions-Netz mit Kapazitäten in der Matrixdarstellung ist ein Tupel $N = (P, T, pre, post, m, cap)$, für das gilt:*

- *P, eine endliche Menge der Stellen*
- *T, eine endliche Menge der Transitionen*
- *pre, Matrix aus $P \times T \to \mathbb{N}_0$ - definiert die Anzahl zu nehmender Token von Stellen des Vorbereichs beim Schalten der jeweiligen Transition*
- *post, Matrix aus $P \times T \to \mathbb{N}_0$ - definiert die Anzahl abzulegender Token auf Stellen des Nachbereichs beim Schalten der jeweiligen Transition*
- *m, der Markierungsvektor $P \to \mathbb{N}_0$ - definiert die (Anfangs-)Markierung des Netzes. Ordnet jeder Stelle die Anzahl der jeweils vorliegenden Token zu*
- *cap, der Kapazitätsvektor $P \to \mathbb{N}_+^\omega$ - ordnet jeder Stelle eine Kapazität für Token zu, wobei ω eine unbegrenzte Kapazität bedeutet*

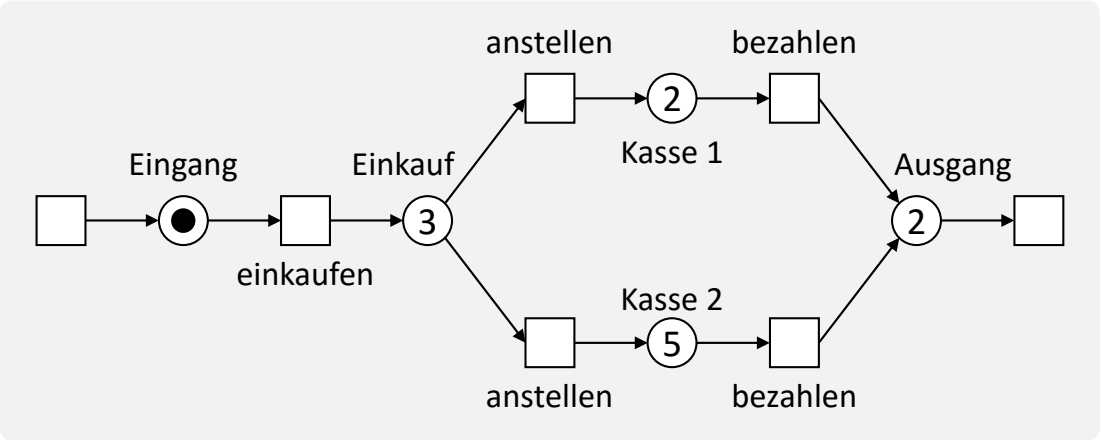

Abbildung 2.1: Supermarkt als Petrinetz

Die Matrizen pre und $post$ definieren dementsprechend die Kanten und Kantengewichtungen zwischen Stellen und Transitionen. Ist das Kantengewicht einer Kante 0, wird diese Kante in der grafischen Darstellung des Petrinetzes nicht dargestellt (quasi nicht-existent).

Die Dynamik eines Petrinetzes ergibt sich durch das Schalten von Transitionen und der einhergehenden Veränderung der Markierung des Netzes. Dieser Vorgang wird auch als Tokenspiel (engl. *token game*) bezeichnet. Das in Abbildung 2.1 dargestellte Petrinetz soll zur Erläuterung des Tokenspiels dienen. Das Petrinetz kann hierbei als Spielbrett betrachtet werden, wobei Token als Spielfiguren auf Stellen platziert werden können. Jede Transition steht hierbei für einen möglichen Spielzug. Ein Spielzug ist dann möglich, wenn die Stellen der eingehenden Kanten einer Transition mindestens so viele Token haben, wie die jeweilige Kante, gemessen an ihrem Gewicht, verlangt. Man spricht hierbei auch von einer *aktivierten* Transition. Die Transition *einkaufen* ist demnach aktiviert, da auf der Stelle *Eingang* ein Token liegt und die Kante zur Transition ein Gewicht von 1 hat. Ein Spielzug ist dementsprechend möglich, oder anders ausgedrückt: die Transition kann schalten. Beim Schalten einer Transition werden Token von Stellen eingehender Kanten entfernt und auf Stellen ausgehender Kanten hinzugefügt. Wie viele Token entfernt und hinzugefügt werden ist abhängig von den jeweiligen Kantengewichten. Beim Schalten der Transition *einkaufen* würde ein Token von der Stelle *Eingang* entfernt werden und die Stelle *Einkauf* würde ein zusätzliches Token erhalten.

Wie in Abbildung 2.1 zu sehen ist, kann das Verhalten eines Netzes intuitiv anhand der grafischen Darstellung nachvollzogen werden. Die Kunden, repräsentiert durch Token, be-

treten den Supermarkt, erledigen ihre Einkäufe, bezahlen an einer der Kassen und verlassen schließlich den Supermarkt. Neben dieser Anschaulichkeit haben Petrinetze noch einen weiteren Vorteil: der Formalismus von Petrinetzen ist mathematisch fundiert. Dies ermöglicht die formale Verifikation von Netzeigenschaften der modellierten Systeme. Durch die Analyse des sogenannten Erreichbarkeitsgraphen lassen sich beispielsweise Rückschlüsse auf Netzeigenschaften wie Verklemmungsfreiheit, Beschränktheit oder Lebendigkeit schließen. Aufgrund der Simplizität, Allgemeingültigkeit und Möglichkeiten zur formalen Verifikation finden Petrinetze in vielen verschiedenen Bereichen Anwendung. Seit Einführung der Petrinetze von Carl Adam Petri wurden diverse weitere Klassen von Petrinetzen erforscht und untersucht. Beispiele hierfür sind gefärbte Petrinetze [Jen81], Netze mit individuellen Marken [Rei85] sowie Prädikat/Transitions-Netze [GL81].

Eine Erweiterung des klassischen S/T-Netzes sind dekorierte S/T-Netze. Hierbei wird die oben genannte Definition wie folgt erweitert $N = (P, T, pre, post, m, cap, pname, tname, tlb, rnw)$.

Definition 2 (dekoriertes Stellen/Transitions-Netz)**.** *Ein dekoriertes Stellen/Transitions-Netz ist ein Stellen/Transitions-Netz mit dem Tupel* $N = (P, T, pre, post, m, cap, pname, tname, tlb, rnw)$*, für das zusätzlich gilt:*

- *pname, eine Abbildung von Stellen auf Namen* $P \to A_P$
- *tname, eine Abbildung von Transitionen auf Namen* $T \to A_T$
- *tlb, eine Abbildung von Transitionen auf Label* $T \to W$
- *rnw, eine Abbildung von Transitionen auf einen Endomorphismus über* W

  $T \to END$ *mit* $END : W \to W$

  z.B. $rnw : \{something \mapsto \{(0,1),(1,2),(2,3),(3,4)...\}\}$

Abbildung 2.2 zeigt ein dekoriertes S/T-Netz. Die Stellen werden jeweils auf die Namen a bzw. b abgebildet, die Transitionen jeweils auf $reset$ bzw. $something$. Die Transition $something$ hat hierbei zusätzlich das Label 0. Eine Label-Erneuerungsfunktion $rnw : count$ würde beim Schalten der Transition $something$ das Label um eins erhöhen. Weitere Erneuerungsfunktionen wären beispielsweise not, zum invertieren boolscher Labels, oder id, welche das Label nicht verändert. Dekorationen verleihen eine höhere Ausdrucksmächtigkeit, welche unter anderem bei der Modellierung von Rekonfigurations-Regeln genutzt werden kann.

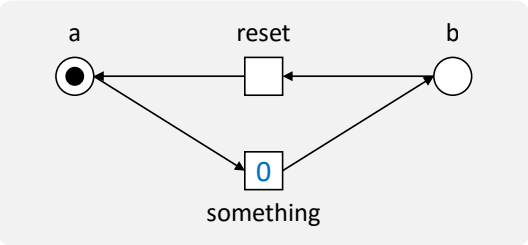

Abbildung 2.2: dekoriertes S/T-Netz

2.2 Rekonfigurierbare Petrinetze

Die Dynamik klassischer Petrinetze ergibt sich durch das Schalten von Transitionen und der damit einhergehenden Veränderung der Stellen-Markierung, welche den aktuellen Zustand des System repräsentiert. Dadurch ergibt sich eine Dynamik auf Prozessebene.

Rekonfigurierbare Petrinetze sind eine Erweiterung der klassischen (dekorierten) Petrinetze [EP04, LO04, EHP+07, PEHP08]. Bei rekonfigurierbaren Petrinetzen äußert sich die Dynamik zusätzlich darin, dass sich das jeweilige Petrinetz zur Laufzeit auch in seiner Struktur verändern kann. Dadurch ergibt sich eine Dynamik auf Strukturebene. Dies erlaubt eine Modellierung dynamischer Strukturen wie beispielsweise sich verändernde Geschäftsprozesse, Abläufe in dynamischen Infrastrukturen (z.B. flexible Fertigungssysteme), oder sonstige Prozesse, welche durch äußere Einflüsse verändert werden können (z.B. Prozesse im Living-Place der HAW). Die Rekonfiguration erfolgt mittels Regeln. Ein rekonfigurierbares Petrinetz wird demnach mit dem Tupel $RN = (N, \mathcal{R})$ ausgedrückt.

Definition 3 (Rekonfigurierbares Petrinetz)**.** *Ein rekonfigurierbares Petrinetz ist ein Tupel* $RN = (N, \mathcal{R})$*, für das gilt:*

- *N, das (dekorierte) Petrinetz*
- $\mathcal{R}$*, die Menge an Rekonfigurationsregeln*

Im Gegensatz zu klassischen Petrinetzen können bei rekonfigurierbaren Petrinetzen Teilsysteme zur Laufzeit ersetzt werden, anstatt das Gesamtsystem zu ersetzen. Dies ist vor allem in hochverfügbaren Systemen erstrebenswert, da die Rekonfiguration das laufende System nicht unterbricht. Sie bieten somit mehr Flexibilität gegenüber herkömmlichen Petrinetzen mit statischer Struktur. Des Weiteren skalieren rekonfigurierbare Petrinetze besser, da das Netz analog zu den jeweiligen Anforderungen wachsen und schrumpfen kann. Weiterhin

können sich dadurch Performance-Gewinne ergeben, da etwaige Graphalgorithmen auf einem durchschnittlich kleineren Netz arbeiten.

2.3 Netzmorphismen

Netzmorphismen bilden die Grundlage für die spätere Rekonfiguration von Petrinetzen. Ein Netzmorphismus $f : N_1 \to N_2 = \langle f_P : P_1 \to P_2, f_T : T_1 \to T_2 \rangle$ besteht aus den Abbildungen f_P und f_T, welche jeweils Stellen auf Stellen bzw. Transitionen auf Transitionen zwischen zwei Netzen N_1 und N_2 abbilden. Des Weiteren müssen folgende Bedingungen für einen Netzmorphismus erfüllt sein [Pad12, vgl. S.5-6].

Definition 4 (Netzmorphismus). *Gegeben seien zwei dekorierte Petrinetze* $N_i = (P_i, T_i, pre_i, post_i, m_i, cap_i, pname_i, tname_i, tlb_i, rnw_i)$ *für* $i \in \{1, 2\}$. *Für einen Netzmorphismus* $f : N_1 \to N_2$ *gilt:*

- *Bewahrung der Vor- und Nachbereiche*
  1. $f_P^{\oplus} \circ pre_1 = pre_2 \circ f_T$
  2. $f_P^{\oplus} \circ post_1 = post_2 \circ f_T$
- *Bewahrung der Markierungen und Kapazitäten*
  3. $m_1(p_1) \leq m_2(f_P(p_1))$ *für alle* $p \in P_1$
  4. $cap_1(p_1) = cap_2(f_P(p_1))$
- *Bewahrung der Dekorationen*
  5. $pname_1(p_1) = pname_2(f_P(p_1))$ *und* $tname_1(t_1) = tname_2(f_T(t_1))$
  6. $tlb_1(t_1) = tlb_2(f_T(t_1))$
  7. $rnw_1(t_1) = rnw_2(f_T(t_1))$

Bedingung 1 und 2 drücken aus, dass die Vor- und Nachbereiche der Transitionen, also die ein- und ausgehenden Kanten, hinsichtlich ihrer Gewichtungen in Summe erhalten bleiben müssen. Bedingung 3 legt fest, dass eine Stelle, auf die abgebildet wird, mindestens genau so viele Token hat wie ihr Urbild. Dies ist notwendig, um die Aktivierung einer Transition zu bewahren. Wenn eine Transition t_1 aktiviert ist, ist die entsprechende Transition $f_T(t_1)$ ebenfalls aktiviert. Bedingung 4 besagt, dass die Kapazitäten exakt erhalten bleiben müssen. Die restlichen Bedingungen 5 bis 7 legen fest, dass die Dekorationen wie Stellen- und

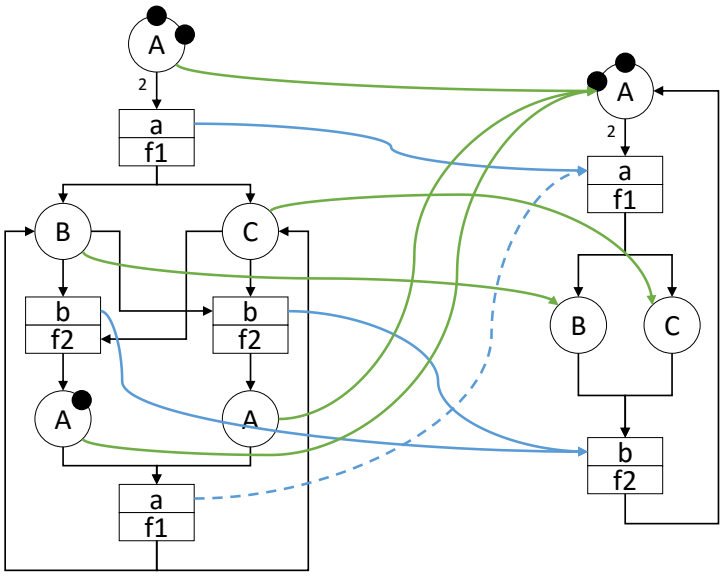

Abbildung 2.3: Netzmorphismus

Transitionsnamen, Transitionslabels sowie Erneuerungsfunktionen erhalten bleiben müssen. Eine strengere Form des Netzmorphismus ist der strikte Netzmorphismus, bei dem f_P und f_T injektiv sind und die Markierungen der Stellen exakt bewahrt werden müssen [RPL+08, S.6].

In Abbildung 2.3 ist ein Netzmorphismus vom linken ins rechte Netz dargestellt. Die gestrichelte Linie verdeutlicht dabei die in Definition 4 beschriebene Bedingung, dass die Vor- und Nachbereiche erhalten bleiben müssen. Im linken Netz hat die Transition a zwei separate eingehende Kanten von Stelle A. Im rechten Netz hat die Transition a hingegen nur eine eingehende Kante, jedoch mit der Gewichtung von 2, sodass der Vorbereich der Transition insgesamt erhalten bleibt. Die Markierungen, Kapazitäten sowie Dekorationen bleiben ebenfalls bewahrt.

2.4 Regelbasierte Rekonfiguration

Die Rekonfiguration von Petrinetzen erfolgt mittels Regeln. Wie bereits angeführt enthalten rekonfigurierbare Petrinetze eine Menge an Regeln, welche für die Rekonfiguration herangezogen werden $RN = (N, R)$. Eine Regel besteht aus einer linken Seite L und einer rechten Seite R, welche mittels striktem Netzmorphismus über ein sogenanntes Klebenetz K in Beziehung stehen $r = \langle L \to K \leftarrow R \rangle$.

Die Komponenten L, K und R sind jeweils (dekorierte) Petrinetze. Die jeweiligen Komponenten beschreiben, welche Teile des Netzes bei einer Rekonfiguration erhalten bleiben, hinzugefügt bzw. gelöscht werden. Die Anwendung einer Regel auf ein Netz wird auch Transformationsschritt genannt. Damit eine Regel angewendet werden darf, muss ein Netzmorphismus $o : L \to N$ existieren. Die linke Seite L einer Regel legt also fest, welcher Teil in dem zu transformierenden Netz vorhanden sein muss, damit diese Regel angewendet werden kann. Die Menge $K - R$ repräsentiert die Elemente, welche bei der Transformation gelöscht werden. $K - L$ die Elemente, die bei der Transformation hinzugefügt werden. Die Schnittmenge $L \cap R$ repräsentiert die Elemente, die bei der Transformation erhalten bleiben.

Cospan-Verfahren

Die Transformation des Netzes erfolgt durch ein algebraisches Ersetzungssystem. Im Kontext dieser Arbeit wird das sogenannte Cospan-Verfahren verwendet, welches eine Variante des geläufigeren Double-Pushout-Verfahrens darstellt. Die Transformation selbst wird in zwei Schritten durchgeführt. Im ersten Schritt werden zunächst alle Elemente $K - L$ zu N hinzugefügt, was in einem Zwischennetz D resultiert. Im zweiten Schritt werden nun alle Elemente $K - R$ aus D gelöscht. Hiermit entsteht das transformierte Netz N' (s. Abbildung 2.4).

Das Prinzip ist in Abbildung 2.5 beispielhaft dargestellt. Das Zwischennetz sowie die einzelnen Morphismen $L \to K$ und $R \to K$ wurden aus Gründen der Übersichtlichkeit nicht weiter betrachtet. Zunächst muss ein Netzmorphismus $o : L \to N$ gefunden werden. Anschließend wird das Netz gemäß der Regel transformiert. $K - L$, also der untere Teil aus K wird dem Netz hinzugefügt. $K - R$, also der obere Teil von K wird aus dem Netz entfernt. Auf diese Weise wird die alte Transition *something* mit der neuen Transition *somethingNew* ersetzt. Da beim Netzmorphismus o die Dekoration erhalten bleiben muss, kann erst dann ein Morphismus gebildet werden, wenn die Transition *something* 10 Mal geschaltet hat. Dekorationen können dementsprechend verwendet werden, um die Transformationen eines Netzes präziser zu steuern.

Double-Pushout vs. Cospan Double-Pushout

Das klassische Double-Pushout-Verfahren arbeitet umgekehrt. Im Gegensatz zu Cospan ($L \to K \leftarrow R$) gehen die Netzmorphismen von K ab zu den jeweiligen Seiten der Regel ($L \leftarrow K \to R$). Die Transformation verläuft umgekehrt, sodass im ersten Schritt zunächst Elemente gelöscht und im zweiten Schritt hinzugefügt werden. Dies resultiert zwangsläufig in einem

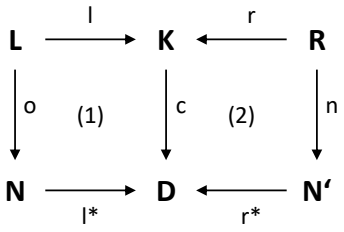

Abbildung 2.4: Regel $r = \langle L \rightarrow K \leftarrow R \rangle$

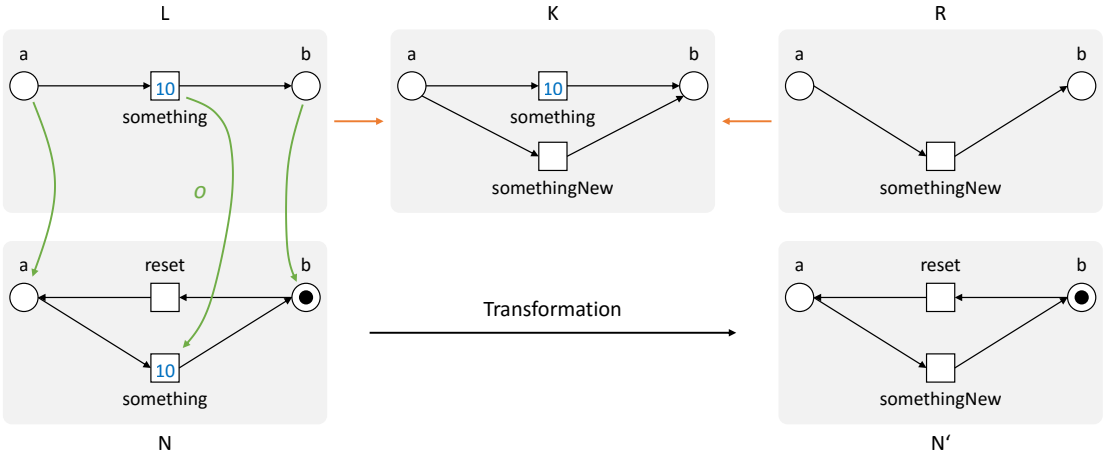

Abbildung 2.5: Regelbasierte Rekonfiguration

vergleichsweise „lückenhaften“ Zwischennetz D. Beim Cospan-Verfahren ist das Zwischennetz D noch ausgeprägter, da zunächst Elemente hinzugefügt und anschließend entfernt werden. Aus implementierungstechnischer Sicht bietet das Cospan-Verfahren dadurch eine komfortable Weiterverarbeitung. Es wurde bewiesen, dass das Cospan-Verfahren äquivalent zum klassischen Double-Pushout-Verfahren arbeitet [EHP09].

2.5 ReConNet

ReConNet (ReConfigurable Net) [PEOH12] ist ein Tool zur Modellierung und Simulation rekonfigurierbarer Petrinetze (s. Abbildung 2.6). Es wurde an der Hochschule für Angewandte Wissenschaften Hamburg entwickelt. Das Hauptaugenmerk liegt dabei auf der grafischen Modellierung von Netzen und Regeln, sowie der Simulation des Tokenspiels und regelbasierter Transformationen. Die grafische Benutzeroberfläche wurde auf diese Kernaufgaben ausgelegt, um ein intuitives Arbeiten zu gewährleisten. ReConNet ist vollständig in Java implementiert

und ist somit plattformunabhängig ausführbar. Für die Darstellung der Netze wird das *Java Universal Network/Graph Framework* (JUNG) verwendet [JUNa].

Modellierung

Die grafische Benutzeroberfläche ist in Abbildung 2.6 dargestellt. Über den File Tree [1] kann der Benutzer neue Netze und Regeln erstellen, sowie zuvor gespeicherte Netze und Regeln importieren. Die eigentliche Modellierung erfolgt über die beiden Hauptbereiche [3] und [4]. Die Selektion im File Tree bestimmt dabei, welches Netz bzw. welche Regel angezeigt werden soll. Die Radiobuttons [2] legen den Bearbeitungsmodus fest. Je nach Auswahl können Stellen, Transitionen oder Kanten hinzugefügt werden. Der Modus *Auswählen* wird verwendet, um Knoten bzw. Kanten zu bearbeiten. Nach Auswahl eines Knoten bzw. einer Kante können die jeweiligen Attribute im Bereich [6] editiert werden. Dies umfasst unter anderem Markierungen und Kapazitäten von Stellen, Labels und Erneuerungsfunktionen von Transitionen sowie Gewichtungen von Kanten. Weiterhin erlaubt dieser Modus das Verschieben von Knoten per Drag and Drop. Per Mausrad können die jeweiligen Darstellungen vergrößert bzw. verkleinert werden und mittels Drag and Drop in einen weißen Bereich können die Netze im Ganzen bewegt werden.

Simulation

Nach abgeschlossener Modellierung kann das rekonfigurierbare Petrinetz simuliert werden. Die Toolbox [5] wird für die Steuerung der Simulation verwendet. Der Benutzer hat mit den Buttons *Einmal schalten* und *Transformieren* die Möglichkeit, manuelle Simulationsschritte durchzuführen. Beim *einmaligen Schalten* wird eine beliebige aktive Transition geschaltet und die Folgemarkierung berechnet. Zur Übersicht werden aktive Transitionen dunkelgrau hervorgehoben. Beim *Transformieren* wird eine beliebige Regel auf das Netz angewendet. Voraussetzung hierfür ist, dass ein Morphismus existiert und die Klebebedingungen eingehalten sind. Das resultierende Netz wird berechnet und anschließend angezeigt.

Die Simulation kann ebenfalls automatisiert erfolgen. Der Benutzer kann dabei entscheiden, ob nur das Tokenspiel, nur die Transformationen oder beides simuliert werden soll. Hierbei kann entweder kontinuierlich oder mit fest vorgegebener Anzahl von Schritten simuliert werden. Weiterhin kann die Simulationsgeschwindigkeit über einen Regler dynamisch angepasst werden.

