Diplomarbeit

Inkrementelle Interaktionsmodellierung mit farbigen Petri-Netzen - Formale Beschreibung und Implementierung

Jan Stückrath

30. März 2010

Erstgutachter: Prof. Dr. Wolfram Luther
Zweitgutachterin: Prof. Dr. Barbara König
Betreut durch: Dipl.-Inform. Benjamin Weyers

Universität Duisburg-Essen
Fachbereich Ingenieurwissenschaften
Abteilung Informatik und Angewandte Kognitionswissenschaft (INKO)
Fachgebiet „Computergraphik, Bildverarbeitung, Wissenschaftliches Rechnen“

Inhaltsverzeichnis

1 Einführung

Das Design von Benutzerschnittstellen ist ein nicht zu vernachlässigendes Gebiet der praktischen Informatik. Der beste Algorithmus kann durch eine schlecht konstruierte Benutzerschnittstelle unbenutzbar werden. Daher ist das Design von Benutzerschnittstellen ein nicht unerheblicher Kostenfaktor bei der Softwareentwicklung und ein im wissenschaftlichen Bereich viel erforschtes Gebiet. Dabei wird die gesamte Interaktion zwischen Mensch und Maschine von der Wahrnehmung des Menschen über sein Verhalten bis hin zu Eingabe- und Datenverarbeitungsmöglichkeiten der Maschine während der Interaktion betrachtet. Die Schnittstelle zwischen Mensch und Maschine kann nur dann optimal sein, wenn sowohl die menschliche als auch die maschinelle Informationsverarbeitung berücksichtigt wird.

Im Zusammenhang mit Mensch-Maschine Interaktion wird das Verhalten einer Maschine durch das sogenannte Prozessmodell beschrieben. Parallel zu diesem Modell existiert das Aufgabenmodell, das den Plan des Menschen beschreibt, den er zur Erfüllung einer Aufgabe entwickelt. Bei der Mensch-Maschine Interaktion muss der Mensch sein Aufgabenmodell auf sein mentales Prozessmodell abbilden, um die Maschine bedienen zu können. Hierzu ist es notwendig, die einzelnen Teilaufgaben aus dem Aufgabenmodell auf mögliche Teilprozesse des mentalen Prozessmodells des Benutzers abzubilden. Nur so ist es möglich, aus den Teilaufgaben hinreichend genaue Anweisungen für die Maschine zu formulieren. Problematisch bei dieser Abbildung ist eine möglicherweise falsche Vorstellung des Benutzers vom Prozessmodell oder eine zu große Differenz zwischen Aufgabenmodell und mentalem Prozessmodell, die eine verlustfreie Abbildung unmöglich macht. Das Problem für den Benutzer liegt demnach darin, dass seine Vorstellung über das Prozessmodell der Maschine konkret genug sein muss, um ihr Verhalten zu verstehen, aber gleichzeitig immer ungenau sein wird, da er in der Regel wenig oder gar kein Wissen über den internen Aufbau der Maschine besitzt. Eine gute Benutzerschnittstelle kann helfen, dem Benutzer das notwendige Verständnis der Maschine in der Form zu vermitteln, verschiedene zielführende und ausführbare Teilprozesse des Prozessmodells anzubieten und darzustellen. Durch diese Kapselung werden Fehler z.B. auftretend durch die Reihenfolge der Eingaben vermieden. Auch die Repräsentation der Teilprozesse der Schnittstelle ist dabei von großer Relevanz. Die Verwendung von Tooltipps beispielsweise ist eine Möglichkeit dem Benutzer Informationen anzubieten ohne die Übersichtlichkeit der Benutzerschnittstelle zu verringern. Ein weiteres Problem tritt auf, da es keine Schnittstelle gibt, die für jeden Benutzer optimal ist. Dies liegt daran, dass zwei unterschiedliche Benutzer nicht zwangsläufig dasselbe Aufgabenmodell oder dieselbe Vorstellung des Prozessmodells haben. Vor allem bei umfangreichen Maschinen, wie beispielsweise heutige Textverarbeitungsprogramme kann die Motivation der Verwendung von Nutzer zu Nutzer variieren. Je nachdem, ob das Textverarbeitungsprogramm zum Erstellen von Berichten, Serienbriefen oder wissenschaftlichen Arbeiten genutzt wird, werden unterschiedliche Funktionen benötigt. Eine Verbesserung der Interaktion kann daher erreicht werden, wenn dem Benutzer eine Möglichkeit der Individualisierung der Schnittstelle angeboten wird. Die Verwendung dieser adaptiven Benutzerschnittstellen ist zwar ein Mehraufwand bei der Implementierung, verspricht jedoch eine Verbesserung der Mensch-Maschine Interaktion durch die angesprochene Individualisierbarkeit.