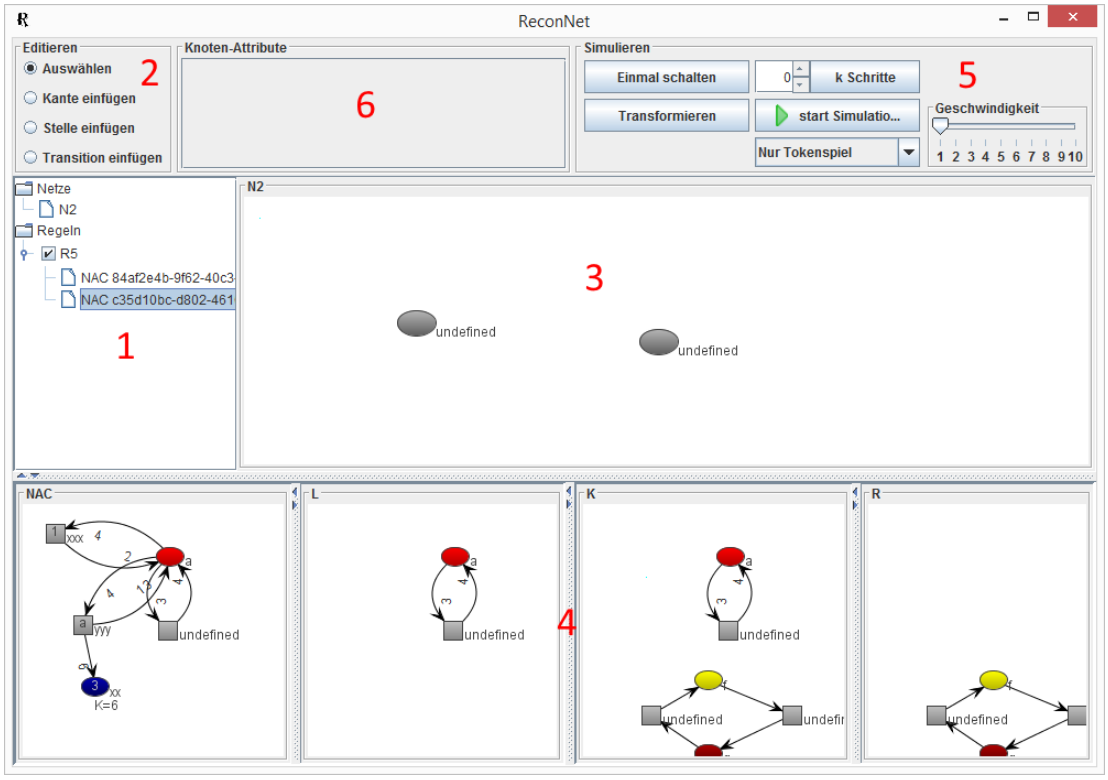

Abbildung 2.6: Grafische Benutzeroberfläche von ReConNet

2.6 ANTLR

ANTLR oder auch ***AN**other **T**ool For **L**anguage **R**ecognition* ist ein Parsergenerator und wurde federführend von Terence Parr an der *Purdue University* entwickelt [ANT]. Parallel zur Entwicklung von ANTLR lieferte Terence Parr diverse Beiträge zur Theorie des Parsens. Er führte unter anderem die in ANTLR genutzte Parsing-Strategie LL(*) ein [PF11]. ANTLR besitzt eine große Akzeptanz in der Industrie und wird beispielsweise von Firmen wie Twitter, Google, Oracle und Yahoo für diverse umfangreiche Anwendungen verwendet. Als Beispiel seien folgende Anwendungen genannt:

- Parsen von Suchanfragen bei Twitter
- Hibernate Query Language (HQL)
- C++ Parser der NetBeans IDE
- Oracle SQL Developer IDE

ANTLR stellt die notwendigen Tools für die Entwicklung von Anwendungen bereit, bei denen strukturierte Texte und Daten verarbeitet werden müssen. Auf Basis einer Grammatik, also einer formalen Definition einer Sprache, generiert ANTLR einen Parser. Dieser Parser kann anschließend verwendet werden, um Eingaben dieser Sprache zu verarbeiten und in einen Parsebaum zu überführen. ANTLR generiert hierbei ebenfalls die notwendigen Tools, um den Parsebaum traversieren zu können. Beim Traversieren des Parsebaums kann die anwendungsspezifische Logik durchgeführt werden, um das gewünschte Verhalten der Anwendung zu ermöglichen.

```
1  grammar Beispiel;
2
3  expr    : expr ASSIGN expr
4          | expr PLUS expr
5          | INT
6          ;
7
8  INT     : [0-9]+ ;
9  ASSIGN  : '=' ;
10 PLUS    : '+' ;
```

Listing 2.1: Beispiel.g4

In Listing 2.1 ist beispielhaft eine Grammatik für einfache mathematische Ausdrücke dargestellt. Das Parsen von Eingaben erfolgt üblicherweise in zwei Schritten. Zunächst erfolgt die lexikalische Analyse und anschließend das eigentliche Parsen [Par13, S.10]. Bei der lexikalischen

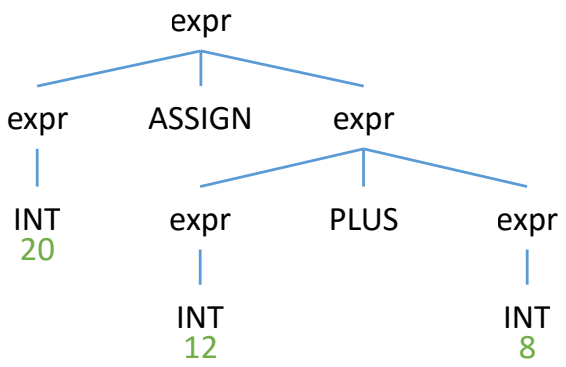

Abbildung 2.7: Parsebaum

Analyse werden die einzelnen Symbole der Eingabe zu sogenannten *Token* zusammengefasst. Als Beispiel soll hierbei der Eingabesatz *20 = 12 + 8* dienen. Die lexikalische Analyse überführt den Satz gemäß der Lexer-Regeln in die Token *INT*, *ASSIGN*, *INT PLUS* und *INT*.

Beim eigentlichen Parsen liest der Parser die einzelnen Token ein und überführt sie in einen Parsebaum (s. Abbildung 2.7). Die Struktur des Parsebaums ergibt sich dabei durch die in der Grammatik definierten Parser-Regeln. Diese strukturierte Form kann nun innerhalb der Anwendung verarbeitet werden, um anwendungsspezifische Logik durchzuführen.

3 Kontrollstrukturen

3.1 Einleitung

Unter dem Begriff *Kontrollstrukturen* werden diverse Techniken zusammengefasst, welche bei der Modellierung rekonfigurierbarer Petrinetze helfen, das Schalt- und Transformationsverhalten präziser zu steuern. Sie erweitern den Sprachschatz bei der Modellierung und steigern somit die Praxistauglichkeit rekonfigurierbarer Petrinetze [PH15, S.22]. Im Folgenden wird auf Kontrollstrukturen eingegangen, welche bei der Modellierung rekonfigurierbarer Petrinetze zur Verfügung stehen.

Dekorierte Petrinetze sind eine Erweiterung der klassischen Petrinetze und verfügen über zusätzliche Eigenschaften, die als Kontrollstrukturen herangezogen werden können [PH15, S.24]. Sie erweitern Petrinetze unter anderem um **Stellen- und Transitionsnamen**. Bei der Suche eines Morphismus $o : L \to N$ von der linken Seite einer Regel in das zu transformierende Petrinetz müssen die Namen von Stellen und Transitionen ebenfalls korrekt abgebildet werden. Auf diese Weise lässt sich die Vorbedingung einer Regel präziser ausformulieren, wodurch Transformationen feiner gesteuert werden können. Weiterhin können Transitionen in dekorierten Petrinetzen mit sogenannten **Transitionslabels** versehen werden. Diese müssen ebenfalls bei der Bildung eines Morphismus $o : L \to N$ berücksichtigt werden. Zudem ermög-

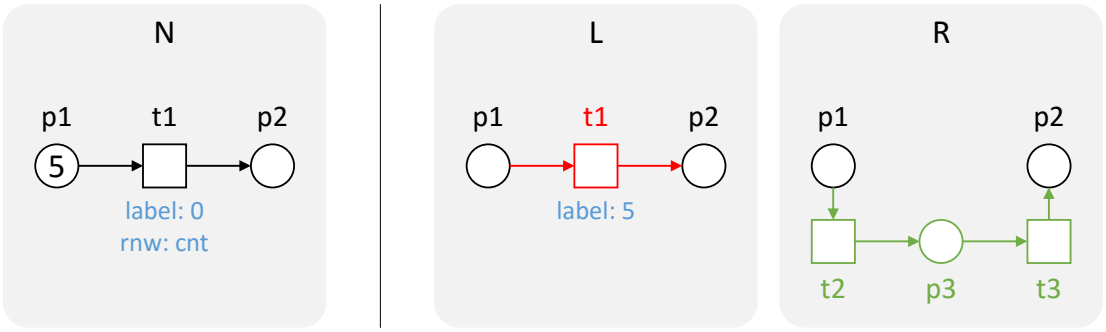

Abbildung 3.1: Wechselnde Transitionslabels als Kontrollstruktur

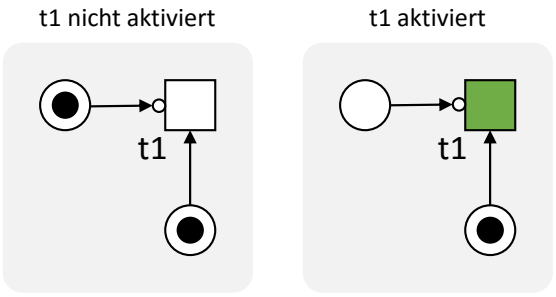

Abbildung 3.2: Inhibitor-Kante

lichen dekorierte Petrinetze die Verwendung sogenannter Label-Erneuerungsfunktionen. Eine Label-Erneuerungsfunktion weist einer Transition beim Schalten ein neues Transitionslabel zu. Die Label-Erneuerungsfunktion $t1 \mapsto \{(0,1),(1,2),(2,3),(3,4)...\}$ würde beispielsweise das Label der Transition $t1$ inkrementieren, wenn diese geschaltet wird. Mit dem Konzept der **wechselnden Transitionslabels** kann das Verhalten von Rekonfigurationen gesteuert werden. Es kann beispielsweise modelliert werden, dass eine Transformation erst dann stattfinden kann, sobald eine Transition fünf Mal geschaltet hat. Ein Beispiel hierfür ist in Abbildung 3.1 dargestellt. Nach fünfmaligem Schalten der Transition $t1$ kann ein Morphismus $o : L \to N$ gebildet werden. Das Netz wird dementsprechend gemäß der linken und rechten Seite der Regel transformiert. Der rot dargestellte Teil des Netzes wird hierbei durch den grün dargestellten Teil ersetzt.

Als eine weitere Kontrollstruktur können die sogenannten **Inhibitor-Kanten** verwendet werden [Pad15]. Inhibitor-Kanten steuern das Schaltverhalten von Transitionen. Eine Transition mit einer eingehenden Inhibitor-Kante kann nur dann aktiviert sein, wenn die vorausgehende Stelle dieser Inhibitor-Kante kein Token aufweist (s. Abbildung 3.2). Die Erweiterung klassischer S/T-Netze um Inhibitor-Kanten macht den Formalismus Turing-Vollständig und erlaubt somit die Modellierung von Systemen, die mit klassischen S/T-Netzen nicht modelliert werden können [Age74].

Weiterhin können Prioritäten als mögliche Kontrollstruktur herangezogen werden. Zum einen können **Prioritäten von Transitionen** das Schaltverhalten beeinflussen. Hierbei wird eine partielle Ordnungsrelation $\leq$ über der Menge von Transitionen definiert. Eine Transition kann demnach schalten, wenn diese aktiviert ist und unter allen anderen aktivierten Transitionen keine mit einer höheren Priorität existiert. Zum anderen können **Prioritäten von Regeln** das Transformationsverhalten steuern. Hierfür wird eine partielle Ordnungsrelation $\leq$ über der

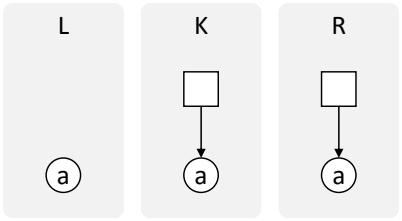

Abbildung 3.3: Regel *addTransition*

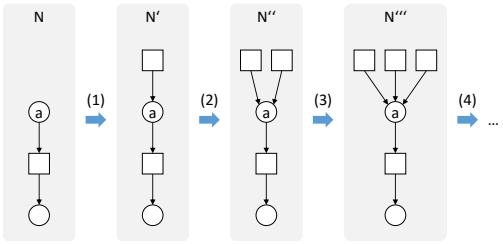

Abbildung 3.4: Beispiel [Rei08, vgl. S.8]

Menge aller Regeln definiert. Eine Regel ist dann anwendbar, wenn ein Morphismus $o : L \to N$ existiert und keine andere anwendbare Regel mit einer höheren Priorität existiert [PH15, S.23].

Neben den oben genannten Kontrollstrukturen kann im Kontext zeitabhängiger Petrinetze, wie *time Petri nets*, *deterministic timed Petri nets* oder *timed colored Petri nets*, das **zeitabhängige Schaltverhalten** von Transitionen als Kontrollstruktur herangezogen werden [PH15, S.25].

3.2 Negative Anwendungsbedingungen

3.2.1 Motivation

Die primäre Voraussetzung für die Anwendung einer Regel ist ein gültiger Netzmorphismus von L nach N. Es gibt in dieser Form also nur die Möglichkeit Strukturen festzulegen, wann eine Regel ausgeführt werden darf. Dies ist für manche Problemstellungen jedoch nicht aussagekräftig genug.

Die Abbildungen 3.3 und 3.4 verdeutlichen dieses Problem. Die Regel *addTransition* setzt durch L nur eine Stelle voraus. Bei der Transformation wird dieser Stelle eine Transition vorgeschaltet. Bezogen auf das Beispiel kann dabei immer ein Netzmorphismus von L nach N gebildet werden, sodass die Regel permanent zur Rekonfiguration führen kann (s. Abbildung 3.4). Für solche Problemstellungen werden zusätzliche Kontrollstrukturen benötigt, welche die unerwünschte Anwendung von Regeln in bestimmten Situationen verhindern. In manchen Fällen können mit Dekorationen gewisse Logiken ausgedrückt werden, um die Anwendung von Regeln zu verhindern [Rei08, S.8]. Dies ist jedoch nicht immer praktikabel, da man hierfür eventuell das eigentliche Netz präparieren muss, um solche Logiken gewährleisten zu können.

3.2.2 Prinzip

Negative Anwendungsbedingungen (engl. negative application condition, *NAC*) sind eine Kontrollstruktur, die das Ausführen von Regeln in bestimmten Situationen verhindern. Die in Abschnitt 2.4 geführte Definition einer Regel wird hierbei um die Menge an negativen Anwendungsbedingungen erweitert $r = (NACs, L \to K \leftarrow R)$.

Definition 5 (Negative Anwendungsbedingung)**.** *Eine (linke) negative Anwendungsbedingung einer Regel $r = (L \to K \leftarrow R)$ hat die Form $NAC(p)$, wobei $p : L \to NAC$ ein Morphismus ist.*
Ein Morphismus $o : L \to N$ erfüllt $NAC(p)$, geschrieben als $o \models NAC(p)$, wenn kein Morphismus $q : NAC \to N$ existiert mit $q \circ p = o$ (s. Abbildung 3.5).

Ein negative Anwendungsbedingung wird ebenfalls als Petrinetz repräsentiert. Abbildung 3.5 zeigt die bei einer Transformation vorliegende Beziehung zwischen Regel, Netz und den negativen Anwendungsbedingungen. Eine Regel darf demnach nur dann angewandt werden, wenn ein Morphismus von L nach N gebildet werden kann und dieser Morphismus die gegebenen negativen Anwendungsbedingungen erfüllt. Dies ist der Fall, wenn kein injektiver Morphismus $q : NAC \to N$ existiert mit $q \circ p = o$. Da eine Regel mehrere NACs beinhalten kann, muss dieses für jede $NAC \in NACs$ gelten.

Das eingangs angeführte Problem mit der repetitiven Anwendung einer Regel kann auf diese Weise mittels einer negativen Anwendungsbedingung gelöst werden. Die in Abbildung 3.3 dargestellte Regel müsste demnach um eine NAC erweitert werden. Die NAC könnte beispielsweise äquivalent zur rechten Seite R der Regel sein. Die in Abbildung 3.4 dargestellte, erste Transformation $\xrightarrow{(1)}$ kann weiterhin durchgeführt werden, weil kein injektiver Morphismus $q : NAC \to N_1$ gebildet werden kann. Die Stelle a bekommt demnach eine Transition vorgeschaltet. Die zweite Transformation $\xrightarrow{(2)}$ darf jedoch nicht mehr ausgeführt werden, weil durch die neue Transition ein injektiver Morphismus $q : NAC \to N_2$ gebildet werden kann.

Regeln mit negativen Anwendungsbedingungen ermöglichen implizit eine Formulierung logischer Ausdrücke. Sei r eine Regel $\langle L \to K \leftarrow R \rangle$ und seien $NAC_{1,2}$ zwei negative Anwendungsbedingungen. Gibt es zwei separate Zuordnungen $(\{NAC_1\}, r)$ und $(\{NAC_2\}, r)$, muss nur eine der beiden NACs erfüllt sein, damit eine Transformation durch Regel r möglich ist (logisches $\vee$). Gibt es nur eine Zuordnung beider NACs zur Regel $(\{NAC_1, NAC_2\}, r)$ müssen dementsprechend beide NACs erfüllt sein, damit eine Transformation möglich ist (logisches $\wedge$).

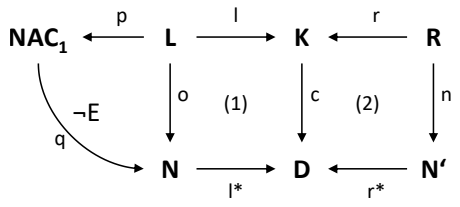

Abbildung 3.5: Regel mit einer negativen Anwendungsbedingung

3.2.3 Ergebnisse

In [Rei08] wurde bewiesen, dass Stellen/Transitions-Netze eine *weak adhesive HLR category with NACs* bilden. Dadurch können wichtige Ergebnisse von *weak adhesive HLR category with NACs* auf Stellen/Transitions-Netze übertragen werden. Dies umfasst unter anderem Ergebnisse zur lokalen Church-Rosser Eigenschaft, Parallelität, Konflikte und kritischen Paaren, sowie der Nebenläufigkeit von Transformationen.

Lokale Church-Rosser Eigenschaft und Parallelität

Im Kontext von Transformationen mit negativen Anwendungsbedingungen besagt die lokale Church-Rosser Eigenschaft, dass die Anwendung zweier Regeln in beliebiger Reihenfolge zum selben Ergebnis führt, genau dann wenn beide Transformationen parallel unabhängig voneinander sind.

Um herauszufinden ob zwei Regeln parallel unabhängig voneinander sind, muss überprüft werden, ob eine der Regeln ggf. Strukturen löscht, die von der anderen Regel vorausgesetzt werden bzw. Strukturen hinzufügt, die von der anderen Regel durch eine negative Anwendungsbedingung ausgeschlossen werden. Mehrere parallel unabhängige Transformationen können zu einer parallelen Transformation zusammengefasst werden [RPL+08, S.10].

Konflikte und kritische Paare

Zwei Transformationen stehen in Konflikt, wenn sie nicht parallel unabhängig voneinander sind. Dies kann dann eintreten, wenn eine Transformation Strukturen erzeugt bzw. löscht, sodass die Ausführung der zweiten Transformation verhindert wird. Eine solche Situation wird mittels sogenanntem kritischen Paar im minimalen Kontext beschrieben. Kritische Paare sind vollständig, d.h. für jeden möglichen Konflikt gibt es ein kritisches Paar, welches diesen Konflikt beschreibt [RPL+08, S.10].

Nebenläufigkeit

Bei sequentieller Unabhängigkeit, also wenn zwei Transformationen in beliebiger Reihenfolge zum gleichen Ergebnis führen, sind diese Transformationen auch parallel unabhängig. Wie im obigen Abschnitt beschrieben, können diese Transformationen zu einer Transformation zusammengefasst werden. Herrschen jedoch sequentielle Abhängigkeiten in einer Sequenz von Transformationen, kann keine parallele Transformation konstruiert werden. In diesem Fall kann jedoch eine *nebenläufige Regel* mit NACs konstruiert werden, welche mit einer Transformation zum selben Ergebnis führt, wie die sequentielle Anwendung der Transformationen. Das *Concurrency Theorem* besagt, dass eine äquivalente nebenläufige Regel mit NACs zum gleichen Ergebnis führt, genau dann wenn die zugrundeliegende Sequenz von Regeln mit NACs anwendbar ist [RPL+08, S.10-11].

3.3 Transformationseinheiten

3.3.1 Einleitung

Graph-Transformationseinheiten wurden als elementare Einheit zur Spezifikation und Programmierung auf Basis von Graph-Transformationen vorgestellt [AEH+99]. Netz-Transformationseinheiten (engl. *net transformation units*) sind die Adaption von Graph-Transformationseinheiten im Kontext rekonfigurierbarer Petrinetze [PH15, S.23]. Sie bilden eine weitere Form der Kontrollstrukturen.

Die Motivation für Transformationseinheiten liegt in der Beherrschbarkeit komplexer Systeme [AEH+99, S.36]. Die Modellierung realer Probleme mit Hilfe von rekonfigurierbaren Petrinetzen führt häufig zu sehr komplexen und unüberschaubaren Systemen mit riesigen Netzen und einer hohen Anzahl von Regeln. Je größer das jeweilige Netz, desto größer werden auch die Regeln, um komplexe Prozesse in diesem Netz abbilden zu können. Um der wachsenden Komplexität Herr zu werden ist es sinnvoll, Teilprozesse zu identifizieren, welche anschließend zu großen Prozessen zusammengefasst werden können. Durch die Identifizierung von Teilprozessen wird weiterhin die Wiederverwendbarkeit von Regeln gefördert. Transformationseinheiten ermöglichen diesen Vorgang und helfen somit bei der Modellierung komplexer Systeme.

3.3.2 Prinzip

Mehrere Teilprozesse können in einer Transformationseinheit zu einem großen Prozess gebündelt werden. Da die Teilprozesse einer solchen Einheit häufig in geregelter Weise ausgeführt

werden müssen, ist ein Hilfsmittel notwendig, welches den Ablauf einer Transformationseinheit klar definiert. Hierfür wird der sogenannte *Kontrollausdruck* verwendet. Der Kontrollausdruck ist ein syntaktisches Hilfsmittel mit dem der Ablaufplan einer Transformationseinheit definiert werden kann. Bei der Definition stehen hierbei diverse Operatoren zur Verfügung, mit denen der Ablaufplan gestaltet werden kann.

Definition 6 (Transformationseinheit)**.** *Eine Transformationseinheit ist definiert als Tupel* $NT = (\mathcal{R}, nm : \mathcal{R} \rightarrow Names, \mathcal{C})$*, wobei* $\mathcal{R}$ *die Menge an Regeln ist,* nm *eine Abbildung der Regeln auf Namen sowie dem Kontrollausdruck* $\mathcal{C}$*, welcher wie folgt rekursiv definiert ist:*

1. $r \in nm(R) \rightarrow r \in \mathcal{C}$
2. $C_1 \in \mathcal{C} \wedge C_2 \in \mathcal{C} \rightarrow C_1; C_2 \in \mathcal{C}$
3. $C_1 \in \mathcal{C} \wedge C_2 \in \mathcal{C} \rightarrow C_1 | C_2 \in \mathcal{C}$
4. $C_1 \in \mathcal{C} \rightarrow C_1! \in \mathcal{C}$
5. $C_1 \in \mathcal{C} \rightarrow C_1^* \in \mathcal{C}$

Punkt 1 besagt hierbei, dass jeder Name einer Regel ein atomarer Kontrollausdruck ist. In diesem Fall wird die jeweilige Regel auf das Netz angewendet.

Punkt 2 definiert die Sequenz zweier Kontrollausdrücke. Wenn C_1 und C_2 Kontrollausdrücke sind, so ist auch $C_1; C_2$ ein gültiger Kontrollausdruck. Bei der Sequenz werden die jeweiligen Kontrollausdrücke nacheinander, von links nach rechts, durchgeführt.

Punkt 3 definiert die Alternative. Wenn C_1 und C_2 Kontrollausdrücke sind, so ist auch $C_1 | C_2$ ein gültiger Kontrollausdruck. Bei der Alternative wird nur einer der beiden Kontrollausdrücke durchgeführt. Die Wahl der durchzuführenden Alternative erfolgt hierbei zufällig.

Punkt 4 definiert den unären *asLongAsPossible*-Operator. Wenn C_1 ein Kontrollausdruck ist, so ist auch $C_1!$ ein gültiger Kontrollausdruck. Der *asLongAsPossible*-Operator ! sorgt dafür, dass der jeweilige Kontrollausdruck C_1 so oft wie möglich durchgeführt wird. Ist der Kontrollausdruck C_1 beispielsweise eine einfache Regel, so wird diese so oft angewendet, bis die Regel aus etwaigen Gründen nicht mehr angewendet werden kann. Dies ist beispielsweise der Fall, wenn kein Morphismus $o : L \rightarrow N$ mehr gebildet werden kann, oder eine negative Anwendungsbedingung die Anwendung der Regel verhindert. Theoretisch kann der *asLongAsPossible*-Operator auch zu einer unendlichen Anwendung des jeweiligen Kontrollausdrucks

führen. Dies wäre beispielsweise der Fall, wenn der Kontrollausdruck nur aus einer Regel besteht, welche keine Vorbedingung und weiterhin keine negative Anwendungsbedingung besitzt. Die Anwendung der Regel wäre demnach immer möglich. Diese Eigenschaft des Operators ist bei der Implementierung von Transformationseinheiten zu berücksichtigen. Ein anderer Extremfall liegt vor, wenn der Kontrollausdruck kein einziges Mal angewendet wird. Dies hat keine negativen Auswirkungen auf die Transformationseinheit. Die Durchführung des Ausdrucks C_1 beim *asLongAsPossible*-Operator ist somit optional.

Der Kontrollausdruck C_1 kann hierbei beliebig komplex sein. Dabei ist zu beachten, dass der Kontrollausdruck bei einer Iteration immer als Einheit durchgeführt werden muss. Besteht der Kontrollausdruck C_1 beispielsweise aus einer Sequenz von Regeln, müssen bei einer *asLongAsPossible*-Iteration immer alle Regeln der Sequenz angewendet werden.

Punkt 5 definiert den unären *randomNumberOfTimes*-Operator. Wenn C_1 ein Kontrollausdruck ist, so ist auch C_1^* ein gültiger Kontrollausdruck. Dieser Operator wendet den jeweiligen Kontrollausdruck beliebig häufig an, maximal jedoch so häufig, wie es möglich ist. Das Verhalten des Operators ähnelt dem Verhalten des *asLongAsPossible*-Operators mit der Ausnahme, dass der *randomNumberOfTimes*-Operator auch weniger Iterationen durchführen kann. Die Anzahl an Iterationen wird hierbei zufällig aus einer Zahl zwischen 0 und x gewählt. Die Durchführung des Ausdrucks C_1 beim *randomNumberOfTimes*-Operator ist dadurch ebenfalls optional. Analog zum *asLongAsPossible*-Operator muss bei einer Iteration der Ausdruck C_1 immer vollständig ausgeführt werden.

Beispiele

Die rekursive Definition ermöglicht es, beliebig geschachtelte Kontrollausdrücke zu formen, um komplexe Transformationseinheiten zu modellieren.

$\mathcal{C} = Regel_A; Regel_B; Regel_C^*$
Führt zunächst $Regel_A$ und anschließend $Regel_B$ aus. Danach wird beliebig häufig $Regel_C$ ausgeführt.

$\mathcal{C} = Regel_A!; Regel_B | Regel_C$
Führt zunächst $Regel_A$ so oft wie möglich aus. Anschließend wird entweder $Regel_B$ oder $Regel_C$ ausgeführt.

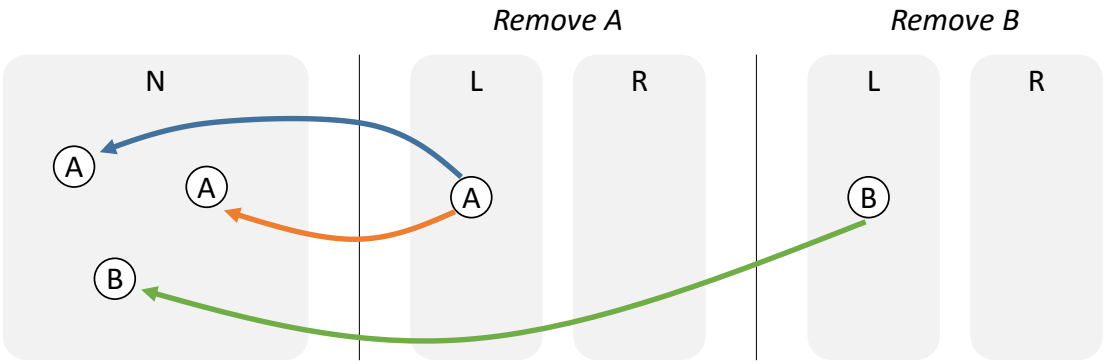

Abbildung 3.6: Nichtdeterminismus bei Anwendung von Regeln

3.3.3 Nichtdeterminismus bei Transformationen

Die Transformation rekonfigurierbarer Petrinetze erfolgt unter Umständen nichtdeterministisch [AEH+99, S.37]. Zum einen haben Regeln immer einen lokalen Charakter und können eventuell an unterschiedlichen Stellen im Netz durchgeführt werden. Dies ist der Fall wenn gleich mehrere Morphismen von der linken Seite einer Regel in das zu transformierende Netz gefunden werden, unter Berücksichtigung der negativen Anwendungsbedingungen. Der jeweilige Morphismus $o : L \to N$ wird hierbei nichtdeterministisch gewählt (s. Abbildung 3.6, Regel *RemoveA*). Zum anderen ergibt sich ein Nichtdeterminismus durch die Tatsache, dass es möglicherweise mehrere Regeln gibt, die auf das vorliegende Petrinetz angewendet werden können. Die Auswahl der durchzuführenden Regel erfolgt hierbei nichtdeterministisch (s. Abbildung 3.6, Regel *RemoveA* und *RemoveB*).