1.1 Motivation

Eine Verbesserung der Mensch-Maschine Interaktion kann durch Adaption der Benutzerschnittstelle an den menschlichen Benutzer erreicht werden. Motiviert ist diese Arbeit daher aus der Notwendigkeit eine Plattform zu entwickeln, welche eine Adaption einer Schnittstelle durch formale, von der Maschine zu verstehende Rekonfiguration ermöglicht. Eine formale Beschreibung ermöglicht dabei eine automatisierte Veränderung der Benutzerschnittstelle und erleichtert die Analyse und Validierung von Benutzerschnittstellen.

Die formale Beschreibung von Schnittstellen kann unterteilt werden in die formale Beschreibung der physikalischen Repräsentation und des logischen Verhaltens der Schnittstelle, im Folgenden als Interaktionslogik bezeichnet [19]. Die Interaktionslogik stellt dabei einen getrennten Formalismus dar, der die Kommunikation zwischen der physischen Repräsentation der Schnittstelle und der eigentlichen Funktionalität der Maschine (Aktionslogik) beschreibt. Auf dieser Basis sind flexible Veränderungen der physischen Repräsentation möglich, da durch Anpassungen der Interaktionslogik die Verbindung zur Aktionslogik eindeutig bleibt. Die Veränderung der physischen Repräsentation wird als Redesign bezeichnet und umfasst sowohl Veränderungen von Farbe, Position oder Ähnlichem von Elementen der Schnittstelle als auch das Hinzufügen oder Entfernen von Elementen. Dabei können Redesignoperationen auch Veränderungen der Interaktionslogik nach sich ziehen. Dies ist beispielsweise meist beim Einfügen neuer Elemente in die physische Repräsentation oder bei der Fusion von vorhandenen Elementen der Fall, da die hinter den veränderten Elementen befindliche Logik entsprechend der Redesignoperation angepasst werden muss. Jede Veränderung des Nachrichtenaustauschs zwischen physischer Repräsentation und Aktionslogik, welcher durch die Interaktionslogik realisiert ist, wird als eine Rekonfiguration bezeichnet. Redesign und Rekonfiguration werden meist in Kombination genutzt, können jedoch auch getrennt voneinander stattfinden. Ein einfaches Verschieben oder Umfärben von Elementen der Schnittstelle stellt beispielsweise ein Redesign dar, welches jedoch keine Rekonfiguration nach sich zieht. Gleichzeitig kann Redesign sowohl Rekonfigurationen nach sich ziehen, als auch eine Folge von Rekonfiguration sein.

Ein weiterer Aspekt von Redesign und Rekonfiguration ist die Frage, wer diese Veränderungen auslösen soll. Außerdem ist die Frage zu klären, wer entscheidet, wie ein mögliches Redesign bzw. eine mögliche Rekonfiguration konkret aussieht. Geschehen kann dies sowohl direkt durch den Benutzer als auch durch das System oder einen Administrator. Es sind ebenfalls Mittelwege denkbar, wie Systeme, die dem Benutzer Vorschläge der Rekonfiguration oder des Redesigns unterbreiten, so dass dieser eine eingeschränkte Auswahl hat.