Mit Hilfe des Kontrollausdrucks kann der Nichtdeterminismus von Transformationen teilweise reduziert werden. Zum einen erlaubt ein Kontrollausdruck die Bestimmung der Reihenfolge von Transformationen. Die Tatsache, dass gleichzeitig mehrere Regeln für eine mögliche Transformation zur Auswahl stehen, wird hiermit ausgeschlossen. Zum anderen können mit Hilfe von Transformationseinheiten Situationen im Netz gezielter hergestellt werden, sodass die Wahrscheinlichkeit für mehrfache Morphismen $o : L \to N$ reduziert wird.

3.3.4 Strukturierte Transformationseinheiten

Der Kontrollausdruck einer Transformationseinheit ermöglicht die Definition des Ablaufs durch die Angabe einzelner Regeln. Bei komplexen Systemen mit vielen Regeln können die Kontrollausdrücke ebenfalls sehr komplex werden. Eine zusätzliche Modularisierung auf Ebene der

Transformationseinheiten kann hierbei helfen, die Komplexität zu beherrschen. *Strukturierte Transformationseinheiten* ermöglichen diesen Ansatz. Bei einer strukturierten Transformationseinheit können im Kontrollausdruck ebenfalls andere Transformationseinheiten referenziert werden. Man spricht hierbei auch von einer *Transformationseinheit mit Import* [Lud15, S.19].

$$\mathcal{C} = Regel_A; Transformationseinheit_X; Regel_B | Regel_C$$

Bei strukturierten Transformationseinheiten muss jedoch beachtet werden, dass keine zyklischen Abhängigkeiten zwischen Transformationseinheiten existieren. Eine Transformationseinheit darf sich dementsprechend weder direkt, noch indirekt selbst importieren. Um zyklische Abhängigkeiten zu verhindern, wird jeder Transformationseinheit ein sogenanntes *Import Level* zugeordnet. Jede Transformationseinheit darf hierbei nur Transformationseinheiten mit einem niedrigeren Import Level importieren. Eine Transformationseinheit mit einem Import Level von 0 verfügt über keine importierten Transformationseinheiten. Auf diese Weise ist sichergestellt, dass zyklische Abhängigkeiten zwischen Transformationseinheiten verhindert werden.

3.4 Kontrollstrukturen in ReConNet

Um rekonfigurierbare Petrinetze in der Praxis einsetzen zu können sind Tools notwendig. Das Tool ReConNet wurde bereits in Abschnitt 2.5 angeführt. ReConNet erlaubt die grafische Modellierung und Simulation rekonfigurierbarer Petrinetze. Das Schalt- und Transformationsverhalten kann hierbei im Verbund simuliert werden, was essenziell für die praktische Anwendung rekonfigurierbarer Petrinetze ist. Um eine präzise und intuitive Modellierung rekonfigurierbarer Petrinetze zu gewährleisten, muss das Tool eine Vielzahl an Kontrollstrukturen unterstützen. In Tabelle 3.1 ist die Verfügbarkeit der einzelnen Kontrollstrukturen in ReConNet dargestellt. Wie der Tabelle zu entnehmen ist verfügt ReConNet bereits über diverse Kontrollstrukturen, welche in diesem Kapitel erläutert wurden. Die Anwendung von Regeln kann demnach bereits über Stellen- und Transitionsnamen sowie Transitionslabels präziser gesteuert werden. Weiterhin werden wechselnde Transitionslabels mit den Labelerneuerungsfunktionen *id*, *count* und *toggle* unterstützt. Die negativen Anwendungsbedingungen wurden im Rahmen einer zuvor durchgeführten Projektarbeit ebenfalls implementiert.

Im Rahmen dieser Arbeit wurden Transformationseinheiten als weitere Kontrollstruktur in ReConNet integriert. Bisher konnten komplexe rekonfigurierbare Petrinetze nur umständlich

Kontrollstruktur	**Implementiert in ReConNet**
Labels, names	Ja
Changing transition labels	Ja
Priorities of transitions	Nein
Priorities of rules	Nein
Inhibitor arcs	Nein
Timed token	Nein
Negative application conditions	Ja
Nested application conditions	Nein
Net transformation units	Ja

Tabelle 3.1: Kontrollstrukturen in ReConNet

modelliert werden. Komplexe Transformationen konnten hierbei entweder nur feingranular oder monolithisch abgebildet werden. Die feingranulare Abbildung von komplexen Transformationen durch viele kleine Regeln ist jedoch nicht praxistauglich, da diese händisch nacheinander durchgeführt werden müssen. Die monolithische Abbildung einer komplexen Transformation erfolgt im Extremfall mit einer einzigen großen Regel. Dies führt implizit zu einer schwer nachvollziehbaren Komplexität. Die gesamte Auswirkung solch einer Regel kann dabei nur nach intensiver Auseinandersetzung nachvollzogen werden. Weiterhin ist dieses Vorgehen sehr starr und muss exakt auf eine bestimmte Situation im Netz ausgelegt sein.

Mit Hilfe der implementierten Transformationseinheiten können viele kleine Regeln zu einer komplexen Transformation gebündelt werden. Man bewahrt hierbei die Kernaufgaben und Wiederverwendbarkeit von Regeln und hat einen Mechanismus der die einzelnen Regeln automatisiert auf das Netz anwendet. Mit Hilfe der verfügbaren Operatoren können Transformationen flexibler modelliert werden und müssen nicht mehr exakt auf bestimmte Situationen im Netz ausgelegt sein. Die Erweiterung des Tools ReConNet um Transformationseinheiten ist daher eine große Bereicherung für die Modellierung und praktische Anwendung rekonfigurierbarer Petrinetze.

4 Analyse und Konzeption

4.1 Architektur von ReConNet

ReConNet hat eine komponentenbasierte Architektur und verfügt derzeit über fünf Komponenten (s. Abbildung 4.1).

Die **GUI-Komponente** ist verantwortlich für die grafische Benutzeroberfläche. Dies Umfasst die Anzeige von Netzen und Regeln, sowie die Verarbeitung von Nutzereingaben zur Bearbeitung von Netzen und Regeln und zur Simulation. Die Komponente greift über einen *EngineAdapter* auf Methoden der *Engine*-Komponente zu. Der *EngineAdapter* stellt hierfür den Zugriff auf Instanzen von Singleton-Klassen bereit, welche jeweils die Schnittstellen *IPetrinetManipulation*, *IRuleManipulation* und *ISimulation* implementieren.

Die **Engine-Komponente** ist die zentrale Komponente und stellt Funktionen zum Erstellen, Bearbeiten und Löschen von Netzen und Regeln zur Verfügung. Die jeweiligen Aufgaben werden an die Subkomponenten *PetrinetManipulation* und *RuleManipulation* delegiert. Die Subkomponente *SessionManager* dient als Speicher der aktuell vorliegenden Modelldaten aller Netze und Regeln und speichert zusätzliche Daten für die grafische Darstellung wie z.B. Knotenpositionen, -Farben, -Größen sowie weitere Layoutparameter. Des Weiteren stellt die Subkomponente *SimulationHandler* die notwendigen Funktionen zur Simulation des rekonfigurierbaren Petrinetzes zur Verfügung. Wenn bei der Simulation Netztransformationen stattfinden, wird hierfür die Transformations-Komponente herangezogen.

Die **Transformations-Komponente** implementiert die Logik für die Transformation von Petrinetzen. Dies beinhaltet das bilden von Netzmorphismen bei Einhaltung der Klebebedingungen sowie die anschließende Transformation zum resultierenden Netz.

Die **Persistenz-Komponente** implementiert die Logik zum Exportieren und Importieren von Netzen und Regeln. Hier liegen analog zu den Modelldaten-Klassen jeweils XML-annotierte

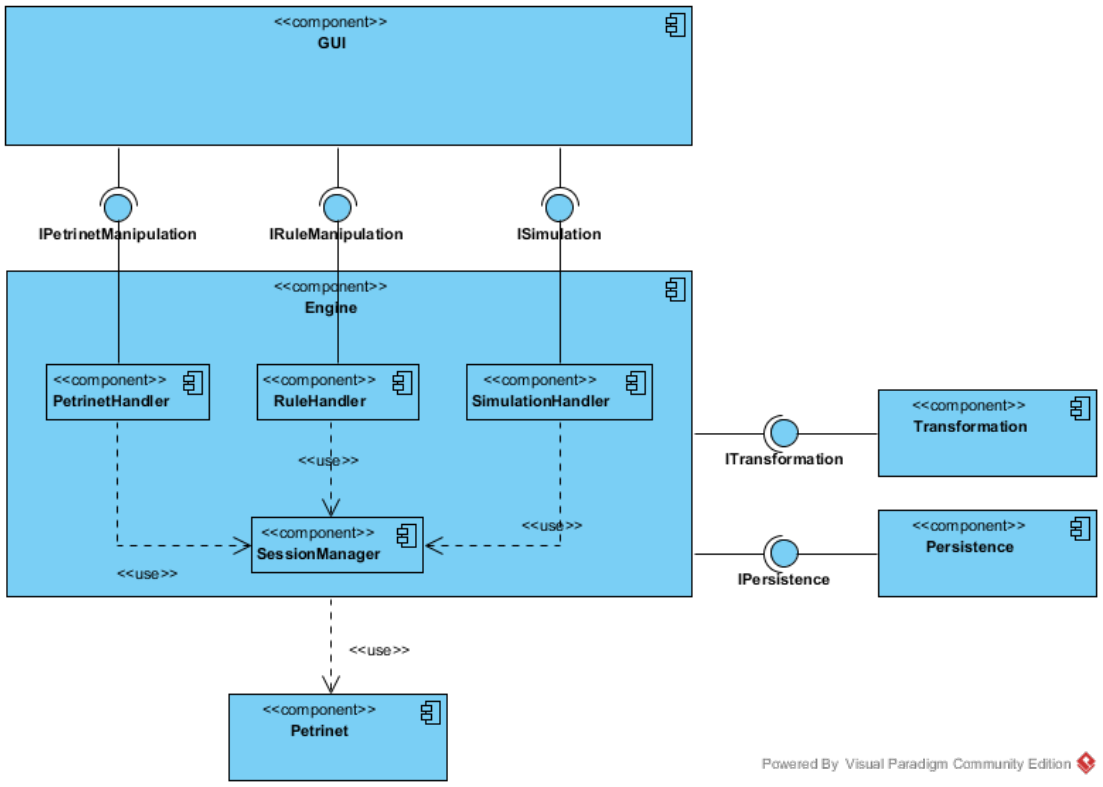

Abbildung 4.1: Architektur von ReConNet

Klassen bereit, die zur Serialisierung und Deserialisierung verwendet werden. Ein Converter wird verwendet, um zwischen beiden Darstellungsformen zu konvertieren. Um die Interoperabilität mit anderen Tools zu gewährleisten, wird für Petrinetze das XML Schema *Petri Net Markup Language (PNML)* verwendet [WK03]. Da Regeln nicht explizit im PNML Schema spezifiziert sind, werden diese entsprechend der Spezifikation für Petrinetze gespeichert. Die drei Seiten einer Regel werden hierbei jeweils als Petrinetz in einer gemeinsamen Datei gespeichert. Die Morphismen $\langle L \rightarrow K \leftarrow R \rangle$ einer Regel werden über logische id Attribute abgebildet. Beim Laden einer Regel werden die Morphismen anhand dieser id Attribute wiederhergestellt.

4.2 Erweiterungspunkte

Für die Erweiterung von ReConNet um Transformationseinheiten wurde zunächst erörtert, wie diese in der vorhandenen Architektur abgebildet werden sollen. Transformationseinheiten sind eigenständige Artefakte im Kontext von ReConNet und stehen somit auf der

Ebene von Petrinetzen und Regeln. Der Benutzer soll dabei in der Lage sein, Transformationseinheiten zu erstellen, bearbeiten, speichern und zu laden. Aus diesem Grund muss die *Engine*-Komponente um eine weitere Subkomponente *TransformationUnitManipulation* erweitert werden. Die Funktionen der Komponente sollen der grafischen Oberfläche über eine Schnittstelle *ITransformationUnitManipulation* zur Verfügung gestellt werden. Neben der neuen Subkomponente müssen bisherige Komponenten um Funktionen erweitert werden, um das Arbeiten mit Transformationseinheiten zu gewährleisten. Dazu zählt unter anderem die *Session-Komponente*, welche die aktuellen Daten von Transformationseinheiten verwaltet. Weiterhin muss die *Persistenz-Komponente* erweitert werden, sodass Transformationseinheiten auf das Dateisystem gespeichert und davon geladen werden können. Zudem muss die *GUI-Komponente* erweitert werden, um dem Benutzer die Arbeit mit Transformationseinheiten zu ermöglichen.

4.3 Transformationseinheiten in ReConNet

Gemäß der Definition 6 ist eine Transformationseinheit definiert als Tupel $NT = (\mathcal{R}, nm : \mathcal{R} \rightarrow Names, \mathcal{C})$, wobei $\mathcal{R}$ die Menge an Regeln ist, nm eine Abbildung der Regeln auf Namen sowie dem Kontrollausdruck $\mathcal{C}$, welcher wie folgt rekursiv definiert ist:

1. $r \in nm(R) \rightarrow r \in \mathcal{C}$
2. $C_1 \in \mathcal{C} \land C_2 \in \mathcal{C} \rightarrow C_1; C_2 \in \mathcal{C}$
3. $C_1 \in \mathcal{C} \land C_2 \in \mathcal{C} \rightarrow C_1 | C_2 \in \mathcal{C}$
4. $C_1 \in \mathcal{C} \rightarrow C_1! \in \mathcal{C}$
5. $C_1 \in \mathcal{C} \rightarrow C_1^* \in \mathcal{C}$

Im Kontext von ReConNet ist hierbei primär der Ausdruck $\mathcal{C}$ des Tupels zu betrachten. Durch die rekursive Definition können Kontrollausdrücke beliebig verschachtelt werden, um beliebig komplexe Transformationseinheiten zu bilden. Der vom Benutzer eingegebene Kontrollausdruck muss hierbei in eine Struktur überführt werden, die letztendlich die Abfolge der anzuwendenden Transformationen regelt. Es muss also definiert sein, wie beispielsweise der Ausdruck

$\mathcal{C} = Regel_A \mid (Regel_X ; Regel_Z) ; Regel_B \mid Regel_T!$

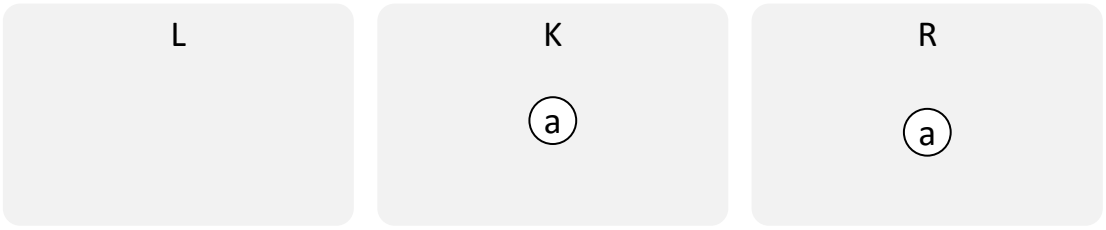

Abbildung 4.2: Regel ohne Vorbedingung

angewendet wird. Intuitiv wird der Ausdruck von links nach rechts abgearbeitet, wobei die einzelnen Operatoren einer Rangfolge unterliegen sollten, welche wie folgt definiert wird (von hoch nach niedrig):

1. $(\mathcal{C})$
2. $\mathcal{C}!$
3. $\mathcal{C}^*$
4. $\mathcal{C}|\mathcal{C}$
5. $\mathcal{C};\mathcal{C}$

Durch die Operatorrangfolge ist die Ausführungsreihenfolge von Teilausdrücken klar definiert.

4.3.1 Nichtdeterminismus und Terminationsverhalten

Die letztendliche Abfolge von Transformationen ist dabei nichtdeterministisch aufgrund der Operatoren | und *. Eine Transformationseinheit kann demnach bei gleichen Anfangsbedingungen zu unterschiedlichen Ergebnissen führen. Weiterhin ist das Terminationsverhalten der Operatoren * und **!** zu beachten. Die Verwendung des *asLongAsPossible*-Operators würde in der Praxis zu endlosen Transformationen führen können und das Programm zwangsläufig zum Absturz bringen. In Abbildung 4.2 ist eine Regel ohne Vorbedingung dargestellt. Eine Transformationseinheit mit dem Kontrollausdruck $Regel!$ würde aufgrund der fehlenden Vorbedingung der Regel zur endlosen Anwendung dieser Regel führen. Würde der Kontrollausdruck $Regel^*$ verwendet werden, wäre ebenfalls ungewiss, wann die Transformationseinheit terminiert. Aus praktischen Gründen muss das Terminationsverhalten beider Operatoren dementsprechend gesteuert werden können. Der Benutzer hat demnach die Möglichkeit, die maximale Anzahl an Transformationen bei den Operatoren ! und * zu bestimmen.

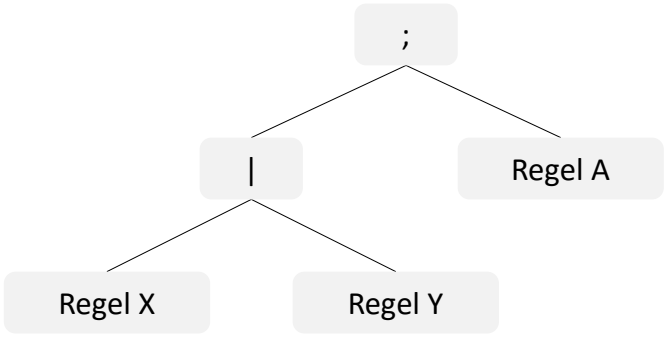

Abbildung 4.3: Parsebaum

4.3.2 Kontrollausdruck als Baum

Um intuitiv mit dem Kontrollausdruck arbeiten zu können, ist es naheliegend, den Kontrollausdruck in eine Baumstruktur zu überführen. Das naheliegende Verfahren hierfür ist die Entwicklung einer Grammatik, mit dessen Hilfe der Kontrollausdruck strukturiert eingelesen und in einen Parsebaum überführt werden kann. Mit Hilfe der Grammatik lässt sich ebenfalls die Operatorrangfolge spezifizieren, sodass der resultierende Parsebaum automatisch die Rangfolgen der Operatoren berücksichtigt. Der Kontrollausdruck $\mathcal{C} = Regel_X|Regel_Y; Regel_A$ muss demnach in den Parsebaum überführt werden, welcher in Abbildung 4.3 dargestellt ist. Der Parsebaum kann anschließend traversiert werden um die jeweiligen Regeln der Transformationseinheit anzuwenden.

4.3.3 Algorithmus

Der Parsebaum kann nun für die Durchführung der Transformationseinheit herangezogen werden. Beim Traversieren des Parsebaums wird in Abhängigkeit des Knotentyps ein bestimmtes Verhalten ausgeführt, um die Transformationseinheit entsprechend ihrer Definition durchzuführen. Abbildung 4.4 stellt beispielhaft den Parsebaum für den Kontrollausdruck $\mathcal{C} = (Regel_A; Regel_B)!; Regel_C^*; Regel_D|Regel_E; Regel_F$ dar. Nachfolgend wird das Verhalten des Algorithmus für die unterschiedlichen Knotentypen erläutert.

Regel-Knoten

Befindet sich der Algorithmus in einem Regel-Knoten, wird lediglich die jeweilige Regel angewendet (s. Abbildung 4.4, grüne Knoten).

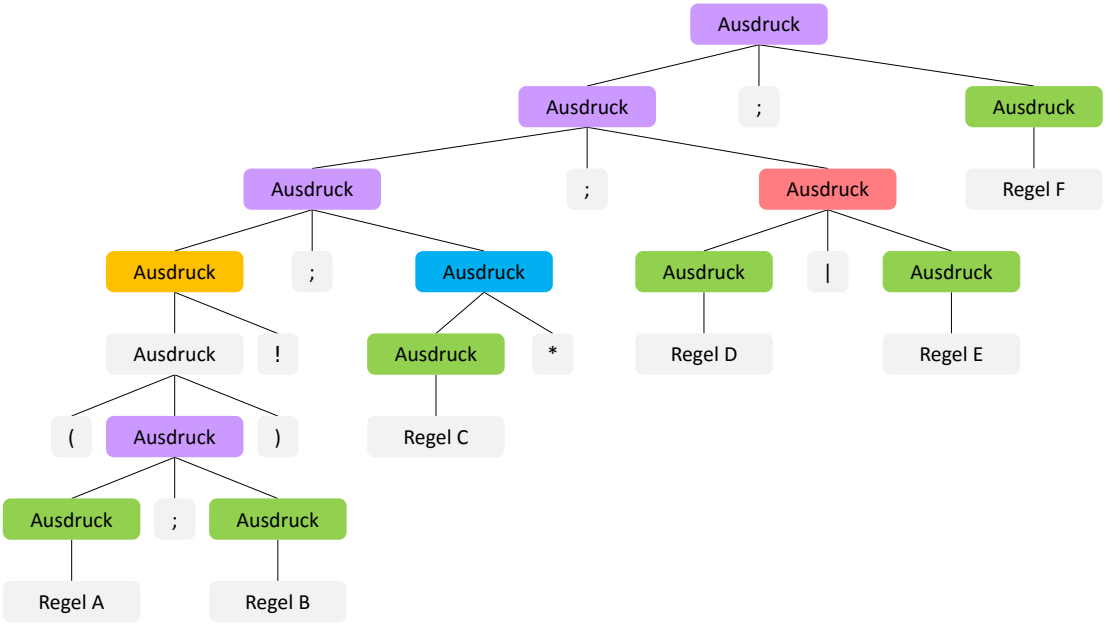

Abbildung 4.4: Knotentypen im Parsebaum

Sequenz-Knoten

Befindet sich der Algorithmus in einem Sequenz-Knoten, besucht er zunächst den linken und anschließend den rechten Kindknoten (s. Abbildung 4.4, violette Knoten).

Alternative-Knoten

Befindet sich der Algorithmus in einem Alternative-Knoten, besucht er entweder den linken oder den rechten Kindknoten (s. Abbildung 4.4, roter Knoten). Die Auswahl des jeweiligen Kindknoten erfolgt zufällig. Dadurch ergibt sich ein nichtdeterministisches Verhalten.

randomNumberOfTimes-Knoten

Befindet sich der Algorithmus in einem randomNumberOfTimes-Knoten, so besucht er beliebig häufig den linken Kindknoten, maximal jedoch so oft, wie kein Fehler auftritt (s. Abbildung 4.4, blauer Knoten). Die Anzahl an Durchläufen wird hierbei zufällig aus einer Zahl zwischen 0 und x gewählt. Der Parameter x kann dabei vom Benutzer bestimmt werden. Durch die zufällige Anzahl an Durchläufen ergibt sich ebenfalls ein nichtdeterministisches Verhalten.

asLongAsPossible-Knoten

Befindet sich der Algorithmus in einem asLongAsPossible-Knoten, besucht er solange den linken Kindknoten, wie kein Fehler aus dem Besuch des Kindknotens hervorgeht (s. Abbildung 4.4, orangefarbener Knoten). Ein Fehler geht beispielsweise dann hervor, wenn der linke Kindknoten ein Regel-Knoten ist und die Regel aufgrund eines nicht-vorhandenen Matches nicht angewendet werden kann. Da der Kindknoten eine Regel ohne Vorbedingung sein könnte, könnten auch hier unendlich viele Besuche stattfinden. Aus diesem Grund kann der Benutzer hier ebenfalls festlegen, wie oft der Kindknoten maximal besucht werden soll.

Das Verhalten des Algorithmus ist somit für alle Typen von Knoten definiert. Der Algorithmus kann mit Hilfe des sogenannten *Visitor-Pattern* implementiert werden. Der rekursive Ansatz bietet hierbei den Vorteil, dass nur wenige Fälle betrachtet werden müssen, um die Funktionalität von Transformationseinheiten mit beliebig komplexen Kontrollausdrücken zu gewährleisten.

Durchführbarkeit einer Transformationseinheit

Wie der Name schon andeutet wird eine Transformationseinheit immer nur als Einheit durchgeführt. Durch diverse nichtdeterministische Faktoren lässt sich im Voraus jedoch nicht ohne Weiteres bestimmen, ob die Transformationseinheit überhaupt fehlerfrei durchgeführt werden kann. Aus diesem Grund ist der hier vorgesehene Ansatz *trial and error*. Bei *trial and error* werden die jeweiligen Regeln einer Transformationseinheit nacheinander angewendet, sofern es möglich ist. Sollte eine Regel nicht angewendet werden können, so muss entschieden werden, wie dies die Ausführung der Transformationseinheit beeinflusst. Hierbei muss grundsätzlich ermittelt werden, ob die Anwendung einer Regel bzw. eines Teilausdrucks in dem jeweiligen Kontext notwendig oder optional war.

Eine Transformationseinheit kann genau dann nicht vollständig durchgeführt werden, wenn eine Regel nicht angewendet werden kann, die jedoch zwingend angewendet werden muss. Angenommen es existiert eine Transformationseinheit mit folgendem Kontrollausdruck:

$$\mathcal{C} = Regel_A; Regel_B | Regel_C$$

Der Kontrollausdruck lässt die beiden folgenden Abfolgen von Transformationen zu:

- $Regel_A; Regel_B$
- $Regel_A; Regel_C$

Unabhängig davon, welche Abfolge letztendlich durchgeführt wird, müssen bei beiden Abfolgen alle Regeln durchgeführt werden, damit die Transformationseinheit als korrekt durchgeführt gilt.