Das Redesign einer Schnittstelle kann allgemein beschrieben werden, ist jedoch in der Implementierung in der Regel stark von der Bibliothek abhängig, welche zum Design der physischen Repräsentation verwendet wird. Dieser Abhängigkeit kann beispielsweise durch die Verwendung einer XML-basierten Definition der physischen Repräsentation entgegen gewirkt werden. Flexibler sind die Möglichkeiten der Modellierung von Interaktionslogik. Um diese zu realisieren, wird ein Formalismus benötigt, der nebenläufig und unabhängig vom restlichen System ausgeführt werden kann. Eine weitere Anforderung ist die Veränderbarkeit der Interaktionslogik. Es hat sich gezeigt, dass farbige Petri Netze [12] im Stande sind dies zu leisten [19]. Farbige Petri Netze bieten von sich aus Nebenläufigkeit und ermöglichen die Definition der für die Interaktionslogik nötigen Programmlogik. Allein die dynamische Anpassbarkeit der Interaktionslogik muss durch einen weiteren Formalismus abgebildet werden. Bei der Nutzung von Petri Netzen entspricht eine Anpassung der Interaktionslogik einer Veränderung der Struktur des Petri Netzes. Es bieten sich daher Graphtransformationssysteme an, um eine Plattform zur Umstrukturierung formal beschriebener Petri Netze zu implementieren.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, ein Graphtransformationssystem zu entwickeln und zu implementieren, welches eine formale Rekonfiguration auf Basis von farbigen Petri Netzen ermöglicht. Dabei sind verschiedene Aspekte und Anforderungen zu beachten.

Zunächst muss ein geeigneter Ansatz der Graphtransformation gewählt werden (Kapitel 2). Dazu müssen geeignete Ersetzungssysteme für allgemeine Graphen und Petri Netze betrachtet und bezüglich ihrer Verwendbarkeit für die Rekonfiguration von Interaktionslogik bewertet werden. Weiterhin müssen zusätzliche Erweiterungen der Ansätze für farbige Petri Netze erarbeitet werden, da die Farbe von Petri Netzen in der Modellierung von Interaktionslogik eine wichtige Rolle spielt. Nur farbige Petri Netze bieten die nötige Syntax zur Definition von Programmlogik. Es ist dabei ein Formalismus für farbige Petri Netze zu wählen, der gut in die verwendete Programmiersprache integriert werden kann.

Auf Basis der gewählten theoretischen Grundlage muss ein System zu Graphtransformation modelliert und implementiert werden (Kapitel 3). Zentral ist dabei zu klären, wie der allgemeine Aufbau eines solchen System aussieht und wie dieser in der gewählten Programmiersprache optimal implementiert werden kann, in Hinblick auf die theoretischen Grundlagen. Ein weiteres primäres Ziel ist die Definition von Datenstrukturen sowohl für Petri Netze, als auch für Transformationsregeln. In diesem Zusammenhang spielen auch Korrektheitsprüfungen von Netzen oder Transformationsregeln eine Rolle und müssen daher geeignet in das Gesamtsystem integriert werden. Weiterhin ist eine Fehlerbehandlung zu implementieren, die Probleme behandelt, wie beispielsweise fehlerhaft definierte Regeln.

Schlussendlich soll vorgestellt werden, wie eine Implementierung und Einbindung von Interaktionslogik zur inkrementellen Interaktionsmodellierung in eine Benutzerschnittstelle praktisch erfolgen kann (Kapitel 4). Dabei soll die erstellte Implementierung des Systems zur Graphtransformation verwendet werden, sowie zu diesem Zweck geeignete Bibliotheken vorgestellt werden.

Erstmals wird im Rahmen dieser Arbeit eine Implementierung des Graphtransformationsansatzes nach Ehrig et al. in einer objektorientierten Sprache präsentiert, die auf einer Übertragung des allgemeinen und formal spezifizierten Ansatzes auf XML-basierte Petri Netze beruht. Es soll dadurch ein hohes Maß an Austauschbarkeit erreicht werden, das es ermöglicht, die zu erstellende Software in verschiedenen Implementierungen wiederzuverwenden, genauso wie den verwendeten Petri Netz-Formalismus auf eine sehr einfache Weise anzupassen. Dabei versteht sich die zu erstellende Bibliothek nicht als Software zur Transformation von beliebigen Arten von Graphen, sondern allein zur Transformation von Petri Netzen. Schlussendlich ist es Ziel, eine robuste und hoch modularisierte Softwarelösung zur Transformation von Petri Netzen zur Verfügung zu stellen, die ein Höchstmaß an Konformität zum theoretischen Ansatz gewährleistet.

2 Graphtransformation

2.1 Überblick

Die Zielsetzung von Graphtransformationen ist die gezielte Manipulation eines Urgraphens zur Erzeugung eines Bildgraphen. Dabei werden durch formal definierte Regeln Transformationen realisiert. Graphtransformationssysteme lassen sich unterscheiden in Graphersetzungssysteme und Graphgrammatiken. Der Aufbau der Transformationsregeln ist bei Graphersetzungssystemen und Graphgrammatiken gleich, allerdings unterscheiden sie sich in ihrer Zielsetzung.