Es gibt jedoch auch Fälle in denen Regeln nicht zwingend angewendet werden müssen. Kann eine Regel, welche sich im Kontext von *asLongAsPossible* oder *randomNumberOfTimes* befindet, nicht angewendet werden, gilt die Transformationseinheit weiterhin als korrekt durchgeführt. Eine Transformationseinheit mit dem Kontrollausdruck $Regel_A!$ ist dann erfolgreich, wenn $Regel_A$ beliebig oft angewendet werden konnte. Die Regel wird demnach so lange angewendet, bis dies nicht mehr möglich ist, oder die maximale Anzahl an Anwendungen erreicht ist. Die Transformationseinheit gilt demnach auch als korrekt durchgeführt, wenn die Regel kein einziges mal angewendet wurde.

trial, error und recovery

Bei der Durchführung einer Transformationseinheit wird das Petrinetz sukzessiv verändert. Da der Ausgang einer Transformationseinheit im Voraus jedoch nicht bekannt ist, müssen auch sukzessiv Abbilder des Petrinetzes zwischengespeichert und gegebenenfalls wiederhergestellt werden. Die möglichen Fälle werden im Folgenden beispielhaft erläutert.

Ist die Anwendung einer zwingend notwendigen Regel nicht möglich, so gilt die Transformationseinheit als gescheitert und der Ausgangszustand muss wiederhergestellt werden. Dies wird beispielhaft an der Transformationseinheit mit dem folgenden Kontrollausdruck erläutert:

$$\mathcal{C} = Regel_A; Regel_B; Regel_C$$

Der Ablauf eines möglichen Szenarios ist in Abbildung 4.5 dargestellt. Zu Beginn der Transformationseinheit wird der Ausgangszustand zwischengespeichert (engl. Snapshot). Die Regeln werden bei dem genannten Kontrollausdruck sequenziell auf das Petrinetz angewendet. Scheitert nun wie dargestellt $Regel_C$, kann der Ausgangszustand mit dem Snapshot wiederhergestellt werden. Die Transformationseinheit ist in diesem Fall nicht anwendbar und das Petrinetz

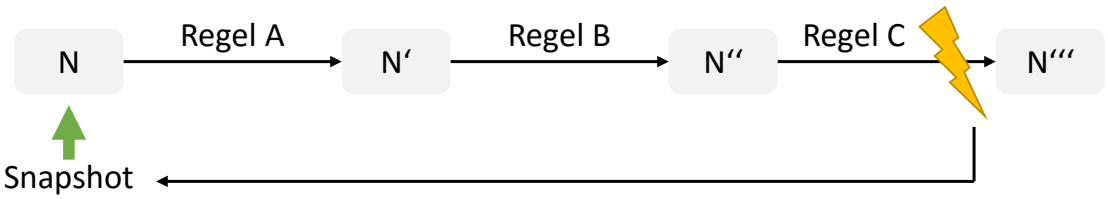

Abbildung 4.5: Wiederherstellung des Ausgangszustandes

bleibt unverändert.

Anders verhält sich die Ausführung einer Transformationseinheit bei nicht zwingend notwendigen Regeln bzw. Teilausdrücken. Hierbei sei beispielhaft die Transformationseinheit mit dem folgenden Kontrollausdruck zu betrachten:

$$\mathcal{C} = Regel_A; (Regel_B; Regel_C)!; Regel_D$$

Der *asLongAsPossible*-Operator ! sorgt dafür, dass der geklammerte Ausdruck so oft wie möglich angewendet werden soll. Ein möglicher Ablauf ist dabei in Abbildung 4.6 dargestellt. Wie im vorherigen Beispiel wird zunächst ein Snapshot vom Ausgangszustand gespeichert (Snapshot 1). Danach wird das Petrinetz N mit $Regel_A$ in das Petrinetz N' überführt. Anschließend beginnt die erste Iteration des geklammerten Ausdrucks. Hierbei wird zunächst ein weiterer Snapshot gespeichert (Snapshot 2). Nun wird das Netz N' durch Anwendung von $Regel_B$ und $Regel_C$ in das Netz N''' überführt. Zu beginn der zweiten Iteration wird erneut ein Snapshot gespeichert (Snapshot 3). Anschließend wird das Netz N''' mit $Regel_B$ in das Netz N'''' überführt. Wie in der Abbildung zu sehen ist, scheitert die darauffolgende Anwendung von $Regel_C$. Da der geklammerte Ausdruck als Einheit durchgeführt werden muss, wird nun Snapshot 3 wiederhergestellt, um die bis dahin getätigten Änderungen zu verwerfen. Anschließend kann das Netz N''' mit $Regel_D$ in das finale Netz N'''' (dunkelgrau) überführt werden.

Durch das Erstellen eines Snapshots zu Beginn jeder Iteration im Kontext von *asLongAsPossible* ist somit gewährleistet, dass der jeweilige Ausdruck immer als Einheit ausgeführt werden kann und, im Falle einer nicht anwendbaren Regel, zum zuvor validen Zustand zurückgekehrt werden kann. Das erläuterte Verhalten findet ebenfalls im Kontext des *randomNumberOfTimes*-Operator statt.

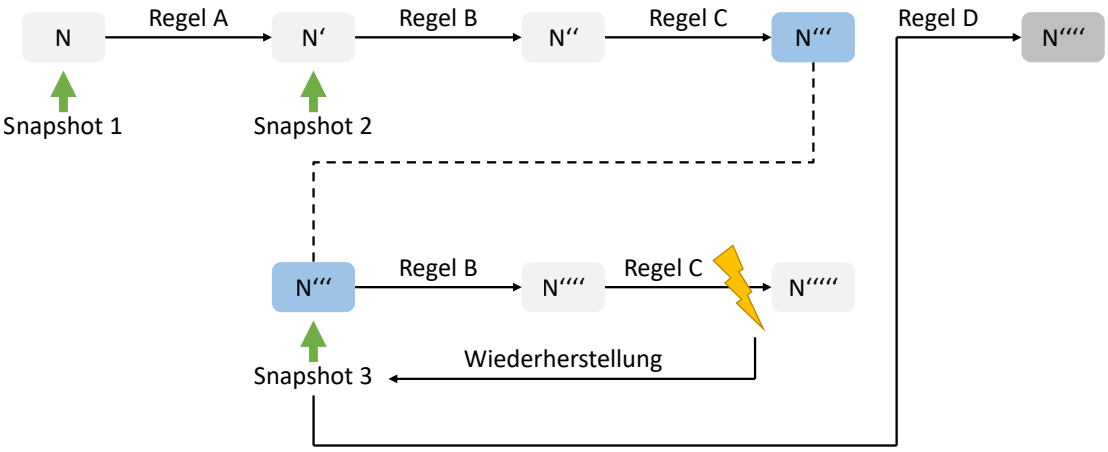

Abbildung 4.6: Wiederherstellung einer Zwischenspeicherung

Der Ansatz von *trial, error & recovery* wurde somit für die zwei elementaren Fälle definiert. Der rekursive Charakter des Parsebaums bietet auch hier den Vorteil, dass beliebig verschachtelte Kontrollausdrücke gleichermaßen abgedeckt sind und somit die ordnungsgemäße Durchführung von Wiederherstellungsmaßnahmen gewährleistet wird.

4.4 GUI Konzept

In Abbildung 4.7 ist die grafische Oberfläche von ReConNet vor Implementierung der Transformationseinheiten dargestellt. Transformationseinheiten sind eigenständige Artefakte im Kontext von ReConNet und stehen somit auf gleicher Ebene wie Petrinetze und Regeln. Aus diesem Grund soll der File Tree [1], analog zu den bestehenden Ordnern, um den Ordner *Transformationseinheiten* erweitert werden. Der Benutzer soll hierbei über ein Kontextmenü neue Transformationseinheiten hinzufügen können. Da die bisher einzige Maske der grafischen Oberfläche bereits sehr viele Informationen darstellt und nicht weiter eingeschränkt werden soll, sollen Transformationseinheiten über ein separates Fenster editierbar sein. In Abbildung 4.8 ist ein Mock-up des benötigten Fensters dargestellt. Der Benutzer soll hierbei den Kontrollausdruck der Transformationseinheit eingeben können. Wurde ein gültiger Kontrollausdruck eingegeben, soll der ermittelte Parsebaum grafisch dargestellt werden. Auf diese Weise kann der Benutzer den Kontrollausdruck verifizieren und weiterhin die Ausführungsreihenfolge der einzelnen Regeln intuitiv nachvollziehen. Wenn ein ungültiger Kontrollausdruck eingegeben wurde, soll der Benutzer durch eine Fehlermeldung darauf hingewiesen werden.

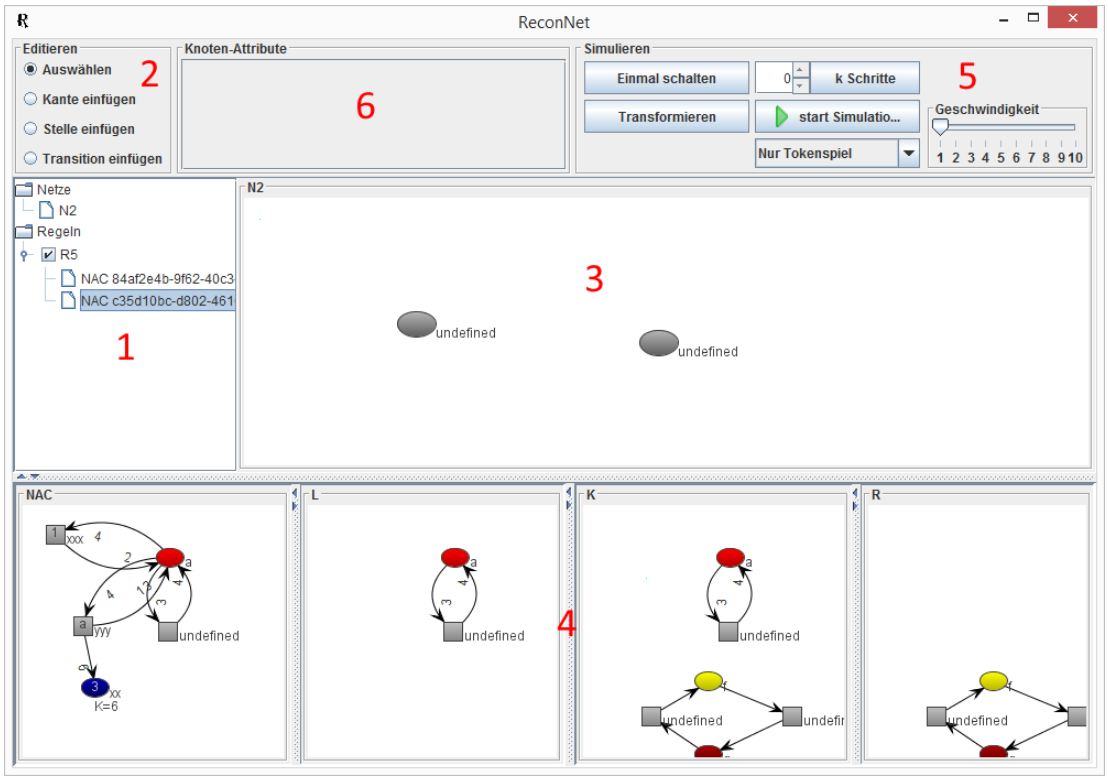

Abbildung 4.7: Grafische Benutzeroberfläche von ReConNet

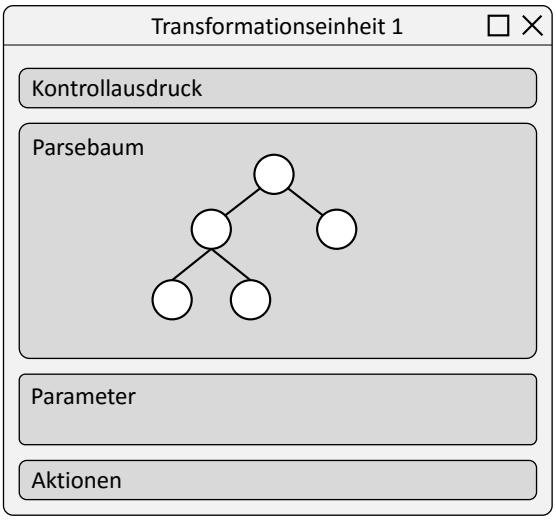

Abbildung 4.8: Mock-up der grafischen Oberfläche für Transformationseinheiten

Die für die Durchführung notwendigen Parameter der Operatoren *asLongAsPossible* und *randomNumberOfTimes* sollen ebenfalls über die grafische Oberfläche eingegeben werden können.

Über einen Button soll der Benutzer die Ausführung der Transformationseinheit initiieren können. Wurde die Transformationseinheit erfolgreich durchgeführt, so sollen dem Benutzer Informationen über die letztendlich durchgeführten Transformationen ausgegeben werden. Konnte die Transformationseinheit nicht durchgeführt werden, soll der Benutzer darüber informiert werden.

Da bei einem komplexen System auch mehrere Transformationseinheiten zum Einsatz kommen können, soll für jede Transformationseinheit eine neue Instanz des Fenster geöffnet werden. Die Interaktion mit der Hauptmaske der grafischen Oberfläche soll hierbei immer möglich sein. Auf diese Weise kann der Benutzer die jeweiligen Transformationseinheiten durchführen und behält dabei den Blick auf das rekonfigurierbare Petrinetz.

5 Implementierung

5.1 Transformationseinheiten in ReConNet

Wie in Abschnitt 4.3 beschrieben ist der Kontrollausdruck das ausschlaggebende Element einer Transformationseinheit. Um eine möglichst lose Kopplung der einzelnen Objekte zu gewährleisten, wird die mathematische Definition im Kontext von ReConNet etwas gelockert. Eine Transformationseinheit in ReConNet besteht daher allein aus dem Kontrollausdruck (s. Listing 5.1). Die letztendliche Referenzierung der Regeln erfolgt programmatisch anhand dieses Kontrollausdrucks.

```
1  public class TransformationUnit {
2
3    private final UUID id;
4    private String controlExpression;
5
6    public TransformationUnit() {
7      this.id = UUID.randomUUID();
8      this.controlExpression = "";
9    }
10
11   public UUID getId() {
12     return this.id;
13   }
14
15   public String getControlExpression() {
16     return this.controlExpression;
17   }
18
19   public void setControlExpression(String controlExpression) {
20     this.controlExpression = controlExpression;
21   }
22 }
```

Listing 5.1: TransformationUnit.java

Zu der Transformationseinheit an sich werden in ReConNet noch Informationen gehalten, welche nur für die jeweilige Sitzung relevant sind. Um diese Informationen zu halten, wurde die Klasse *TransformationUnitData* implementiert (s. Listing 5.2). Die Klasse kapselt die

Transformationseinheit und speichert Informationen über den aktuellen Speicherort sowie die Simulationsparameter für die Operatoren *asLongAsPossible* und *randomNumberOfTimes*.

```
public class TransformationUnitData {

  private String fileName;
  private String filePath;
  private int asLongAsPossibleExecutionLimit;
  private int randomNumberOfTimesUpperRange;
  private TransformationUnit transformationUnit;

  public TransformationUnitData(TransformationUnit transformationUnit,
    String fileName, String filePath) {
    this.transformationUnit = transformationUnit;
    this.fileName = fileName;
    this.filePath = filePath;
  }

  public String getFileName() {
    return this.fileName;
  }

  public String getFilePath() {
    return this.filePath;
  }

  public TransformationUnit getTransformationUnit() {
    return transformationUnit;
  }

  public void setAsLongAsPossibleExecutionLimit(
    int asLongAsPossibleExecutionLimit) {
    this.asLongAsPossibleExecutionLimit = asLongAsPossibleExecutionLimit;
  }

  public int getAsLongAsPossibleExecutionLimit() {
    return this.asLongAsPossibleExecutionLimit;
  }

  public int getRandomNumberOfTimesUpperRange() {
    return this.randomNumberOfTimesUpperRange;
  }

  public void setRandomNumberOfTimesUpperRange(
    int randomNumberOfTimesUpperRange) {
    this.randomNumberOfTimesUpperRange = randomNumberOfTimesUpperRange;
  }

```

```
}
```

Listing 5.2: TransformationUnitData.java

In Abschnitt 4.2 wurden die Erweiterungspunkte erläutert. Transformationseinheiten sind eigenständige Artefakte im Kontext von ReConNet und sollten somit über eine eigene Subkomponente verwaltet werden. Die Komponente muss im Kern folgende Funktionen erfüllen:

- Anlegen und Entfernen von Transformationseinheiten
- Verwaltung von Transformationseinheiten (Kontrollausdruck, Parameter, Sitzungsdaten)
- Ausführen von Transformationseinheiten
- Persistieren von Transformationseinheiten (Dateisystem)

Das Interface *ITransformationUnitManipulation* der neuen Komponente ist in Listing 5.3 dargestellt und stellt die oben genannten Funktionen zur Verfügung.

```
public interface ITransformationUnitManipulation {

  int createTransformationUnit(String fileName, String filePath);
  void removeTransformationUnit(int transformationUnitId);

  String getFileName(int transformationUnitId);

  void
    setControlExpression(int transformationUnitId, String controlExpression);
  String getControlExpression(int transformationUnitId);

  void executeTransformationUnit(int transformationUnitId, int petrinetId,
    Map<String, Integer> ruleNameToId)
      throws EngineException;

  void setAsLongAsPossibleExecutionLimit(int transformationUnitId,
    int executionLimit);
  int getAsLongAsPossibleExecutionLimit(int transformationUnitId);

  void setRandomNumberOfTimesUpperRange(int transformationUnitId,
    int randomNumberOfTimesUpperRange);
  int getRandomNumberOfTimesUpperRange(int transformationUnitId);

  void saveToFileSystem(int transformationUnitId)
    throws EngineException;

  int loadFromFileSystem(String displayName, String filePath)
    throws EngineException;
}
```

Listing 5.3: ITransformationUnitManipulation.java

Die Subkomponente *SessionManager* der Engine-Komponente dient als Speicher der Sitzungsdaten von Netzen und Regeln. Die Komponente wurde erweitert, um Sitzungsdaten von Transformationseinheiten halten zu können (*TransformationUnitData*). Die Komponente für Transformationseinheiten implementiert das oben aufgeführte Interface *ITransformationUnitManipulation* und ermöglicht über den SessionManager die Manipulation von Sitzungsdaten (s. Listing 5.4).

```
1  private TransformationUnitHandler() {
2    this.sessionManager = SessionManager.getInstance();
3  }
4
5  public int createTransformationUnit(String fileName, String filePath) {
6    TransformationUnit transformationUnit = new TransformationUnit();
7
8    return this.sessionManager.createTransformationUnitData(
9      transformationUnit, fileName, filePath);
10  }
11
12 public void removeTransformationUnit(int transformationUnitId) {
13   this.sessionManager.removeTransformationUnitData(transformationUnitId);
14 }
15
16 public void setControlExpression(int transformationUnitId,
17   String controlExpression) {
18     this.sessionManager.getTransformationUnitData(transformationUnitId)
19           .getTransformationUnit().setControlExpression(controlExpression);
20 }
21
22 public String getControlExpression(int transformationUnitId) {
23   return this.sessionManager.getTransformationUnitData(transformationUnitId)
24     .getTransformationUnit().getControlExpression();
25 }
26
27 ...
```

Listing 5.4: TransformationUnitHandler.java

5.2 Grammatik und Parsergenerierung

Auf Basis von Kapitel 4 erfolgt die Implementierung der Transformationseinheiten. Hierbei wurde unter anderem erarbeitet, dass der Kontrollausdruck einer Transformationseinheit in einen Parsebaum überführt werden sollte, um einen intuitiven Umgang damit zu gewährleisten.

Für die Implementierung der Kontrollausdrücke wird der Parsergenerator ANTLR verwendet.

Die formale Sprache zum Ausdrücken aller möglichen Kontrollausdrücke wird mit einer Grammatik definiert. Auf Basis dieser Grammatik kann mit ANTLR ein Parser generiert werden. Dieser Parser kann anschließend herangezogen werden, um Kontrollausdrücke einzulesen und zu verarbeiten.

In Listing 5.5 ist die implementierte Grammatik dargestellt. Der Einstieg in die Grammatik erfolgt über die Startregel *prog* in Zeile 7. Die Regel verlangt einen Ausdruck *expression*, gefolgt vom terminierendem Symbol *EOF*. Der eigentliche Ausdruck *expression* ist dabei rekursiv in den Zeilen 11 bis 18 definiert. Durch das Zeichen | werden jeweils Alternativen definiert. Eine *expression* kann demnach eine der sechs Formen annehmen, welche in den Zeilen 12 bis 17 definiert sind. Durch die Reihenfolge der Alternativen in der Grammatik ergibt sich die benötigte Operatorrangfolge beim späteren Parsen der Kontrollausdrücke. Geklammerte Ausdrücke haben demnach die höchste Priorität (s. Zeile 12), in Sequenz vorliegende Ausdrücke die niedrigste (s. Zeile 16). Zeile 17 bildet sozusagen die Abbruchbedingung der Rekursion, da $expression$ zuallerletzt ein Terminalsymbol annehmen muss, welches durch *IDENTIFIER* spezifiziert wird. Gültige Terminalsymbole sind demnach beliebige Zeichenketten aus Buchstaben und Zahlen, welche zwingend mit einem Buchstaben anfangen müssen (s. Zeile 21). Die syntaktischen Zusätze *left=*, *right=* sowie *#Name* werden im späteren Verlauf näher erläutert.

```
1  grammar ExpressionGrammar;
2
3  @header {
4      package transformation.units.utils;
5  }
6
7  prog
8          : expression EOF
9          ;
10
11 expression
12         : '(' expression ')'                    # parenthesesExpression
13         | left=expression '!'                   # asLongAsPossibleExpression
14         | left=expression '*'                   # randomNumberOfTimesExpression
15         | left=expression '|' right=expression  # choiceExpression
16         | left=expression ';' right=expression  # combinedExpression
17         | IDENTIFIER                            # atomExpression
18         ;
19
20 IDENTIFIER
21         : [a-zA-Z] [a-zA-Z0-9]*
22         ;
```

Listing 5.5: ExpressionGrammar.g4

Nach Implementierung der Grammatik wurde mit dem Parsergenerator ANTLR ein Parser für diese Grammatik generiert. Die Generierung erfolgte über die Kommandozeile (s. Listing 5.6). ANTLR hat hierbei die Klassen erzeugt, welche zum Parsen der Kontrollausdrücke benötigt werden. Durch den Parameter *-visitor* wurden zusätzliche Visitor-Klassen erzeugt, die zum Traversieren des Parsebaums herangezogen werden.

```
1 java -jar antlr-4.5.3-complete.jar ExpressionGrammar.g4 -visitor
```

Listing 5.6: Generierung des Parsers

Die von ANTLR erzeugten Klassen wurden dem ReConNet Projekt hinzugefügt:

- ExpressionGrammarLexer.java
- ExpressionGrammarParser.java
- ExpressionGrammarVisitor.java
- ExpressionGrammarBaseVisitor.java
- ExpressionGrammarListener.java
- ExpressionGrammarBaseListener.java

Die *@header* Konfiguration in der Grammatik hat dafür gesorgt, dass die Klassen mit der spezifizierten Paketangabe generiert wurden (s. Listing 5.5, Zeile 4). Nachträgliche Anpassungen der generierten Klassen sind somit nicht notwendig, wodurch der Einbindungsprozess beschleunigt wird.

5.3 Visitor- und Listener-Pattern

Im vorherigen Abschnitt wurde die Grammatik erläutert und mit Hilfe von ANTLR die notwendigen Klassen zum Parsen von Kontrollausdrücken generiert. Der Parser überführt den jeweiligen Kontrollausdruck in einen Parsebaum. Hierbei gibt es prinzipiell zwei verschiedene Techniken, wie mit dem Parsebaum gearbeitet werden kann, um die gewünschte Anwendungslogik auszuführen. Zum einen kann das *Listener-Pattern* verwendet werden und zum anderen das *Visitor-Pattern*. Das *Listener-Pattern* kann verwendet werden, um auf Ereignisse zu reagieren, wenn der ANTLR-eigene *ParseTreeWalker* den Parsebaum traversiert. Das *Visitor-Pattern* kann verwendet werden, um den Parsebaum manuell zu traversieren. Hier kann eigenhändig implementiert werden, welche Knoten des Parsebaums durchlaufen werden.

Um beide Techniken anwenden zu können sind in der Grammatik sogenannte *alternativ labels* notwendig. Trifft die Parser-Regel *expression* zu, muss unterscheidbar sein, welche der Alternativen letztendlich gewählt wurde (s. 5.5, Zeile 12-17). Ein *alternativ label* kennzeichnet eine Alternative einer Parser-Regel mit einem Namen. Auf diese Weise kann innerhalb der Listener- und Visitor-Klassen die jeweilige Alternative explizit behandelt werden.

Listing 5.7 zeigt das von ANTLR generierte Interface, welches von einer Visitor-Klasse implementiert werden kann. Aufgrund der *alternativ labels* in der Grammatik gibt es auch für jede Alternative eine eigene Methode. Die ebenfalls von ANTLR generierte Klasse *ExpressionGrammarBaseVisitor* implementiert dieses Interface mit Standardfunktionalität. Die Klasse kann dabei als Basisklasse verwendet werden, wenn nicht alle Methoden für das Visitor-Pattern zwingend implementiert werden müssen.

```
1  package transformation.units.utils;
2
3  import org.antlr.v4.runtime.tree.ParseTreeVisitor;
4
5  public interface ExpressionGrammarVisitor<T> extends ParseTreeVisitor<T>
6  {
7    T visitProg(ProgContext ctx);
8    T visitParenthesesExpression(ParenthesesExpressionContext ctx);
9    T visitCombinedExpression(CombinedExpressionContext ctx);
10   T visitAtomExpression(AtomExpressionContext ctx);
11   T visitChoiceExpression(ChoiceExpressionContext ctx);
12   T visitAsLongAsPossibleExpression(AsLongAsPossibleExpressionContext ctx);
13   T visitRandomNumberOfTimesExpression(RandomNumberOfTimesExpressionContext ctx);
14 }
```

Listing 5.7: ExpressionGrammarVisitor.java

Listing 5.8 zeigt analog das Interface nach dem Listener-Pattern. Hier werden jeweils Methoden bereitgestellt, welche bei Regel-Ein und -Austritt aufgerufen werden, wenn der ParseTreeWalker den Parsebaum traversiert. Auch hier wurde von ANTLR eine Basisklasse *ExpressionGrammarBaseListener* generiert, welche Standardfunktionalität für alle Methoden implementiert.

```
1  package transformation.units.utils;
2
3  import org.antlr.v4.runtime.tree.ParseTreeListener;
4
5  public interface ExpressionGrammarListener extends ParseTreeListener
6  {
7    void enterProg(ProgContext ctx);
8    void exitProg(ProgContext ctx);
9
10   void enterParenthesesExpression(ParenthesesExpressionContext ctx);
11   void exitParenthesesExpression(ParenthesesExpressionContext ctx);
```

```
12
13    void enterCombinedExpression(CombinedExpressionContext ctx);
14    void exitCombinedExpression(CombinedExpressionContext ctx);
15
16    ...
17 }
```

Listing 5.8: ExpressionGrammarListener.java

5.4 Eingabe und Überprüfung des Kontrollausdrucks

Bevor eine Transformationseinheit ausgeführt werden kann, muss zunächst überprüft werden, ob der eingegebene Kontrollausdruck korrekt ist. Der Kontrollausdruck muss zum einen syntaktisch Korrekt sein und zum anderen dürfen im Kontrollausdruck nur Regeln referenziert werden, welche im aktuellen ReConNet Kontext auch vorhanden sind.