Bei Graphgrammatiken werden Transformationsregeln zur Erzeugung von Graphen verwendet. Gegeben sind dabei eine Menge von Regeln und ein Startgraph. Durch sukzessives Anwenden der Regeln erhält man aus dem Startgraphen eine Menge von so genannten "abgeleiteten" Graphen.

Bei Graphersetzungssystemen liegt der Schwerpunkt darauf, gegebene Graphen anzupassen. Sei beispielsweise eine Straßenkarte als Graph gegeben. Neu gebaute Straßen können durch Graphersetzungssysteme in den vorhandenen Graphen übernommen werden. Es ist dabei ausreichend nur den zu verändernden Teil des Graphen zu betrachten.

Nach Schürr und Westfechel [17] lassen sich die Graphersetzungssysteme in drei Gruppen einteilen:

Mengentheoretischer Ansatz Der mengentheoretische Ansatz definiert Transformationsregeln auf Basis der Mengentheorie. Dazu werden Operationen wie Vereinigung, Schnitt, etc. verwendet, um die Mengen der Knoten und Kanten zu verändern. Die in diesem Ansatz verwendete Formalisierung ist einfacher, hat allerdings eine nicht so breite theoretische Fundierung wie die der anderen beiden Ansätze.

Kategorientheoretischer Ansatz Bei dem kategorientheoretischen Ansatz wird auf die Kategorientheorie zurückgegriffen. Die Kategorientheorie ist eine Abstraktion der Idee der Funktion. Statt Mengen aufeinander abzubilden, spricht man allgemein von Objekten und von Pfeilen (Morphismen), die Objekte auf andere Objekte abbilden. Zusätzlich spielen in der Kategorientheorie Abbildungen zwischen Kategorien, sogenannte Funktoren, eine Rolle. Im kategorientheoretischen Ansatz wird eine Kategorie für Graphen definiert und Regeln durch kategorientheoretische Konstrukte wie *pushouts* realisiert. Näheres zu den Grundlagen der Kategorientheorie kann in [1] nachgelesen werden.

Im Gegensatz zum mengentheoretischen Ansatz werden Graphen direkt aufeinander abgebildet. Zur Analyse dieses Ansatzes kann dabei auf die allgemeinen Erkenntnisse der Kategorientheorie zurückgegriffen werden.

Logikorientierter Ansatz Beim logikorientierten Ansatz werden sowohl Graphen, als auch Transformationsregeln durch prädikatenlogische Formeln erster Stufe beschrieben. Dies ist der am wenigsten verbreitete Ansatz, da er noch wenig erforscht ist. Die Grundlagen dieses Ansatzes sind in [17] erläutert.

Im Folgenden wird näher auf den kategorientheoretischen Ansatz eingegangen.

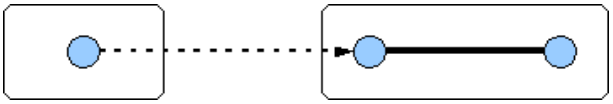


Abbildung 2.1: Beispielregel zur Transformation von Graphen

Beispielhaft sei als Startgraph ein einzelner Knoten sowie die in Abbildung 2.1 angegebene Regel gegeben. Eine Anwendung der Regel bewirkt, dass ein neuer Knoten erzeugt und dieser mit einem bereits vorhandenen Knoten des Graphen durch eine ungerichtete Kante verbunden wird. Die von dieser Graphgrammatik erzeugte Menge von abgeleiteten Graphen beschreibt alle möglichen ungerichteten Bäume (außer dem leeren Graphen). Zum Einen ist jeder so erzeugte Graph ein Baum, denn ein einzelner Knoten ist ein Baum und damit auch jeder davon abgeleitete Graph, da die Anwendung der Regel keine Kreise erzeugt. Zum Anderen kann jeder Baum mit n Knoten in $n-1$ Schritten erzeugt werden. Man wähle dazu einen beliebigen Knoten als Wurzel und erzeuge den Baum ebenenweise. Die Regel wird auf jeden Knoten solange angewandt, bis der Grad des Knoten dem im ursprünglichen Baum entspricht.

Angelegt ist dieses Konzept an formale Grammatiken, wie sie zur Definition der Chomsky-Hierarchie benutzt werden. Diese verwenden Terminale und Nichtterminale in Kombination mit Produktionen, um Wörter zu erzeugen (unter anderem beschrieben in [18]). Die erzeugten Wörter entsprechen hier den Graphen, die Produktionen den Graphtransformationsregeln.

2.2 Regelbasierte Graphtransformation

Um regelbasierte Graphtransformationen formal beschreiben zu können, wird im Folgenden eine Begriffsbildung für die weitere Arbeit angegeben. Die folgenden Definitionen sind angelegt an [5]. Fokus liegt hierbei auf der Definition und Verarbeitung von Produktionen. Weiterführende Eigenschaften der Ansätze werden in [16] erläutert.

Definition 1 (Graph) *Ein (gerichteter) Graph ist ein Tupel $G = (V, E, s, t)$. Dabei ist V die Menge der Knoten (vertices), E die Menge der Kanten (edges) und s, t Abbildungen. Für jede Kante gibt $s : E \to V$ (source) den Startknoten und $t : E \to V$ (target) den Endknoten an.*

Vereinbarung 1 (Produktion und Matching) *Eine Produktion ist eine Abbildung $p : L \to R$, ein Matching eine Abbildung $m : L \to G$, wobei L, R und G Graphen sind. Die zu p und m parallelen Abbildungen werden bezeichnet als $p^* : G \to H$ und $m^* : R \to H$, wobei H ebenfalls ein Graph ist.*

Im Folgenden sei eine Regel die Kombination aus Produktion und Matching. Die Grundidee, die hinter allen Ansätzen der Graphtransformation steht, ist eine Produktion $p : L \to R$ zu definieren, die zwei Graphen L und R verknüpft. In diesem Zusammenhang wird von der linken Produktionsseite L und der rechten Produktionsseite R gesprochen. Zur Anwendung der Produktion wird weiterhin eine Abbildung $m : L \to G$ benötigt, die die linke Seite auf den zu verändernden Graphen G abbildet. Die Anwendung der Produktion ist schließlich darauf reduziert, das Vorkommen von L in G durch R zu ersetzen. Auf Basis dieser Betrachtung ist es möglich, Produktionen zu definieren, die auf beliebige Graphen angewandt werden können, sofern m geeignet gewählt ist. Zu klären ist dabei zum Einen, wie die Abbildung m zu definieren ist, und zum Anderen wie die Ersetzung der linken Seite L durch die rechte Seite R realisiert wird.

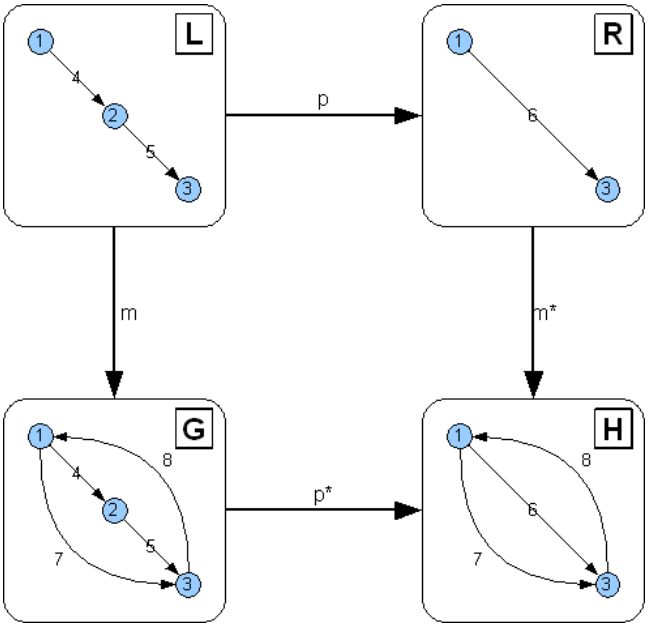


Abbildung 2.2: Beispiel einer Produktion ohne Konflikt

In Abbildung 2.2 ist eine Produktion angegeben, die bei einem gerichteten Weg der Länge 2 den mittleren Knoten löscht und beide Kanten durch eine neue ersetzt. Für die Abbildungen m, p, m^* und p^* identifizieren die Nummern der Kanten und Knoten in Abbildung 2.2 jeweils ein Element und sein Bild bezüglich der jeweiligen Abbildung.