Die Initialisierung des Parsers ist in Listing 5.9 dargestellt. Der Kontrollausdruck wird hierbei als String übergeben (s. Zeile 1, *expression*).

```
1 ANTLRInputStream inputStream = new ANTLRInputStream(expression);
2 ExpressionGrammarLexer lexer = new ExpressionGrammarLexer(inputStream);
3 CommonTokenStream tokenStream = new CommonTokenStream(lexer);
4
5 ExpressionGrammarParser parser = new ExpressionGrammarParser(tokenStream);
```

Listing 5.9: Initialisierung des Parsers

Die syntaktische Analyse erfolgt bei ANTLR automatisch. Das Verhalten des Parsers im Falle eines Verarbeitungsfehlers muss jedoch spezifiziert werden. Im Normalfall wird der Parse-Vorgang bei syntaktischen Fehlern nicht unterbrochen. Dies ist sinnvoll für umfassende Fehlerberichte, beispielsweise zum Anprangern aller Syntaxfehler in einem Java Quellcode. Für einfache Transformationseinheiten wäre dies zu komplex. In diesem Fall ist es ausreichend, dass der Benutzer lediglich darauf hingewiesen wird, wenn der Kontrollausdruck syntaktisch falsch ist. Die sogenannte *BailErrorStrategy* legt fest, dass der Parse-Vorgang bei syntaktischen Fehlern sofort abgebrochen wird (s. Listing 5.10).

```
1 parser.setErrorHandler(new BailErrorStrategy());
```

Listing 5.10: BailErrorStrategy

Ob im Kontrollausdruck nur tatsächlich vorhandene Regeln referenziert sind, wird mit Hilfe einer Listener-Klasse überprüft. Die hierfür implementierte Klasse *UnknownRuleInExpressionListener* ist in Listing 5.11 dargestellt. Dem Listener wird ein Set mit den Namen der zur Zeit

vorhandenen Regeln übergeben. Nachdem der Parser ein Terminalsymbol eingelesen hat, wird die Methode *exitAtomExpression* aufgerufen. Hier wird überprüft, ob das Terminalsymbol eine gültige Regel ist. Ist dies nicht der Fall, wird eine *UnknownRuleInExpressionException* mit dem Terminalsymbol geworfen und der Parsevorgang wird abgebrochen.

```
public class UnknownRuleInExpressionListener
extends ExpressionGrammarBaseListener {

  private Set<String> ruleNames;

  public UnknownRuleInExpressionListener(Set<String> ruleNames) {
    this.ruleNames = ruleNames;
  }

  @Override
  public void exitAtomExpression(AtomExpressionContext ctx) {

    if (!ruleNames.contains(ctx.getText())) {
      throw new RuntimeException(new UnknownRuleInExpressionException(
        ctx.getText()));
    }
  }

  public class UnknownRuleInExpressionException
    extends Exception {

    private static final long serialVersionUID = 1L;
    private String ruleName;

    public UnknownRuleInExpressionException(String ruleName) {
      this.ruleName = ruleName;
    }

    public String getRuleName() {
      return this.ruleName;
    }
  }

}
```

Listing 5.11: UnknownRuleInExpressionListener.java

5.5 Durchführung der Transformationseinheit

In Abschnitt 4.3.3 wurde der prinzipielle Vorgang zur Durchführung einer Transformationseinheit auf Basis eines Parsebaums dargestellt. Für die Implementierung des Algorithmus wird das Visitor-Pattern verwendet, da dieses eine explizite Traversierung des Parsebaums ermöglicht.

Hierfür wurde die Klasse *TransformationUnitExecutionVisitor* auf Basis der Visitor-Basisklasse erstellt (s. Listing 5.12).

```
1  public class TransformationUnitExecutionVisitor
2  extends ExpressionGrammarBaseVisitor<Void> {
3
4    private Random dice = new Random();
5    private List<String> executedRules = new ArrayList<String>();
6
7    private int petrinetId;
8    private Map<String, Integer> ruleNameToId;
9    private int asLongAsPossibleExecutionLimit;
10   private int randomNumberOfTimesUpperRange;
11
12   public TransformationUnitExecutionVisitor(int petrinetId,
13     Map<String, Integer> ruleNameToId, int asLongAsPossibleExecutionLimit,
14     int randomNumberOfTimesUpperRange) {
15
16     this.petrinetId = petrinetId;
17     this.ruleNameToId = ruleNameToId;
18     this.asLongAsPossibleExecutionLimit = asLongAsPossibleExecutionLimit;
19     this.randomNumberOfTimesUpperRange = randomNumberOfTimesUpperRange;
20   }
```

Listing 5.12: TransformationUnitExecutionVisitor.java

Die Klasse wird mit den für die Transformationskomponente notwendigen Parametern initialisiert, sowie den Parametern für die Operatoren *asLongAsPossible* und *randomNumberOfTimes*. Der Parameter *asLongAsPossibleExecutionLimit* steuert die maximale Anzahl an Durchläufen bei Verwendung dieses Operators. Der Parameter *randomNumberOfTimesUpperRange* bestimmt die obere Grenze für die zufällig gewählte Anzahl an Durchläufen bei diesem Operator.

Im folgenden werden die einzelnen Methoden vorgestellt, die beim Traversieren des Parsebaums aufgerufen werden können. In Listing 5.13 ist die Methode dargestellt, welche beim Besuch eines *atomExpression*-Knoten ausgeführt wird. Wie der Grammatik zu entnehmen ist, verbirgt sich im atomExpression-Kontext der Bezeichner *IDENTIFIER* der durchzuführenden Regel. Auf diesen Bezeichner wird in Zeile 25 zugegriffen. Da alle Regeln einer ReConNet Sitzung anhand von IDs referenziert werden, muss für die Transformationskomponente der Bezeichner auf die aktuelle Sitzungs-ID abgebildet werden. Anschließend wird versucht die Regel auf das aktuelle Petrinetz anzuwenden (s. Zeile 29). Nach Anwendung der Regel wird der Name der Regel gespeichert. Nach der Durchführung einer Transformationseinheit kann somit ermittelt werden, welche Regeln letztendlich angewendet worden sind.

Die Transformationskomponente wirft eine *EngineException* sofern die jeweilige Regel nicht angewendet werden kann. Dieses Verhalten wurde sich für die Implementierung der Wiederherstellungslogik zunutze gemacht. Die *EngineException* wird als *RuntimeException* gekapselt und geworfen. Da der Parsebaum rekursiv traversiert wird, wird die *RuntimeException*, bildlich gesprochen, den Parsebaum hinauf geworfen. Eine *RuntimeException* hat dabei die Eigenschaft, dass sie nicht zwingend behandelt werden muss.

```
22  @Override
23  public Void visitAtomExpression(AtomExpressionContext ctx) {
24
25    String ruleName = ctx.IDENTIFIER().getText();
26    int ruleId = this.ruleNameToId.get(ruleName);
27
28    try {
29      SimulationHandler.getInstance().transform(this.petrinetId, ruleId);
30      executedRules.add(ruleName);
31    } catch (EngineException e) {
32      throw new RuntimeException(e);
33    }
34
35    return null;
36  }
```

Listing 5.13: TransformationUnitExecutionVisitor.java

In Listing 5.14 ist die Methode zur Bearbeitung eines *combinedExpression*-Knotens dargestellt. Hier wird lediglich der linke und anschließend der rechte Kindknoten besucht. Die in der Grammatik definierten Label *left* und *right* können hier für das explizite Adressieren der Knoten verwendet werden.

```
38  @Override
39  public Void visitCombinedExpression(CombinedExpressionContext ctx) {
40
41    visit(ctx.left);
42    visit(ctx.right);
43
44    return null;
45  }
```

Listing 5.14: TransformationUnitExecutionVisitor.java

In Listing 5.15 ist der Fall für den *choiceExpression*-Knoten dargestellt. Hier wird zufällig entweder der linke oder rechte Kindknoten besucht.

```
47  @Override
48  public Void visitChoiceExpression(ChoiceExpressionContext ctx) {
49
50    boolean random = dice.nextBoolean();
```

```
51
52      if (random) {
53        visit(ctx.left);
54      } else {
55        visit(ctx.right);
56      }
57
58      return null;
59    }
```

Listing 5.15: TransformationUnitExecutionVisitor.java

Die in Listing 5.14 und 5.15 dargestellten Methoden haben keine Fehlerbehandlung. Etwaig auftretende *RuntimeExceptions* bei Besuch der Kindknoten müssen dementsprechend an anderer Stelle behandelt werden. Bildlich gesprochen werden die *RuntimeExceptions* implizit nach oben gereicht.

In Abschnitt 4.3.3 wurde die Durchführbarkeit einer Transformationseinheit erläutert. Eine Regel bzw. ein Teilausdruck einer Transformationseinheit kann entweder zwingend notwendig oder optional sein. Ein Ausdruck ist immer dann optional, wenn dieser im Kontext von *asLongAsPossible* oder *randomNumberOfTimes* ist. Hier wird nochmals der Kontrollausdruck

$$\mathcal{C} = Regel_A; (Regel_B; Regel_C)!; Regel_D$$

als Beispiel herangezogen. Scheitert die Anwendung von $Regel_C$ in einer beliebigen Iteration, sorgt dies nicht für den Abbruch der Transformationseinheit. Es muss jedoch sichergestellt werden, dass $(Regel_B; Regel_C)$ immer nur als ganzes ausgeführt wird. Scheitert $Regel_C$ in einer beliebigen Iteration, müssen auch die bis dahin durch $Regel_B$ verursachten Änderungen dieser Iteration verworfen werden.

In der Implementierung werden die zuvor genannten *RuntimeExceptions* dazu verwendet, dieses Verhalten abzubilden. In Listing 5.16 ist die Methode zur Bearbeitung eines *asLongAsPossible*-Knotens dargestellt. Zu Beginn jeder Iteration wird ein Snapshot des aktuellen Petrinetzes erstellt (s. Zeile 72-73). Dies passiert sowohl für die Modelldaten, als auch für die zugehörigen JUNG-spezifischen Layoutdaten. Für die Erstellung von Snapshots wurden jeweils Kopierkonstruktoren implementiert. Zusätzlich wird noch ein Snapshot der aktuellen Liste von bereits angewandten Regeln erzeugt. Anschließend wird der von *asLongAsPossible* eingeschlossene Ausdruck ausgeführt (s. Zeile 79). Ist der Ausdruck erfolgreich durchgeführt worden,

beginnt die nächste Iteration. Wird bei der Ausführung des Ausdrucks eine *RuntimeException* gefangen, bedeutet dies, dass eine Regel des Ausdrucks nicht angewendet werden konnte.

```
61  @Override
62  public Void visitAsLongAsPossibleExpression(
63    AsLongAsPossibleExpressionContext ctx) {
64
65    for (int i = 0; i < asLongAsPossibleExecutionLimit; i++) {
66
67      Petrinet petrinet =
68        SessionManager.getInstance().getPetrinetData(petrinetId).getPetrinet();
69      JungData jungData =
70        SessionManager.getInstance().getPetrinetData(petrinetId).getJungData();
71
72      Petrinet petrinetSnapshot = new Petrinet(petrinet);
73      JungData jungDataSnapshot = new JungData(jungData, petrinetSnapshot);
74
75      List<String> executedRulesSnapshot =
76        new ArrayList<String>(executedRules);
77
78      try {
79        visit(ctx.left);
80      } catch (RuntimeException e) {
81        SessionManager.getInstance().replacePetrinetData(petrinetId,
82          petrinetSnapshot, jungDataSnapshot);
83        this.executedRules = executedRulesSnapshot;
84        break;
85      }
86
87    }
88
89    return null;
90  }
```

Listing 5.16: TransformationUnitExecutionVisitor.java

Da der von *asLongAsPossible* eingeschlossene Ausdruck immer nur als Ganzes durchgeführt werden soll, muss der zuvor gespeicherte Snapshot wiederhergestellt werden. Die Liste der bereits angewandten Regeln muss dementsprechend ebenfalls zurückgesetzt werden (s. Zeile 81-83). Beim ersten Scheitern wird die Schleife unterbrochen und die Semantik vom *asLongAs-Possible*-Operator ist somit gewährleistet.

Die Methode zur Bearbeitung eines *randomNumberOfTimes*-Knoten arbeitet analog zu der zuvor erläuterten Methode. Hier wird lediglich eine zufällige Anzahl an Durchläufen durchgeführt. Die Anzahl an Durchläufen wird hierbei zufällig zwischen 0 und dem Parameter *randomNumberOfTimesUpperRange* gewählt.

Zuallerletzt ist in Listing 5.17 die Methode für den Einstieg in den Parsebaum dargestellt. Dies ist dementsprechend die oberste Ebene der Aufrufhierarchie. Zunächst wird der Ursprungszustand des Netztes als Snapshot gespeichert (s. Zeile 133-134). Anschließend wird der Kindknoten besucht und somit die Traversierung des Parsebaums eingeleitet (s. Zeile 137). Wird auf dieser Ebene eine *RuntimeException* gefangen, kann die verursachende Regel nur zwingend notwendig gewesen sein. Dies bedeutet, dass die Durchführung der Transformationseinheit gescheitert ist. Dementsprechend muss der Ursprungszustand des Netzes wiederhergestellt werden (s. Zeile 140). Tritt bei der Traversierung des Parsebaums keine *RuntimeException* auf, gilt die Transformationseinheit als erfolgreich.

```
125   @Override
126   public Void visitProg(ProgContext ctx) {
127
128     Petrinet petrinet =
129       SessionManager.getInstance().getPetrinetData(petrinetId).getPetrinet();
130     JungData jungData =
131       SessionManager.getInstance().getPetrinetData(petrinetId).getJungData();
132
133     Petrinet petrinetSnapshot = new Petrinet(petrinet);
134     JungData jungDataSnapshot = new JungData(jungData, petrinetSnapshot);
135
136     try {
137       visit(ctx.expression());
138       visit(ctx.EOF());
139     } catch (RuntimeException e) {
140       SessionManager.getInstance().replacePetrinetData(petrinetId,
141         petrinetSnapshot, jungDataSnapshot);
142       throw e;
143     }
144
145     return null;
146   }
```

Listing 5.17: TransformationUnitExecutionVisitor.java

Der Implementierte Visitor kann nun verwendet werden, um Parsebäume von Kontrollausdrücken zu traversieren und die Transformationseinheit durchzuführen (s. Listing 5.18).

```
1 ParseTree parseTree = parser.prog();
2
3 TransformationUnitExecutionVisitor visitor =
4   new TransformationUnitExecutionVisitor(petrinetId, ruleNameToId,
5         asLongAsPossibleExecutionLimit, randomNumberOfTimesUpperRange);
6
7 try {
8   visitor.visit(parseTree);
```

```
9  } catch (RuntimeException e) {
10   throw new EngineException(
11         "Die Transformationseinheit konnte nicht angewendet werden.");
12 }
```

Listing 5.18: TransformationUnitHandler.java

Die Implementierung der Logik zur Durchführung von Transformationseinheiten ist somit abgeschlossen. Durch den rekursiven Ansatz mit Hilfe des Parsebaums konnte sich auf die elementaren Fälle konzentriert werden. Dadurch sind beliebig komplexe Kontrollausdrücke stets vollständig abgedeckt, wodurch etwaige Fehlerquellen so gut wie möglich vermieden werden. Die Implementierung erfolgte unter Berücksichtigung der in Abschnitt 4.2 genannten Erweiterungspunkte. Die Funktion zur Durchführung einer Transformationseinheit wird über das Interface *ITransformationUnitManipulation* bereitgestellt.

5.6 Grafische Oberfläche

Im Zuge der Implementierung wurde ebenfalls die grafische Oberfläche erweitert, um das Arbeiten mit Transformationseinheiten zu ermöglichen. Die GUI-Komponente greift hierzu über das Interface *ITransformationUnitManipulation* auf Methoden der Transformationseinheit-Komponente zu. Das *FileTreePane* wurde dementsprechend erweitert, um Transformationseinheiten verwalten zu können. Transformationseinheiten können hierbei über ein Kontextmenü erstellt oder geladen werden (s. Abbildung 5.1). Mit Klick auf eine Transformationseinheit öffnet sich der Editor. Hier kann zunächst der Kontrollausdruck eingegeben werden. Bevor die Transformationseinheit ausgeführt werden kann, muss der Kontrollausdruck überprüft werden. Der Benutzer wird dabei auf syntaktische Fehler hingewiesen. Weiterhin wird der Benutzer darauf hingewiesen, wenn der Kontrollausdruck Regeln referenziert, welche nicht in der aktuellen Sitzung von ReConNet geladen sind (s. Abbildung 5.2).

Ist der Kontrollausdruck zulässig, wird der daraus erstellte Parsebaum grafisch dargestellt (s. Abbildung 5.3). Die strukturierte Darstellung durch den Parsebaum gewährleistet auch bei komplexen Kontrollausdrücken ein intuitives Arbeiten. Die Parameter für die Operatoren *asLongAsPossible* sowie *randomNumberOfTimes* können über die Oberfläche eingegeben werden. Mit Klick auf den untersten Button wird die Transformationseinheit ausgeführt. Ist die Transformationseinheit erfolgreich durchgeführt worden, wird dem Benutzer die durchgeführte Abfolge von Regeln dargestellt. Andernfalls wird der Benutzer darüber informiert, dass die Durchführung der Transformationseinheit gescheitert ist.

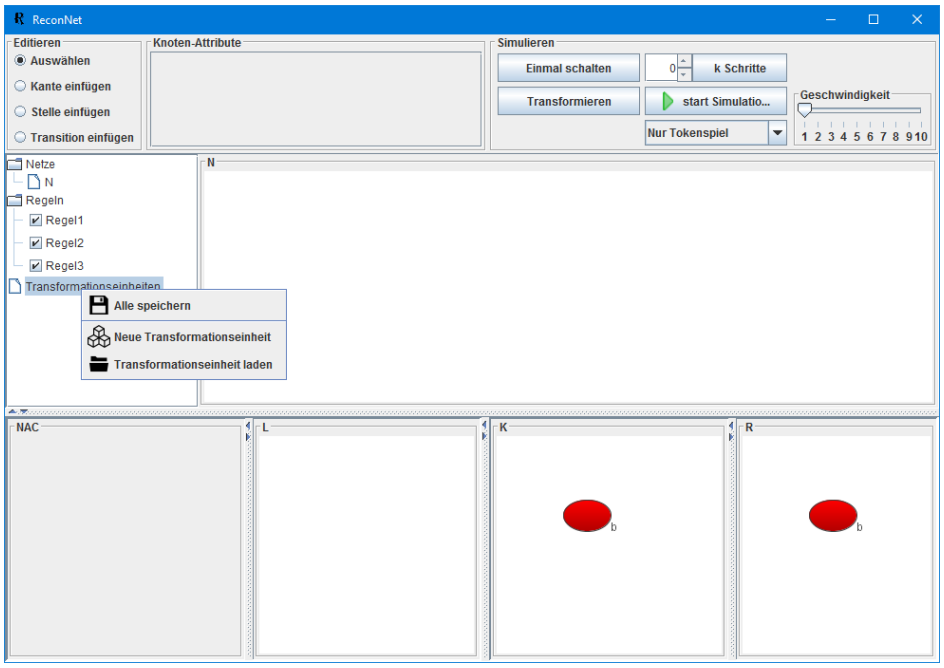

Abbildung 5.1: FileTreePane Kontext Menü

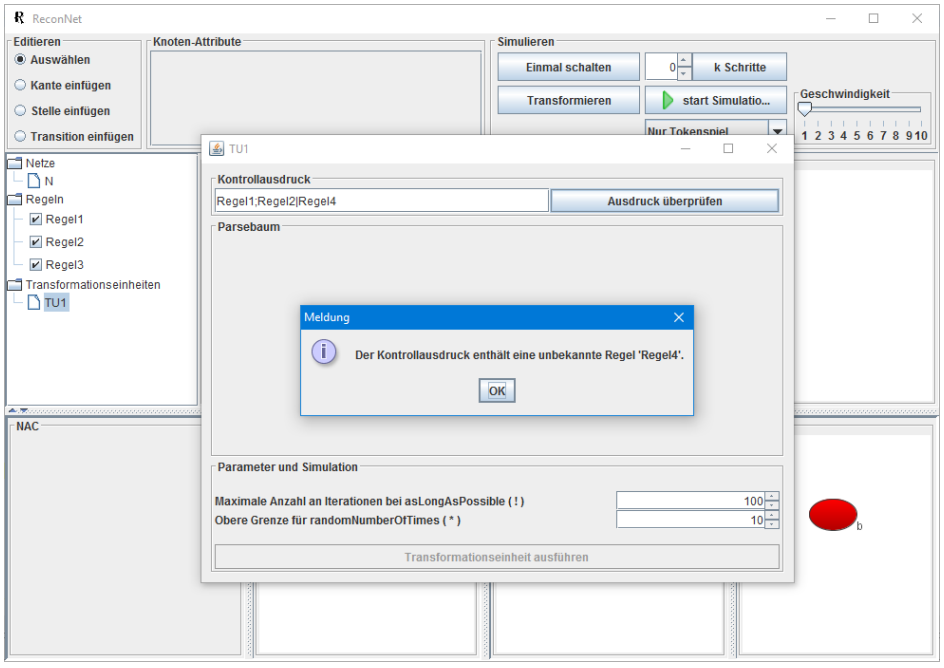

Abbildung 5.2: Unbekannte Regel im Kontrollausdruck

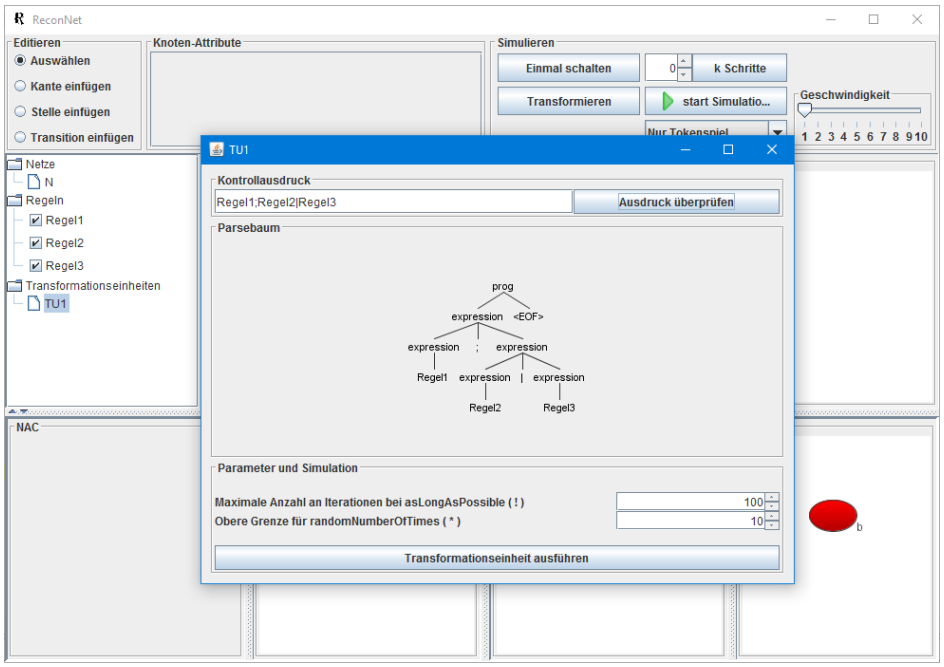

Abbildung 5.3: Darstellung des Parsebaums

5.7 Persistenz

Zur Serialisierung und Deserialisierung von Transformationseinheiten wurde die *Gson* Bibliothek verwendet [Gso]. Die Bibliothek bietet Methoden zum Konvertieren von Java Objekten in die *JavaScript Object Notation* (JSON) und umgekehrt. Die Persistenz-Komponente wurde dementsprechend um die Methoden zum Speichern und Laden von Transformationseinheiten erweitert (s. Listing 5.19).

```
1  public static void saveTransformationUnit(
2    TransformationUnit transformationUnit, String filePath)
3    throws Exception {
4
5    Gson gson = new Gson();
6    String jsonString = gson.toJson(transformationUnit);
7
8    FileOutputStream fos = null;
9
10   try {
11     fos = new FileOutputStream(filePath);
12     fos.write(jsonString.getBytes());
13   }
14   catch
15   ...
```

```
16 }
17
18 public static TransformationUnit loadTransformationUnit(String filePath)
19   throws Exception {
20
21   Gson gson = new Gson();
22
23   FileInputStream fis = null;
24
25   try {
26     fis = new FileInputStream(filePath);
27     TransformationUnit transformationUnit =
28       gson.fromJson(new InputStreamReader(fis), TransformationUnit.class);
29     return transformationUnit;
30   }
31   catch
32   ...
33 }
```

Listing 5.19: Persistence.java

6 Test

6.1 Testdurchführung

Nach der Implementierung erfolgte der Test der jeweiligen Funktionen. Hierfür wurden Regeln definiert mit denen die einzelnen Funktionen separat und im Verbund getestet wurden. Die Regeln sind in den Abbildungen 6.1, 6.2, 6.3 und 6.4 in der Form $r = (NAC, L, K, R)$ dargestellt.

- Regel AddA
  - Fügt dem Netz eine Stelle a hinzu.
  - Kann nicht angewendet werden, sofern im Netz vier Stellen a vorhanden sind.
- Regel AddC
  - Fügt dem Netz eine Stelle c hinzu.
- Regel AtoB
  - Fügt dem Netz eine Transition $t1$ und eine Stelle b hinzu und verbindet diese mit einer bestehenden Stelle a zu $a \rightarrow t1 \rightarrow b$.
  - Die Regel kann nicht angewendet werden, sofern Stelle a bereits eine solche Verbindung hat
- Regel BtoC
  - Fügt dem Netz eine Transition $t2$ hinzu und verbindet hiermit eine vorhandene Stelle b mit einer vorhandenen Stelle c zu $b \rightarrow t2 \rightarrow c$.
  - Die Regel kann nicht angewendet werden, sofern bereits eine solche Verbindung zwischen b und c besteht.

Mit Hilfe der dargestellten Regeln wurden Testfälle entwickelt, welche die elementaren Funktionen der Transformationseinheiten abdecken. Tabelle 6.1 gibt hierbei eine Übersicht der

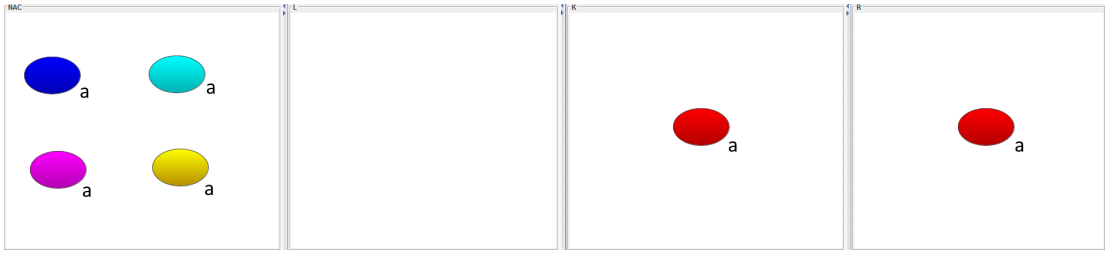

Abbildung 6.1: Regel AddA

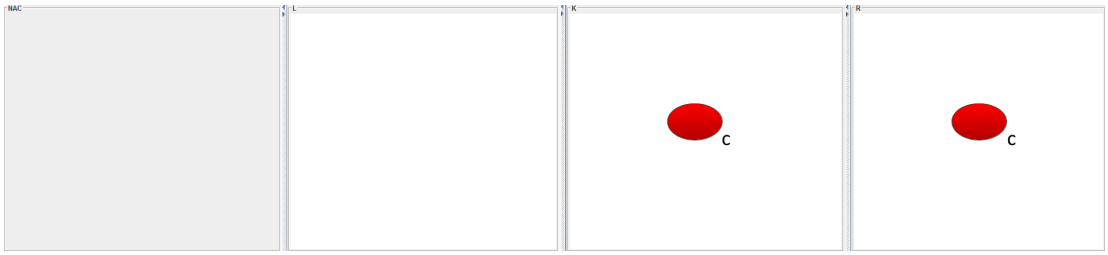

Abbildung 6.2: Regel AddC

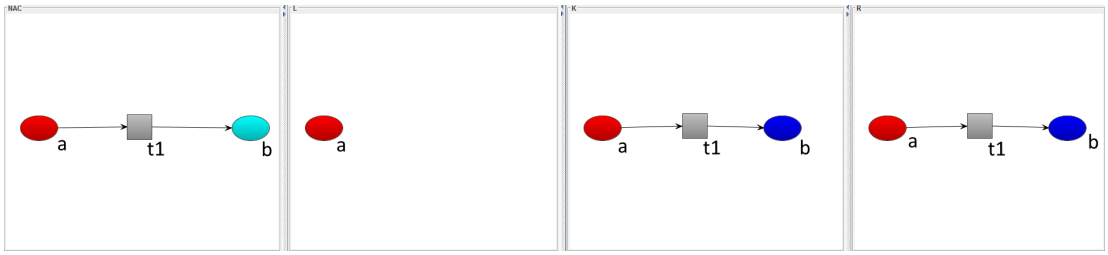

Abbildung 6.3: Regel AtoB

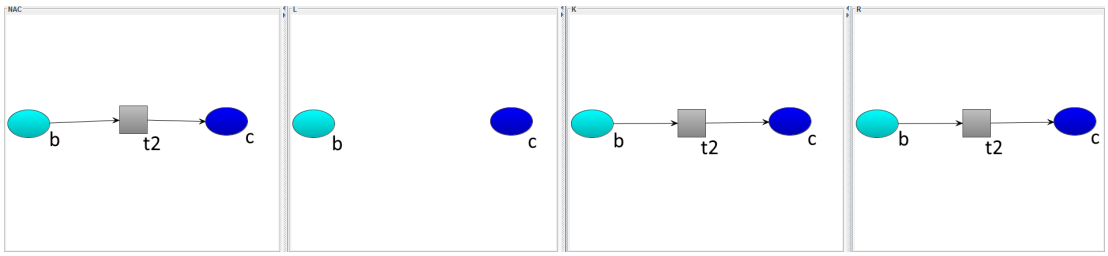

Abbildung 6.4: Regel BtoC

Test #	Was wird getestet
1	Ausführung einer Regel (atomarer Kontrollausdruck)
2	Rollback bei Scheitern einer notwendigen Regel
3	Sequentielle Ausführung von Regeln
4	Sequentielle Ausführung von aufeinander aufbauenden Regeln
5	Alternative Ausführung von Regeln mit und ohne Fehlerfall
6	Ausführung einer Regel, solange dies möglich ist (asLongAsPossible)
7	Rollback bei Scheitern einer teilweise ausgeführten asLongAsPossible Iteration
8	Zufällig häufige Ausführung von Regeln (randomNumberOfTimes)
9	Verbundtest mehrerer Operationen zur Demonstration der Ausdrucksmächtigkeit

Tabelle 6.1: Übersicht der Testfälle

einzelnen Testfälle und den jeweils getesteten Funktionen. Für jeden Test wurden Transformationseinheiten mit entsprechenden Kontrollausdrücken gebildet und durchgeführt. Anschließend wurde das resultierende Petrinetz überprüft. Bei der Testdurchführung wurde für jeden Test das Ausgangsnetz N, der verwendete Kontrollausdruck $\mathcal{C}$, die ReConNet-Konsolenausgabe sowie das resultierende Netz N' dokumentiert. Als Ausgangssituation für jeden Test wurde hierbei ein leeres Petrinetz verwendet.

Einfache Ausführung

Zunächst wurde die elementare Operation getestet: die Ausführung von Regeln.

Test 1:
$N = \{\}$
$\mathcal{C} = AddA$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [1508013317]
... execute rule: AddA .. success
.. execution successful
.. executed rules: [AddA]
```

$N' = \{a\}$

Test 2:
$N = \{\}$
$\mathcal{C} = AtoB$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [1096541101]
... execute rule: AtoB .. failed
... recover snapshot [1096541101]
.. execution failed
```

$N' = \{\}$

Die Ergebnisse beider Tests decken sich mit der erwarteten Funktionalität. Test 2 zeigt hierbei, dass die Wiederherstellung des Petrinetzes im Falle einer nicht anwendbaren Regel, welche jedoch zwingend notwendig ist, ordnungsgemäß funktioniert. Die Regel $AtoB$ kann aufgrund ihrer Vorbedingung nicht angewendet werden. Die Transformationseinheit gilt als gescheitert und der Zustand des Petrinetzes bleibt unverändert.

Sequenz

Bei der Sequenz wurde die sukzessive Ausführung der in Sequenz geschalteten Regeln getestet, unter Einhaltung der Reihenfolge von links nach rechts.

Test 3:
$N = \{\}$
$\mathcal{C} = AddA; AddC$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [893946527]
... execute rule: AddA .. success
... execute rule: AddC .. success
.. execution successful
.. executed rules: [AddA, AddC]
```

$N' = \{a, c\}$

Test 4:
$N = \{\}$
$\mathcal{C} = AddA; AtoB$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [451086392]
... execute rule: AddA .. success
... execute rule: AtoB .. success
.. execution successful
.. executed rules: [AddA, AtoB]
```

$N' = \{a \rightarrow t1 \rightarrow b\}$

Die Tests zeigen, dass die Regeln in der korrekten Reihenfolge angewendet werden, welche durch den Kontrollausdruck vorgegeben wird. Test 4 zeigt dabei, dass Regel $AddA$ die Vorbedingung für Regel $AtoB$ herstellt, sodass diese ebenfalls angewendet werden kann.

Alternative

Bei der Alternative wurde die nichtdeterministische Ausführung von alternativen Regeln getestet. Zur Überprüfung der Funktionalität wurde die dargestellte Transformationseinheit mehrmals durchgeführt.

Test 5:
$N = \{\}$
$\mathcal{C} = AddA; AddC; ((AtoB; BtoC)|(BtoC; AtoB))$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [146496048]
... execute rule: AddA .. success
... execute rule: AddC .. success
... execute rule: BtoC .. failed
... recover snapshot [146496048]
.. execution failed
```

$N' = \{\}$

ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [331112727]
... execute rule: AddA .. success
```

```
... execute rule: AddC .. success
... execute rule: AtoB .. success
... execute rule: BtoC .. success
.. execution successful
.. executed rules: [AddA, AddC, AtoB, BtoC]
```

$N' = \{a \rightarrow t1 \rightarrow b \rightarrow t2 \rightarrow c\}$

Wie in den Ausgaben zu sehen ist, kann die gegebene Transformationseinheit entweder korrekt durchgeführt werden oder scheitern. In der oberen Ausgabe wurde die Alternative *(BtoC;AtoB)* gewählt. Beide Regeln bauen jedoch aufeinander auf, weshalb die Reihenfolge von elementarer Bedeutung ist. Regel $BtoC$ kann in diesem Fall nicht als erstes ausgeführt werden, da noch keine Stelle b im Netz vorhanden ist. Die Transformationseinheit gilt somit als gescheitert. In der unteren Ausgabe wurde die andere Alternative *(AtoB;BtoC)* gewählt. Regel $AtoB$ sorgt hierbei für notwendigen Zustand des Netzes und die nachfolgende Regel $BtoC$ kann angewendet werden. Die Transformationseinheit wurde dementsprechend erfolgreich durchgeführt.

asLongAsPossible

Beim *asLongAsPossible*-Operator wurde die iterative Ausführung von Regeln getestet. Der Operator führt einen (Teil-)Ausdruck solange aus, wie es möglich ist. Die maximale Anzahl der Iterationen kann jedoch mit einem Parameter eingeschränkt werden.

Test 6:
$N = \{\}$
$\mathcal{C} = AddA!$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [1255147163]
... create snapshot [476491304]
... execute rule: AddA .. success
... create snapshot [1796060968]
... execute rule: AddA .. success
... create snapshot [918930600]
... execute rule: AddA .. success
... create snapshot [383878828]
... execute rule: AddA .. success
... create snapshot [1807134726]
... execute rule: AddA .. failed
... recover snapshot [1807134726]
.. execution successful
```

```
.. executed rules: [AddA, AddA, AddA, AddA]
```

$N' = \{a, a, a, a\}$

Wie der Ausgabe zu entnehmen ist, wurde fünf Mal versucht, Regel $AddA$ anzuwenden. Die fünfte Anwendung schlug jedoch fehl, da die negative Anwendungsbedingung von $AddA$ die Anwendung verhinderte. Die Transformationseinheit gilt nichtsdestotrotz als erfolgreich durchgeführt. Der *asLongAsPossible*-Operator funktioniert daher wie spezifiziert. Weiterhin wurde die Implementierung getestet, die maximale Anzahl an Iterationen des *asLongAsPossible*-Operators über einen Parameter zu spezifizieren. Bei einem Parameterwert von 3 wurde Regel $AddA$ nur drei Mal angewendet. Die Implementierung funktioniert daher wie erwartet. Durch die Spezifikation der maximalen Anzahl von Iterationen können eventuelle endlose Transformationen verhindert werden

Test 7:
$N = \{\}$
$\mathcal{C} = (AddC; AddA)!$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [873991221]
... create snapshot [345126068]
... execute rule: AddC .. success
... execute rule: AddA .. success
... create snapshot [441428423]
... execute rule: AddC .. success
... execute rule: AddA .. success
... create snapshot [1159231905]
... execute rule: AddC .. success
... execute rule: AddA .. success
... create snapshot [931906750]
... execute rule: AddC .. success
... execute rule: AddA .. success
... create snapshot [375020648]
... execute rule: AddC .. success
... execute rule: AddA .. failed
... recover snapshot [375020648]
.. execution successful
.. executed rules: [AddC, AddA, AddC, AddA, AddC, AddA, AddC, AddA]
```

$N' = \{a, a, a, a, c, c, c, c\}$

Die Transformationseinheit von Test 7 testet den Wiederherstellungsmechanismus bei Ver-

wendung des *asLongAsPossible*-Operators. Wie der Ausgabe zu entnehmen ist, wird vor jeder Iteration ein Snapshot erstellt. In der fünften Iteration schlägt hierbei die Anwendung von Regel $AddA$ fehl, da die negative Anwendungsbedingung nicht erfüllt ist. Anhand der Ausgabe kann man ebenfalls erkennen, dass Regel $AddC$ in dieser Iteration bereits angewendet wurde. Die Iteration muss dementsprechend als Ganzes rückgängig gemacht werden. Hierfür wird der zuvor erstellte Snapshot wiederhergestellt. Die Transformationseinheit gilt als erfolgreich und das resultierende Netz umfasst die Stellen $\{a, a, a, a, c, c, c, c\}$. Die Wiederherstellung von Snapshots beim *asLongAsPossible*-Operator funktioniert daher wie vorgesehen.

randomNumberOfTimes

Beim *randomNumberOfTimes*-Operator wurde ebenfalls die iterative Ausführung von Regeln getestet. Die Anzahl der durchzuführenden Iterationen ist hierbei jedoch zufällig und wird aus einer Zahl zwischen 0 und x ausgewählt, wobei x als Parameter übergeben wird. Der Operator führt jedoch maximal so viele Durchläufe durch, wie möglich sind (analog zum Abbruchverhalten des *asLongAsPossible*-Operators).

Test 8:
$N = \{\}$
$\mathcal{C} = AddA^*$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [949634985]
... create snapshot [1240694658]
... execute rule: AddA .. success
... create snapshot [553102736]
... execute rule: AddA .. success
... create snapshot [620227185]
... execute rule: AddA .. success
.. execution successful
.. executed rules: [AddA, AddA, AddA]
```

$N' = \{a, a, a\}$

ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [204176749]
... create snapshot [1980845412]
... execute rule: AddA .. success
... create snapshot [675227477]
... execute rule: AddA .. success
```

```
... create snapshot [1479318874]
... execute rule: AddA .. success
... create snapshot [915349976]
... execute rule: AddA .. success
... create snapshot [659561642]
... execute rule: AddA .. failed
... recover snapshot [659561642]
.. execution successful
.. executed rules: [AddA, AddA, AddA, AddA]
```

$N' = \{a, a, a, a\}$

Anhand der Ausgabe ist zu erkennen, dass nach mehrfacher Ausführung der Transformationseinheit der Ausdruck unterschiedlich häufig durchlaufen wurde. Der Test erfolgte hierbei mit einem Parameterwert von 10, welcher die obere Grenze der Zufallszahl festlegt. Anhand der ersten Ausgabe ist zu erkennen, dass drei Iterationen durchgeführt wurden. In der zweiten Ausgabe ist zu erkennen, dass mindestens fünf Iterationen durchgeführt werden sollten. Die negative Anwendungsbedingung von Regel $AddA$ hat die fünfte Iteration jedoch verhindert. Das Abbruchverhalten ist dementsprechend analog zum Abbruchverhalten des *asLongAsPossible*-Operators.

Test 9:
$N = \{\}$
$\mathcal{C} = AddA^*; AddC; AtoB!; BtoC!$
ReConNet Ausgabe:

```
.. execute transformation unit
... create snapshot [2139885754]
... create snapshot [72742169]
... execute rule: AddA .. success
... create snapshot [2112783892]
... execute rule: AddA .. success
... create snapshot [1172600034]
... execute rule: AddA .. success
... execute rule: AddC .. success
... create snapshot [290167103]
... execute rule: AtoB .. success
... create snapshot [116746607]
... execute rule: AtoB .. success
... create snapshot [1752408511]
... execute rule: AtoB .. success
... create snapshot [2435384]
... execute rule: AtoB .. failed
... recover snapshot [2435384]
... create snapshot [1952300047]
```

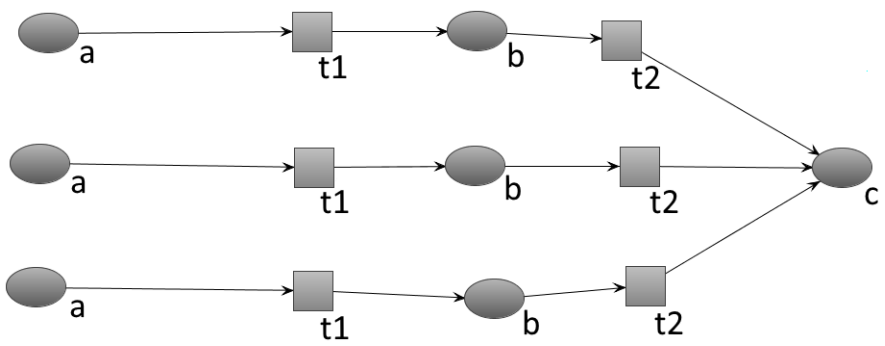

Abbildung 6.5: Resultierendes Petrinetz von Test 9

```
... execute rule: BtoC .. success
... create snapshot [1110322423]
... execute rule: BtoC .. success
... create snapshot [1043773937]
... execute rule: BtoC .. success
... create snapshot [1112956923]
... execute rule: BtoC .. failed
... recover snapshot [1112956923]
.. execution successful
.. executed rules: [AddA, AddA, AddA, AddC, AtoB, AtoB, AtoB, BtoC, BtoC, BtoC]
```

$$N' = \{a_1 \to t1_1 \to b_1 \to t2_1 \to c,$$
$$a_2 \to t1_2 \to b_2 \to t2_2 \to c,$$
$$a_3 \to t1_3 \to b_3 \to t2_3 \to c\}$$

Der obige Test zeigt schließlich den Verbund mehrerer Operatoren und wie diese zur Bildung einer Transformationseinheit herangezogen werden können. Zunächst wird beliebig häufig Regel $AddA$ angewandt. Anschließend wird ein Mal Regel $AddC$ angewandt. Darauf folgend werden jeweils die Regeln $AtoB$ und $BtoC$ angewandt, so oft dies möglich ist. Die hinzugefügten Stellen $a_1 ... a_n$ werden somit über weitere Stellen und Transitionen mit der Stelle c verbunden. Das resultierende Netz der obigen Ausgabe ist in Abbildung 6.5 dargestellt. Das dargestellte Netz zeigt hierbei deutlich, dass durch die Kombination mehrerer Operatoren komplexe Transformationen auf Basis weniger Regeln gebildet werden können.

6.2 Testergebnis

Die Tests haben gezeigt, dass die Transformationseinheiten in ReConNet gemäß ihrer Definition funktionieren. Der rekursive Ansatz der Implementierung bietet hierbei den Vorteil, dass nur wenige elementare Fälle betrachtet werden müssen. Mit Hilfe der einzelnen Tests wurden diese elementaren Funktionen separat getestet. Hierdurch ist implizit gewährleistet, dass auch beliebig komplexe Kontrollausdrücke korrekt verarbeitet werden.

Neben den hier durchgeführten manuellen Tests kommt in ReConNet das JUnit-Framework zum Einsatz, mit dessen Hilfe die Funktionen des Programms automatisiert getestet werden [JUNb]. Zum jetzigen Zeitpunkt werden die Funktionen von ReConNet mit insgesamt 371 JUnit-Tests überprüft. Hierdurch wird sichergestellt, dass bereits vorhandene Implementierungen nicht beeinträchtigt werden, wenn zusätzliche Funktionen oder Anpassungen implementiert werden. Zum automatisierten Testen der Transformationseinheiten wurden weitere JUnit-Tests in ReConNet implementiert. Bei der Implementierung der automatisierten Testfälle wurde sich an den manuellen Testfällen dieses Kapitels orientiert.

7 Evaluation

Die Tests haben gezeigt, dass die Transformationseinheiten gemäß der Definition funktionieren. Nun gilt es zu klären, inwieweit Transformationseinheiten bei der Modellierung rekonfigurierbarer Petrinetze genutzt werden können. Im Rahmen einer vorausgegangenen Projektarbeit wurde eine Fallstudie entwickelt. Anhand der Fallstudie wurden unter anderem Kontrollstrukturen wie Stellen- und Transitionsnamen, negative Anwendungsbedingungen sowie Inhibitor-Kanten evaluiert. Die Fallstudie wird nun herangezogen, um den Mehrwert von Transformationseinheiten zu evaluieren.

ReConNet unterstützt zum jetzigen Zeitpunkt keine Inhibitor-Kanten. Des Weiteren ist die Layout-Engine von ReConNet nicht für derart große Netze optimiert. Aus diesem Grund wurde das Petrinetz der Fallstudie mit *Snoopy* entwickelt. Snoopy ist ein Tool zur Modellierung und Simulation von Petrinetzen [MRH12].

7.1 Fallstudie

7.1.1 Hintergrund

In der Hamburger Innenstadt ist zu den Hauptverkehrszeiten an diversen Orten mit Stau zu rechnen. Die Hochschule für Angewandte Wissenschaften Hamburg befindet sich am Berliner Tor und liegt somit in der Nähe eines stark befahrenen Knotenpunkts zur Innenstadt. In Abbildung 7.1 sind die Kreuzungen im Umkreis Berliner Tor dargestellt. Durch die hohe Verkehrsdichte kommt es hierbei regelmäßig zu dichtem Stau, der das gesamte Straßennetz um Berliner Tor lahm legt.

Im Rahmen der Fallstudie wurde das in Abbildung 7.1 dargestellte System aus Kreuzungen als rekonfigurierbares Petrinetz modelliert. Hierbei wurde evaluiert in wieweit Kontrollstrukturen bei der Modellierung herangezogen werden können. Im Hinblick auf die Rekonfguration wurden Regeln definiert, welche die unterschiedlichen Verkehrssituationen zu bestimmten

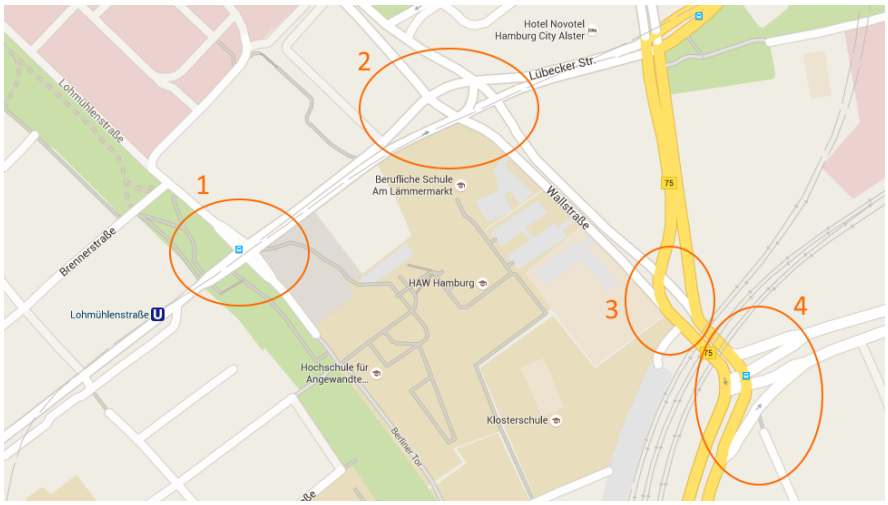

Abbildung 7.1: Kreuzungen im Bereich der HAW Hamburg

Tageszeiten abbilden. Hierbei wurde zum einen das Verkehrsaufkommen rekonfguriert und zum anderen die Adaption der Ampelsysteme auf die neue Verkehrssituation.

7.1.2 Petrinetz

Die in Abbildung 7.1 dargestellten Kreuzungen rund um Berliner Tor wurden in einem großen Petrinetz modelliert. Dies ist wichtig, um das Zusammenspiel aller Kreuzungen analysieren zu können. Der Übersicht halber sind die Kreuzungen separat in den Abbildungen 7.2, 7.3, 7.4 und 7.5 dargestellt. Eine Kreuzung unterliegt dabei mehren Ampelphasen, welche den Verkehr aus den jeweiligen Richtungen steuern. Die jeweiligen Stellen *x_to_y* bilden die Übergänge zwischen den einzelnen Kreuzungen. Der Straßenverlauf zwischen den Kreuzungen wurde dabei nicht weiter betrachtet. Es wird angenommen, dass ein Autofahrer nach einer Kreuzung auf eine beliebige Spur zur nächsten Kreuzung wechseln kann.

Die erste Kreuzung ist in Abbildung 7.2 dargestellt. Sie verfügt über vier Ampeln, welche in zwei Ampelphasen für den Verkehr der Richtungen Nordost ↔ Südwest sowie Nordwest ↔ Südost geschaltet werden. Die jeweiligen Ampelphasen wurden mit Hilfe eines separaten Netzes modelliert (s. Abbildung 7.2, unten). Das Netz hat die beiden Stellen *k1p1* und *k1p2*. Die Token auf diesen Stellen sind Voraussetzung für das Überqueren der Kreuzung in der jeweiligen Richtung. In der ersten Phase befinden sich 50 Token auf der Stelle *k1p1*. Dementsprechend können 50 Autofahrer die Kreuzung in Richtung Nordost ↔ Südwest überqueren. Haben 50 Autofahrer die Kreuzung überquert, schaltet die Ampel in die nächste Phase. Dies wird mit

Hilfe einer Inhibitor-Kante realisiert. Die Inhibitor-Kante setzt voraus, dass auf der jeweiligen Stelle kein Token vorhanden ist. Das Netz schaltet demnach über die Stelle *k1p1p2Uebergang* in die nächste Ampelphase. Hierbei liegen nun 50 Token auf der Stelle *k1p2*. Es herrscht also ein Verhältnis von 50:50 im Hinblick auf die überquerenden Autofahrer der jeweiligen Richtungen.

Das Verhalten von Linksabbiegern wurde mit Hilfe von Inhibitor-Kanten realisiert. Autofahrer können beispielsweise von der Stelle *k1SuedWestIn1* links abbiegen und das System über die Stelle *k1NordWestOut1* verlassen. Voraussetzung hierfür ist, dass kein Auto der Gegenfahrbahn geradeaus oder rechts abbiegen will. Die Inhibitor-Kanten von den Stellen *k1NordOstIn1*, *k1NordOstIn2 k1NordOstIn3* in die Linksabbieger-Transition stellen dies sicher.

Der Zufluss von Autofahrern in das Kreuzungssystem erfolgt über Transitionen ohne Vorbereich bei den Stellen *k1NordWestFlussIn1*, *k1SuedWestFlussIn1* und *k1SuedWestFlussIn2* sowie *k1SuedOstFlussIn1*. Das Verlassen des Systems ist über die Transitionen ohne Nachbereich bei den Stellen *k1NordWestOut1*, *k1SuedWestOut1* und *k1SuedWestOut2* sowie *k1SuedOstOut1* möglich. Kreuzung 1 steht über die Stellen *k1_to_k2* und *k2_to_k1* in Verbindung mit Kreuzung 2.

Die weiteren Kreuzungen wurden nach dem selben Schema modelliert.

7.1.3 Regeln

Für die Simulation verschiedener Verkehrsaufkommen wurden Regeln definiert. Das Verkehrsaufkommen des Systems wird über die jeweiligen Transitionen gesteuert, welche für den Zufluss von Autofahrern in das System verantwortlich sind. Weiterhin wurden Regeln definiert, welche die Ampelphasen der Kreuzungen anpassen, um dem Verkehrsaufkommen der jeweiligen Zuflussrichtungen gerecht zu werden.

Das Petrinetz wurde so modelliert, dass der Zufluss der einfahrenden Autos durch unterschiedliche Kantengewichte konfiguriert werden kann. Auf diese Weise kann der Zufluss aller Einfahrten mit Regeln rekonfiguriert werden. Um auf das veränderte Verkehrsaufkommen zu reagieren, müssen gegebenenfalls die Ampelphasen der Kreuzungen angepasst werden. Dies erfolgt durch rekonfiguration der Teilnetze für die Ampelphasen.

In der Fallstudie wurden hierbei die Verkehrssituationen zu den Tageszeiten *morgens*, *mittags* und *abends* betrachtet. Beim Zufluss wird zwischen normalem, erhöhtem und starkem Ver-

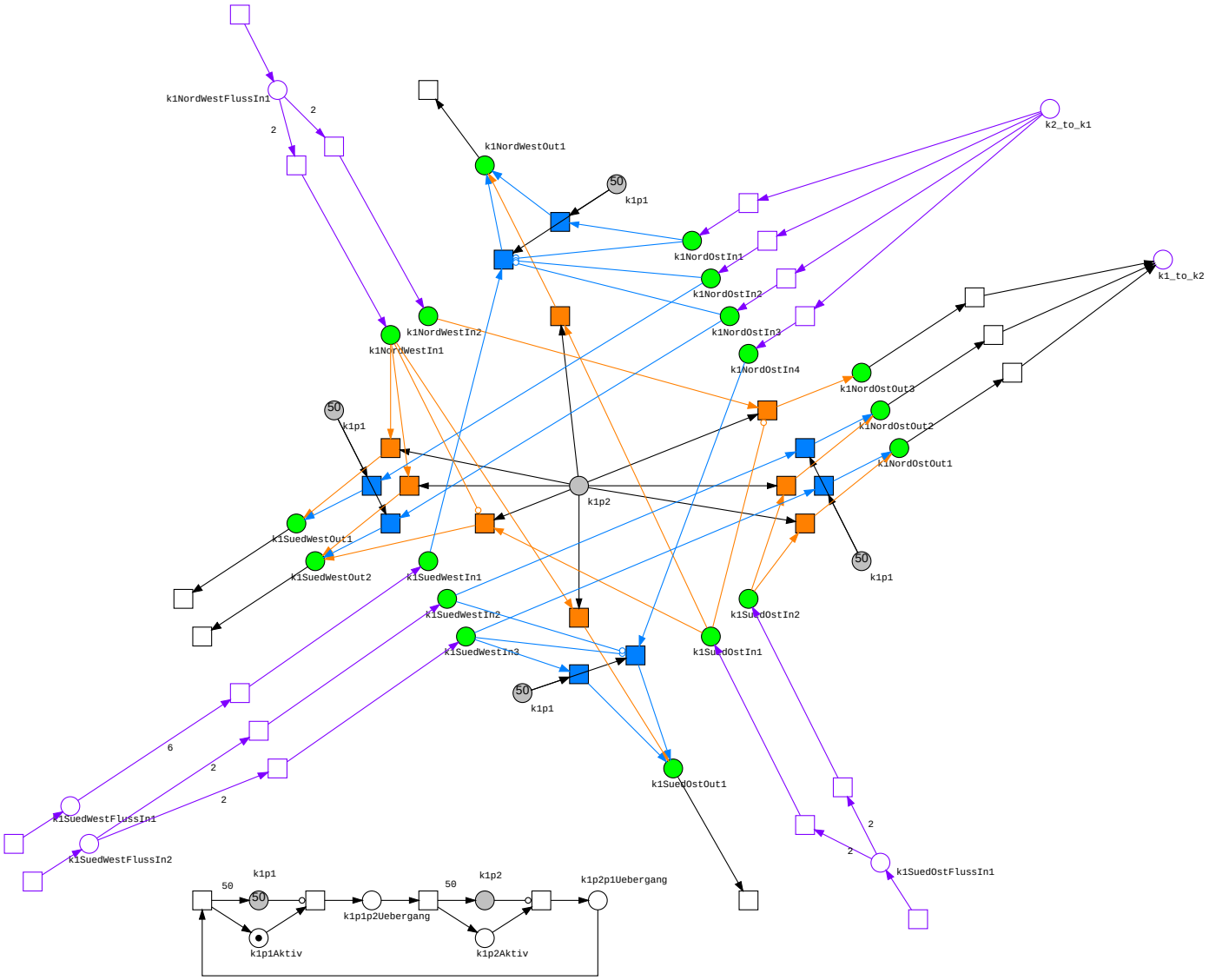

Abbildung 7.2: Petrinetz: Kreuzung 1

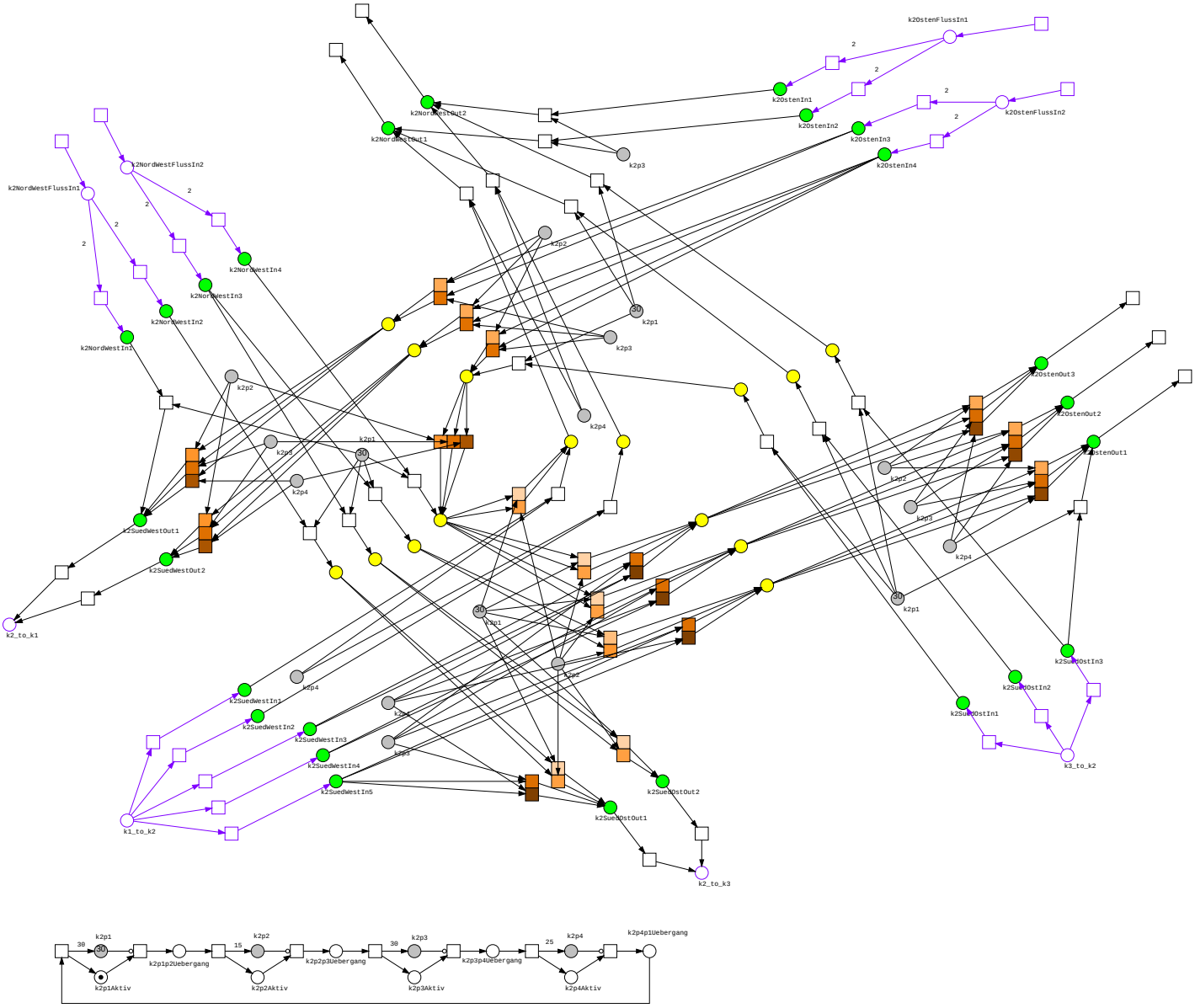

Abbildung 7.3: Petrinetz: Kreuzung 2

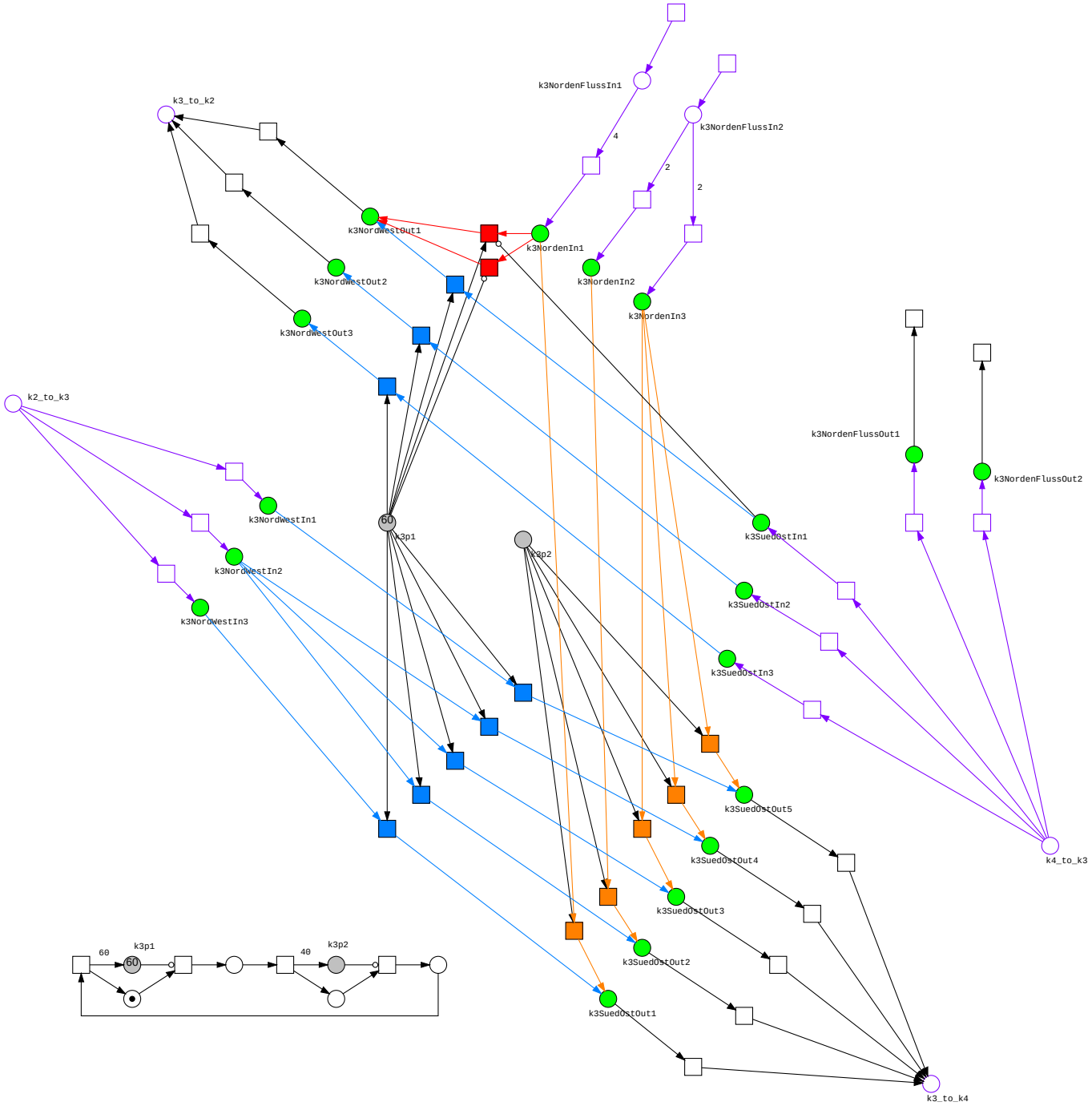

Abbildung 7.4: Petrinetz: Kreuzung 3

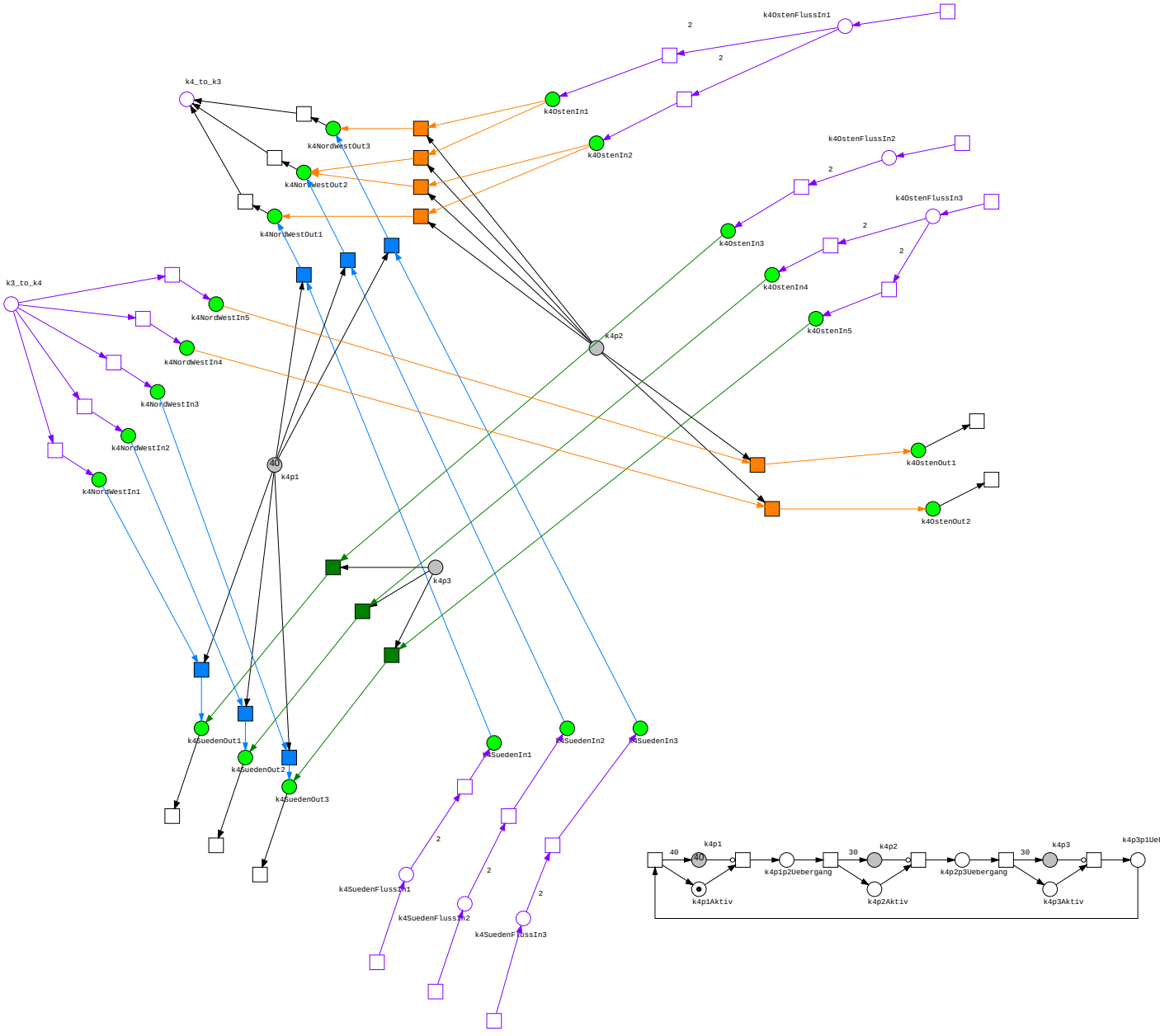

Abbildung 7.5: Petrinetz: Kreuzung 4

Zustand	Zufluss Nordwest	Zufluss Südost	Zufluss Südwest	Phasenverhältnis
morgens	erhöht	normal	normal	50:50
mittags	erhöht	erhöht	erhöht	30:70

Tabelle 7.1: Konfiguration von Kreuzung 1 in den Zuständen *morgens* und *mittags*

kehrsaufkommen unterschieden. Bei dem Verkehrsaufkommen wurde sich an aktuellen Google Maps Verkehrsdaten zu den relevanten Uhrzeiten orientiert (9:00 Uhr, 11:30 Uhr, 18:00 Uhr). Die Ampelphasen wurden dementsprechend gewählt, um dem jeweiligen Verkehrsaufkommen gerecht zu werden.

In den Abbildungen 7.6 und 7.7 sind beispielhaft die benötigten Regeln für Kreuzung 1 dargestellt, welche beim Zustandswechsel von *morgens* $\rightarrow$ *mittags* angewandt werden müssen. Abbildung 7.6 zeigt die Regel zur Rekonfiguration des Verkehrsaufkommens von Kreuzung 1. Die linke Seite der Regel setzt voraus, dass das Verkehrsaufkommen aus den Richtungen Südwest und Südost normal ist. Dies wird über die Kantengewichtung von 1 zu den jeweiligen Stellen wie bspw. *k1SuedWestIn1* ausgedrückt. Die Voraussetzung ist im Zustand *morgens* erfüllt (s. Tabelle 7.1). Um den Zufluss der Autofahrer zu erhöhen, wird das Kantengewicht in diesem Fall auf 2 gesetzt, was in der rechten Seite der Regel zu sehen ist. Der Zufluss an Autofahrern aus Richtung Nordwest bleibt beim Übergang *morgens* $\rightarrow$ *mittags* unverändert und muss dementsprechend nicht rekonfiguriert werden.

In Abbildung 7.7 ist die zugehörige Regel zur Rekonfiguration der Ampelphasen dargestellt. Die Regel sorgt für eine Umverteilung der Verhältnisse der jeweiligen Ampelphasen. Dies wird über die Rekonfiguration der Kantengewichte zu den Stellen *k1p1* bzw. *k1p2* realisiert. Wie der Tabelle 7.1 zu entnehmen ist, soll der Verkehr *Nordwest* $\leftrightarrow$ *Südost* im Zustand *mittags* priorisiert werden. Für den Verkehr dieser Richtungen ist Phase 2 verantwortlich. Die Regel sorgt demnach für eine Rekonfiguration des Verhältnisses von 50:50 zu 30:70 (s. Abbildung 7.7, L und R). Phase 2 wird somit höher priorisiert als Phase 1. Weiterhin verfügt die Regel über zwei negative Anwendungsbedingungen. Diese sollen sicherstellen, dass eine Rekonfiguration nicht während einer aktiven Ampelphase durchgeführt wird. Befindet sich die Kreuzung in Ampelphase 1 oder 2, liegt auf der Stelle *k1p1Aktiv* bzw. *k1p2Aktiv* ein Token. Liegt einer dieser beiden Fälle vor, kann entweder ein Morphismus von *NAC_1* oder von *NAC_2* zum Netz gebildet werden und die Transformation wird verhindert.

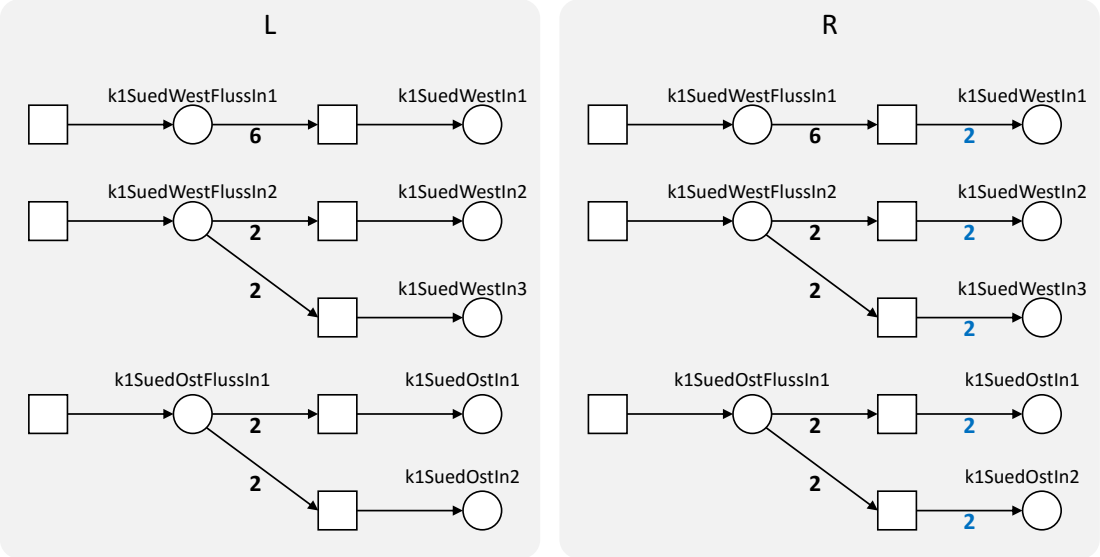

Abbildung 7.6: Regel zur Rekonfiguration des Verkehrsaufkommen bei Kreuzung 1

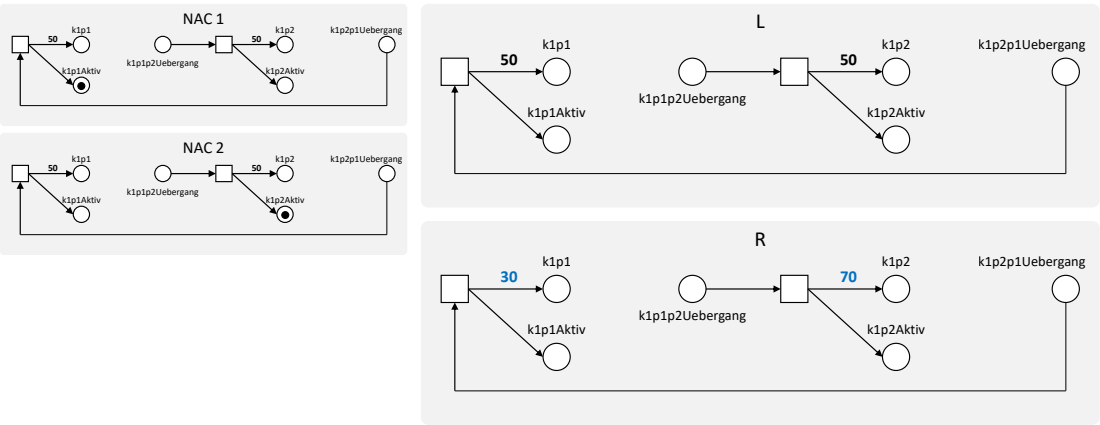

Abbildung 7.7: Regel zur Rekonfiguration der Ampelphasen bei Kreuzung 1

7.2 Transformationseinheiten

Die Grundidee der Fallstudie wurde im vorherigen Abschnitt skizziert. Nun gilt es zu evaluieren, inwieweit Transformationseinheiten in dieser Fallstudie eingesetzt werden können und welchen allgemeinen Mehrwert sie bei der Modellierung rekonfigurierbarer Petrinetze bieten.

7.2.1 Modularisierung und Bündelung

Der wohl größte Mehrwert von Transformationseinheiten liegt darin, dass viele kleine Regeln zu einer komplexen Transformation gebündelt werden können. Auf diese Weise können Regeln auf ihre Kernaufgaben reduziert werden, wodurch die Wiederverwendbarkeit gesteigert wird. Die in Abbildung 7.6 dargestellte Regel zur Rekonfiguration des Verkehrsaufkommens von Kreuzung 1 ist sehr unflexibel, da alle Zuflüsse der Kreuzung gleichzeitig rekonfiguriert werden. Hier ist es wünschenswert, dass die jeweiligen Zuflüsse separat rekonfiguriert werden können. Die Regel wird dementsprechend in drei kleine Regeln modularisiert (s. Abbildung 7.8). Diese werden anschließend mit dem Kontrollausdruck:

$$\begin{aligned}\mathcal{C} = {} & k1SuedWestFlussIn1NormalZuHoch; \\ & k1SuedWestFlussIn2NormalZuHoch; \\ & k1SuedOstFlussIn1NormalZuHoch\end{aligned}$$

zu einer Transformationseinheit gebündelt. Auf diese Weise kann die Kreuzung im Ganzen rekonfiguriert werden und die Wiederverwendbarkeit der einzelnen Regeln wird gesteigert.

Das komplette Netz verfügt hierbei über 4 Kreuzungen mit insgesamt 16 separaten Einfahrten, dessen Zuflüsse rekonfiguriert werden können. Weiterhin müssen die Ampelphasen der einzelnen Kreuzungen ebenfalls rekonfiguriert werden. Die manuelle Transformation des gesamten Netzes bei einem Tageszeitwechsel wäre hierbei nicht mehr praktikabel. Transformationseinheiten bieten deshalb einen hohen praktischen Nutzen, da die einzelnen Regeln zu komplexen Transformationen gebündelt werden können. Durch die Modularisierung können die Regeln auch in anderen Transformationseinheiten wiederverwendet werden, wodurch etwaige Redundanzen vermieden werden.

7.2.2 Nichtdeterministische Transformationen

Die Rekonfiguration von Petrinetzen erfolgt generell nichtdeterministisch unter Berücksichtigung aller Regeln. Um bei einer Simulation zufällige Ereignisse der echten Welt gezielt abbilden

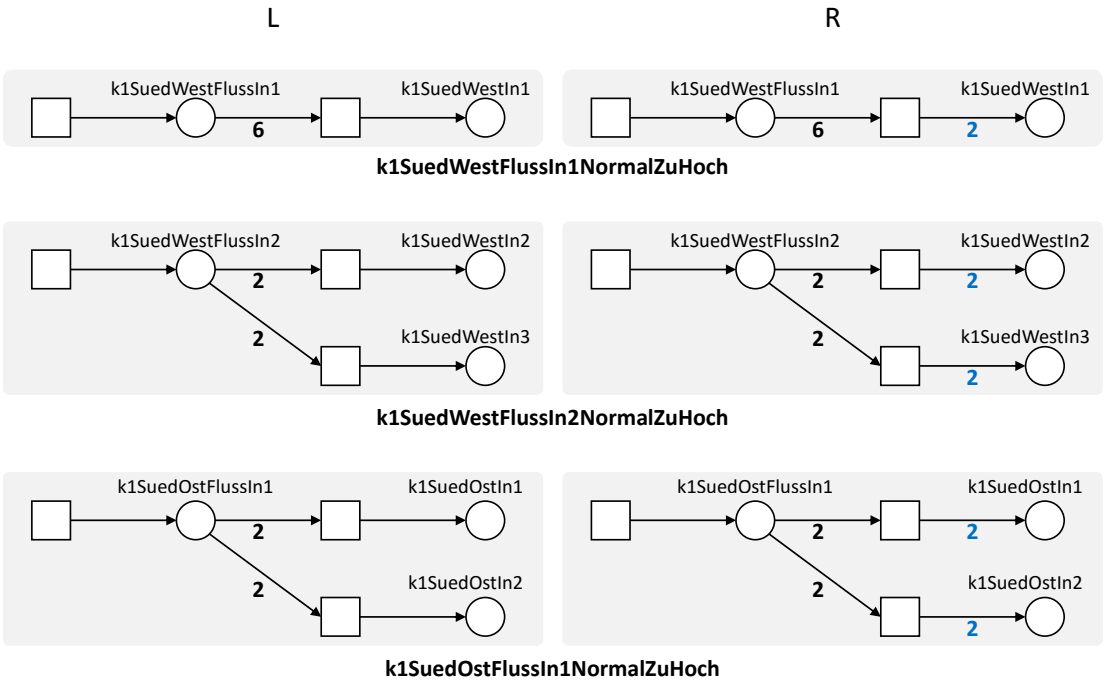

Abbildung 7.8: Regeln zur Rekonfiguration des Verkehrsaufkommen bei Kreuzung 1

zu können, kann es jedoch notwendig sein, nur einen Teil aller Regeln für nichtdeterministische Transformationen zu betrachten. Der Alternativ-Operator von Transformationseinheiten kann hierfür verwendet werden. Im Hinblick auf die Fallstudie ist ein möglicher Einsatz hierfür die Abbildung von Auffahrunfällen und den damit einhergehenden Sperrungen von Fahrspuren. In Abbildung 7.9 sind zwei Regeln für diesen Anwendungsfall dargestellt. Die Regeln beziehen sich hierbei auf die nordwestliche Einfahrt in Kreuzung 3 (s. Abbildung 7.4). Der folgende Kontrollausdruck sorgt für einen Auffahrunfall auf einer der beiden Fahrspuren:

$$\mathcal{C} = CrashNordWestIn1 | CrashNordWestIn3$$

Bei Durchführung der Transformationseinheit wird eine der beiden Fahrspuren durch Entfernen der jeweiligen Transitionen gesperrt. Durch die neue Transition *umleiten* wird der verbliebene Verkehr der gesperrten Fahrspur auf die mittlere Fahrspur umgeleitet. Mit Hilfe des Alternativ-Operators können somit Transformationen in nichtdeterministischer Weise durchgeführt werden, um zufällige Ereignisse der echten Welt abzubilden.

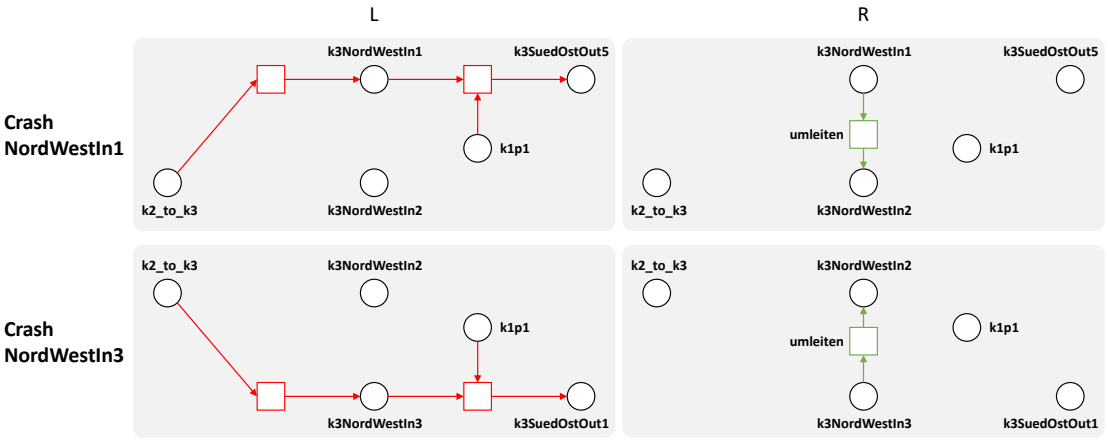

Abbildung 7.9: Nichtdeterministische Transformationen

7.2.3 Transformationen mit Wahrscheinlichkeiten

Ein weiterer Vorteil von Transformationseinheiten ist die Möglichkeit, Wahrscheinlichkeiten für Rekonfigurationen abbilden zu können. Wahrscheinlichkeiten können mit Hilfe des Alternativ-Operators realisiert werden. Bezogen auf das Fallbeispiel können Wahrscheinlichkeiten dafür genutzt werden, um beispielsweise den Zufluss von Autofahrern mit bestimmten Wahrscheinlichkeiten zu beeinflussen. Als Beispiel sollen hier ebenfalls die in Abbildung 7.8 dargestellten Regeln dienen. Im Normalfall werden alle Regeln sequenziell Ausgeführt um den Zufluss der Autofahrer auf ein bestimmtes Level zu rekonfigurieren. Mit Hilfe des Alternativ-Operators kann der Zufluss mit Wahrscheinlichkeiten rekonfiguriert werden. Ein möglicher Kontrollausdruck hierfür ist:

$$\mathcal{C} = ((R1; R2; R3)|(R1; R2))|(R1; R2; R3).$$

mit

$$R1 = k1SuedWestFlussIn1NormalZuHoch$$
$$R2 = k1SuedWestFlussIn2NormalZuHoch$$
$$R3 = k1SuedOstFlussIn1NormalZuHoch$$

Der vereinfachte Parsebaum für diesen Kontrollausdruck ist in Abbildung 7.10 dargestellt. Die Zahlen an den Kanten beschreiben die Wahrscheinlichkeiten für die Auswahl der jeweiligen Alternative. Multipliziert man nun die Wahrscheinlichkeiten der einzelnen Pfade erhält man

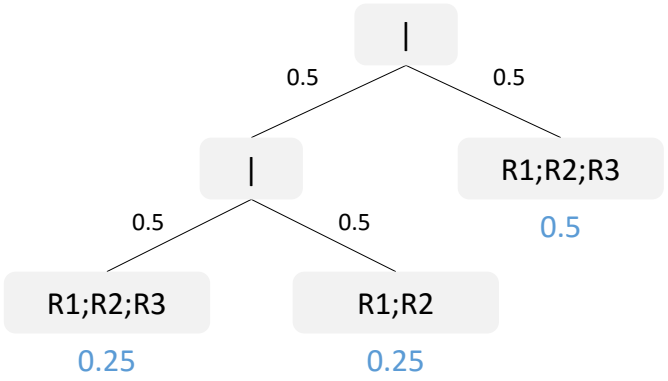

Abbildung 7.10: Wahrscheinlichkeiten mit dem Alternativ-Operator

die Wahrscheinlichkeit für den jeweiligen Knoten. Die gesamte Wahrscheinlichkeit für den Ausdruck $(R1; R2; R3)$ ist hierbei die Summe der einzelnen Wahrscheinlichkeiten für diesen Ausdruck. Dadurch ergibt sich eine Wahrscheinlichkeit von 75%, dass Ausdruck $(R1; R2; R3)$ ausgeführt wird. Ausdruck $(R1; R2)$ wird hingegen nur mit einer Wahrscheinlichkeit von 25% ausgeführt. Bezogen auf das Fallbeispiel wird dementsprechend der Zufluss mit einer Wahrscheinlichkeit von 75% aus den Richtungen Südwesten und Südosten erhöht. Mit einer Wahrscheinlichkeit von 25% erhöht sich lediglich der Zufluss aus Südwesten.

7.2.4 Optionale Transformationen

Wenn bei einer Transformationseinheit viele Regeln in Sequenz durchgeführt werden sollen, muss sichergestellt sein, dass die Vorbedingungen der Regeln auch erfüllt sind. Je mehr Regeln hierbei verwendet werden, desto wahrscheinlicher ist die Tatsache, dass eine der Regel aufgrund einer verletzten Vorbedingung nicht ausgeführt werden kann. Die Transformationseinheit wäre somit als Ganzes gescheitert. Mit Hilfe des *asLongAsPossible*-Operators kann man dieses Verhalten etwas aufweichen, indem jede Regel der Transformationseinheit mit ! versehen wird. Die einzelnen Regeln gelten somit als optional und führen bei Scheitern nicht zum Scheitern der gesamten Transformationseinheit.

Ein möglicher Anwendungsfall hierfür ist eine fehlertolerante Rekonfiguration der Zuflüsse von Autofahrern ins System. In Abbildung 7.11 sind hierfür zwei Regeln dargestellt, welche den Zufluss in Kreuzung $k1$ aus Richtung Südwest erhöhen. Bei der klassischen Transformationseinheit mit Sequenz muss der Benutzer sicherstellen, dass die Vorbedingungen der einzelnen Regeln erfüllt sind. Der folgende Kontrollausdruck weicht diese Bedingung auf:

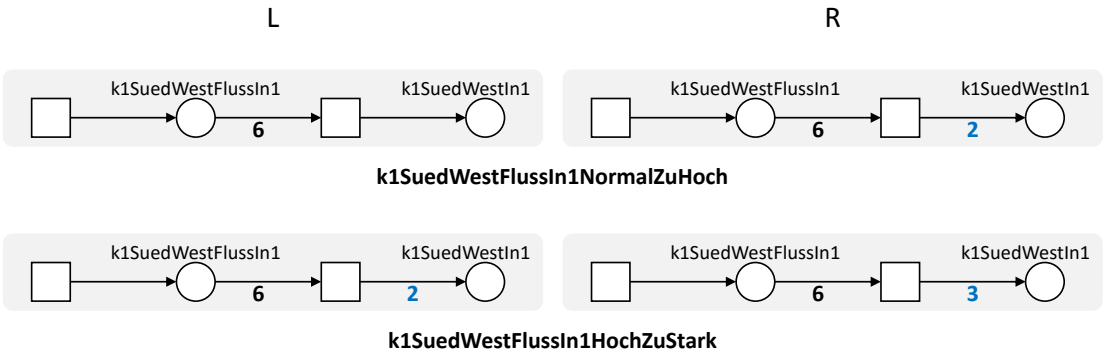

Abbildung 7.11: Optionale Transformationen

$$\mathcal{C} = k1SuedWestFlussIn1NormalZuHoch!;$$
$$k1SuedWestFlussIn1HochZuStark!$$

Durch den *asLongAsPossible*-Operator ! wird die Ausführung der beiden Regeln optional. Wenn der Zufluss *normal* ist, wird dieser mit der ersten Regel auf *erhöht* rekonfiguriert. Wenn dies nicht der Fall ist, wird die erste Regel übersprungen. Mit der zweiten Regel wird der Zufluss schließlich von *erhöht* auf *stark* rekonfiguriert. Ist der Zufluss zu Beginn der Transformationseinheit bereits *stark*, werden beide Regeln übersprungen und die Transformationseinheit hat keine weiteren Auswirkungen. Somit ist sichergestellt, dass die Transformationseinheit für alle möglichen Vorbedingungen erfolgreich durchlaufen wird.

Bei der Modellierung solcher Transformationen muss berücksichtigt werden, dass der *asLongAsPossible*-Operator einen Teilausdruck solange durchführt, wie es möglich ist. Soll eine optionale Transformation höchstens ein Mal durchgeführt werden, muss man sicherstellen, dass die erste Durchführung der Transformation eine weitere Durchführung verhindert. Dies kann unter anderem mit einer negativen Anwendungsbedingung realisiert werden. Weiterhin kann dies realisiert werden, indem gezielt Teile des Netzes rekonfiguriert werden, sodass bei einer weiteren Durchführung kein Morphismus $o : L \to N$ gebildet werden kann und hierdurch die erneute Anwendung der Regel verhindert wird.

7.2.5 Netzweite Transformationen

Wenn an vielen Stellen im Netz ähnliche Änderungen vorgenommen werden müssen, kann der *asLongAsPossible*-Operator verwendet werden, um diesen Vorgang zu optimieren. Ein möglicher Einsatzzweck hierfür ist die Unterstützung der Simulation durch die schnelle Präparation

des Petrinetzes mit definierten Simulationsparametern. Um dies zu ermöglichen muss das Petrinetz an den jeweiligen Stellen möglichst allgemein modelliert sein, damit Regeln global angewendet werden können.

Jede Kreuzung der Fallstudie verfügt über Transitionen ohne Vorbereich, welche als Einfahrten in das System dienen. Da die Transitionen keinen Vorbereich haben, können diese permanent schalten und sorgen hiermit für einen permanenten Zufluss an Autofahrern in das System. Für eine Simulation des Systems kann es jedoch auch notwendig sein, die Anzahl der einfahrenden Autos genau zu spezifizieren. Um dies zu ermöglichen, können dem Petrinetz Stellen hinzugefügt werden, welche als Wertehalter von Simulationsparametern fungieren. In Abbildung 7.12 ist ein Ausschnitt aus Kreuzung 4 dargestellt. Die Vorbereiche der Transitionen, die als Einfahrten in das System dienen, wurden jeweils um die Stellen a bzw. b erweitert. Des Weiteren wurden die Namen der Stellen des Nachbereichs entfernt, damit etwaige Regeln allgemeiner formuliert werden können. Mit Hilfe einer Transformationseinheit kann die Anzahl an Token auf diesen Stellen rekonfiguriert werden, um somit die Parameter für eine Simulation zu spezifizieren. In Darstellung 7.13 sind beispielhaft die hierfür notwendigen Regeln dargestellt. Mit Hilfe der dargestellten Regeln kann folgender Kontrollausdruck gebildet werden:

$$\mathcal{C} = InitA!; InitB!$$

Bei Ausführung der Transformationseinheit werden die Stellen a und b durch neue Stellen a und b mit einer definierten Anzahl an Token ersetzt. Der *asLongAsPossible*-Operator führt hierbei die Regeln solange aus, wie es möglich ist. Die negativen Anwendungsbedingungen der Regeln sorgen dafür, dass bereits initialisierte Stellen nicht erneut initialisiert werden. Mit Hilfe der Transformationseinheit werden jegliche Stellen a und b im Netz rekonfiguriert. Auf diese Weise können Stellen als Wertehalter von Simulationsparametern verwendet werden, welche komfortabel mit Transformationseinheiten initialisiert werden können. Zur Definition neuer Simulationsparameter muss lediglich die Anzahl der Token in den jeweiligen Regeln modifiziert werden. Um eine zufällige Verteilung von Token zu erreichen, kann alternativ auch der *randomNumberOfTimes*-Operator verwendet werden. Das Petrinetz kann somit ohne großen Aufwand mit beliebigen Parametern simuliert werden.

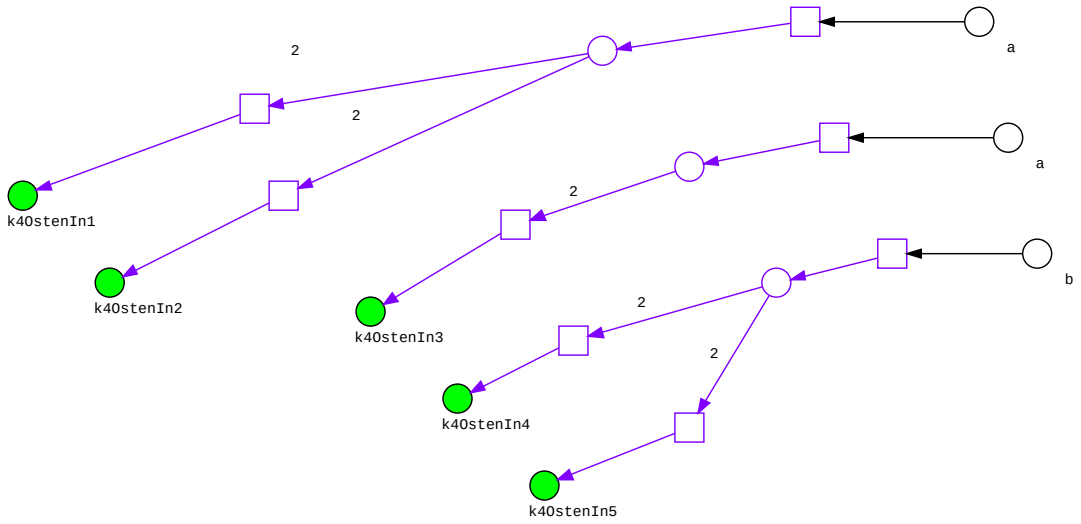

Abbildung 7.12: Modifiziertes Petrinetz bei Kreuzung 4

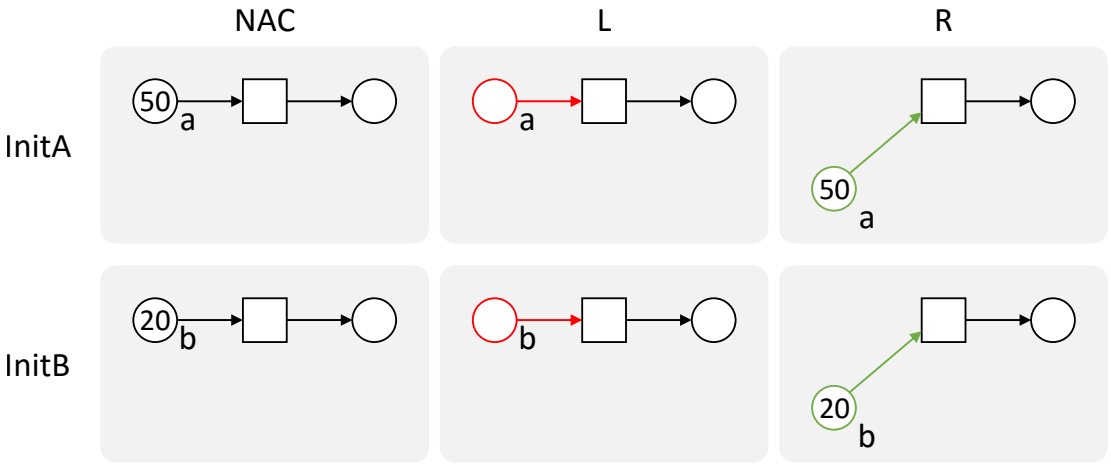

Abbildung 7.13: Regeln zur Initialisierung der Simulationsparameter

8 Schluss

8.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden Transformationseinheiten als weitere Kontrollstruktur in das Simulationstool ReConNet integriert. Anhand einer Fallstudie wurde anschließend evaluiert, wie diese für die Modellierung rekonfigurierbarer Petrinetze herangezogen werden können.

Zu Beginn der Arbeit wurden zunächst die theoretischen Grundlagen erarbeitet (Kapitel 2). Anschließend wurde das Konzept von Kontrollstrukturen erläutert und die zur Verfügung stehenden Kontrollstrukturen für rekonfigurierbare Petrinetze vorgestellt (Kapitel 3). Im Anschluss daran erfolgte die Analyse und Konzeption für die Implementierung der Transformationseinheiten in das Simulationstool ReConNet (Kapitel 4). Im darauffolgenden Kapitel wurde die durchgeführte Implementierung erläutert (Kapitel 5). Die Implementierung wurde anschließend auf Funktionalität getestet (Kapitel 6). Schließlich wurde anhand einer Fallstudie evaluiert, inwieweit Transformationseinheiten bei der Modellierung rekonfigurierbarer Petrinetze eingesetzt werden können (Kapitel 7).

Anhand der Fallstudie wurden diverse Einsatzmöglichkeiten für Transformationseinheiten in rekonfigurierbaren Petrinetzen aufgezeigt. Der offensichtliche Mehrwert von Transformationseinheiten liegt dabei in der Modularisierung komplexer Transformationen. Eine komplexe Transformation kann hierbei in viele Regeln modularisiert werden, welche anschließend mit Hilfe eines Kontrollausdrucks zu einer Transformationseinheit gebündelt werden. Dies steigert zum einen die Wiederverwendbarkeit einzelner Regeln und hilft dabei, die Modellierung monolithischer Regeln zu verhindern.

Die Abbildung zufälliger Ereignisse im rekonfigurierbaren Petrinetz kann mit Hilfe des Alternativ-Operators erfolgen. Weiterhin wurde gezeigt, dass der Alternativ-Operator dazu verwendet werden kann, Wahrscheinlichkeiten für Transformationen zu modellieren. Durch eine geeignete Klammerung der Teilausdrücke im Kontrollausdruck kann hierbei ein Wahrscheinlich-

keitsbaum aufgebaut werden. Eine freie Angabe der Wahrscheinlichkeiten ist jedoch nicht möglich, da die Wahrscheinlichkeit für einen Knoten im Wahrscheinlichkeitsbaum aufgrund der Semantik des Alternativ-Operators immer ein Vielfaches von 0.5 ist. Durch mehrfache Verwendung des selben Teilausdrucks können jedoch auch Zwischenwerte erreicht werden. Für einfache Anwendungsfälle reicht dies oftmals aus.

Einen großen Mehrwert bietet ebenfalls der *asLongAsPossible*-Operator. Unter Berücksichtigung einer möglichst allgemeinen Modellierung des Petrinetzes kann der Operator für netzweite Transformationen verwendet werden. Dies ist dann hilfreich, wenn an vielen Stellen des Netzes ähnliche Transformationen durchzuführen sind. Hierbei hat sich herausgestellt, dass der *asLongAsPossible*-Operator sehr gut mit negativen Anwendungsbedingungen harmoniert. Negative Anwendungsbedingungen können hierbei gezielt verwendet werden, um eine Abbruchbedingung für den *asLongAsPossible*-Operator zu modellieren. Weiterhin kann der Operator dazu verwendet werden, um optionale Transformationen zu realisieren. Die damit einhergehende Flexibilität ermöglicht die Modellierung fehlertoleranter Sequenzen von Transformationen.

Zusammenfassend ist zu sagen, dass Transformationseinheiten aufgrund ihrer vielseitigen Anwendungsmöglichkeiten einen hohen praktischen Nutzen haben und dadurch eine mächtige Kontrollstruktur für rekonfigurierbare Petrinetze darstellen.

8.2 Ausblick

Im Folgenden werden weitere Themen angesprochen, die für nachfolgende Arbeiten in Betracht gezogen werden können.

Strukturierte Transformationseinheiten

Anhand der Fallstudie wurden Einsatzmöglichkeiten für Transformationseinheiten dargestellt. Aus Gründen der Anschaulichkeit wurden hierbei nur Teile des Netzes betrachtet, die mit Hilfe der Transformationseinheiten rekonfiguriert wurden. Der Umfang der Kontrollausdrücke war deshalb noch überschaubar. Hätte man alle Kreuzungen bei der Modellierung einer Transformationseinheit zur Rekonfiguration der Tageszeit betrachtet, wäre der hierfür benötigte Kontrollausdruck deutlich komplexer gewesen. Eine Modularisierung auf Ebene der Transformationseinheiten wäre somit eine wünschenswerte Erweiterung. So könnten beispielsweise separate Transformationseinheiten für die einzelnen Kreuzungen gebildet werden, welche

anschließend in einer umfassenden Transformationseinheit importiert werden. Strukturierte Transformationseinheiten ermöglichen dieses Vorhaben und bilden somit im Kontext von ReConNet einen interessanten Gegenstand für weitere Betrachtungen.

Prioritätsoperator

Die Regeln einer Transformationseinheit werden in der jetzigen Form sukzessiv durchgeführt. Für bestimmte Problemstellungen kann es jedoch auch relevant sein, mehrere Regeln gleichzeitig zu betrachten. Um Prioritäten von Regeln als weitere Kontrollstruktur zu ermöglichen, könnte die Definition des Kontrollausdrucks um einen Prioritätsoperator erweitert werden. Ein Kontrollausdruck mit Prioritätsoperator könnte demnach wie folgt aussehen:

$$\mathcal{C} = (Regel_B > Regel_A > Regel_X)$$

Wird der Prioritätsoperator verwendet, müssen alle damit verbundenen Regeln überprüft werden. Sind gleichzeitig mehrere Regeln anwendbar, so wird die Regel mit der höchsten Priorität durchgeführt. Die Erweiterung der Transformationseinheiten in ReConNet um einen Prioritätsoperator wäre demnach sehr erstrebenswert, um die Ausdrucksmächtigkeit bei der Modellierung rekonfigurierbarer Petrinetze weiter zu steigern.

Literaturverzeichnis

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[Age74] Tilak Agerwala. A complete model for representing the coordination of asynchronous processes. *Hopkins Computer Research Report 32*, 1974. John Hopkins University.

[ANT] ANTLR - ANother Tool for Language Recognition. http://www.antlr.org. Letzter Zugriff: 24.07.2016.

[EHP+07] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Ulrike Prange, and Claudia Ermel. Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume 4546 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin Heidelberg, 2007.

[EHP09] Hartmut Ehrig, Frank Hermann, and Ulrike Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformations. *BEATCS*, pages 139–146, 2009.

[EP04] Hartmut Ehrig and Julia Padberg. Graph Grammars and Petri Net Transformations. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 496–536. Springer Berlin Heidelberg, 2004.

[GL81] H.J. Genrich and K. Lautenbach. Special Issue Semantics of Concurrent Computation System modelling with high-level Petri nets. *Theoretical Computer Science*, 13(1):109–135, 1981.

[Gso] Gson library. https://github.com/google/gson. Letzter Zugriff: 24.07.2016.

[Jen81] Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317–336, 1981.

[JUNa] JUNG - Java Universal Network/Graph Framework. http://jung.sourceforge.net. Letzter Zugriff: 24.07.2016.

[JUNb] JUnit Framework. http://junit.org. Letzter Zugriff: 24.07.2016.

[LO04] Marisa Llorens and Javier Oliver. Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. *IEEE Trans. Comput.*, 53(9):1147–1158, September 2004.

[Lud15] Melanie Luderer. *Control Conditions for Transformation Units - Parallelism, As-long-as-possible, and Stepwise Control.* Dissertation, Universität Bremen, 2015.

[MRH12] W Marwan, C Rohr, and M Heiner. *Petri nets in Snoopy: A unifying framework for the graphical display, computational modelling, and simulation of bacterial regulatory networks*, volume 804 of *Methods in Molecular Biology*, chapter 21, pages 409–437. Humana Press, 2012.

[Pad12] Julia Padberg. Abstract Interleaving Semantics for Reconfigurable Petri Nets. *ECEASST*, 51, 2012.

[Pad15] Julia Padberg. Reconfigurable Petri Nets with Transition Priorities and Inhibitor Arcs. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation*, volume 9151 of *Lecture Notes in Computer Science*, pages 104–120. Springer International Publishing, 2015.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

[PEHP08] Ulrike Prange, Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations in Reconfigurable Place/Transition Systems. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 96–113. Springer Berlin Heidelberg, 2008.

[PEOH12] Julia Padberg, Marvin Ede, Gerhard Oelker, and Kathrin Hoffmann. ReConNet: A Tool for Modeling and Simulating with Reconfigurable Place/Transition Nets. *ECEASST*, 54, 2012.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Dissertation, Universität Hamburg, 1962.

[PF11] Terence Parr and Kathleen Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[PH15] Julia Padberg and Kathrin Hoffmann. A Survey of Control Structures for Reconfigurable Petri Nets. *Journal of Computer and Communications*, 3(2):20–28, 2015.

[Rei85] Wolfgang Reisig. *Systementwurf mit Netzen.* Springer, 1985.

[Rei08] Alexander Rein. Reconfigurable Petri Systems with Negative Application Conditions. Diplomarbeit, Technische Universität Berlin, 2008.

[RPL+08] Alexander Rein, Ulrike Prange, Leen Lambers, Kathrin Hoffmann, and Julia Padberg. Negative Application Conditions for Reconfigurable Place/Transition Systems. *ECEASST*, 10, 2008.

[WK03] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. In *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag Berlin Heidelberg, 1st edition, 2003.

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

Hamburg, 4. August 2016 Christian Hoff