Die Abbildung m definiert, welche Teile von G zum Vorkommen von L in G gehören. Dazu bildet m alle Knoten und Kanten aus L auf G ab. L und das Bild von m müssen dabei nicht isomorph sein, wobei m die Eigenschaften eines (totalen) Homomorphismus erfüllen muss. Ein Homomorphismus ist dabei eine strukturerhaltende Abbildung. Die strukturerhaltende Eigenschaft eines solchen Homomorphismus ist vor allem für die Abbildung der Kanten von Bedeutung, d.h. die Endpunkte des Bildes einer Kante aus L müssen identisch sein mit den Bildern der Endpunkte der Kante. Es muss daher für ein Matching $m : L \to G$ mit den Graphen $L = (V, E, s, t)$ und $G = (V', E', s', t')$ gelten:

$$\forall e \in E : s'(m(e)) = m(s(e)) \wedge t'(m(e) = m(t(e))$$

Auch p muss ein Homomorphismus sein, allerdings nicht notwendigerweise total. Soll die Produktion p angewendet werden, so geht man wie folgt vor:

- Alle Knoten und Kanten aus L, die ein Bild in R besitzen (in Abbildung 2.2 die Knoten 1 und 3), bleiben erhalten.
- Alle Knoten und Kanten aus L, die kein Bild in R haben (hier der Knoten 2 und die Kanten 4 und 5), werden mittels m auf G abgebildet und dort gelöscht.
- Analog werden alle Knoten und Kanten aus R, die kein Urbild in L haben (hier Kante 6), in G eingefügt. Die Endpunkte von so erzeugten Kanten werden mittels m^* identifiziert.

Auftretende Konflikte

Aus dieser Vorgehensweise können Konflikte resultieren, wie in Abbildung 2.3 und 2.4 dargestellt ist.

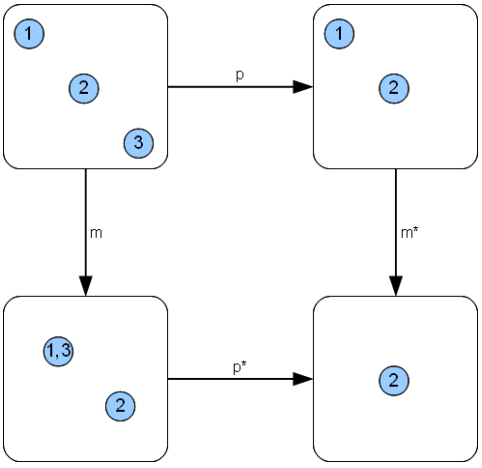


Abbildung 2.3: Produktion mit Löschkonflikt bei Knoten

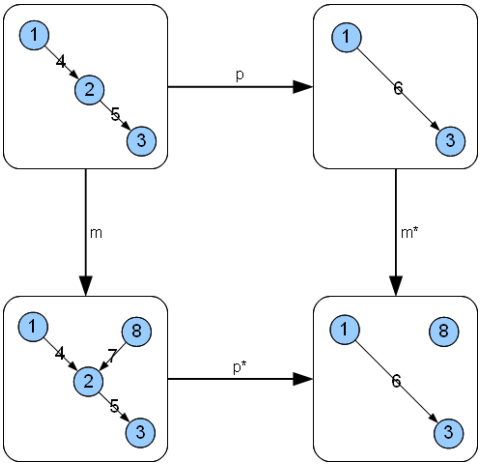


Abbildung 2.4: Produktion mit Löschkonflikt bei Kanten

In Abbildung 2.3 ist m nicht injektiv, da die Knoten 1 und 3 auf 1,3 abgebildet werden. Der Konflikt entsteht durch die Widersprüchlichkeit von p in Kombination mit m, wobei Knoten 3 gelöscht und Knoten 1 erhalten werden müsste. Da diese Knoten das gleiche Bild in G haben, können diese Forderungen offensichtlich nicht gleichzeitig erfüllt werden. Grundsätzlich hat man hier die Möglichkeit den Knoten 1,3 zu erhalten, was dazu führen würde Knoten 3 nicht zu löschen oder den Knoten 1,3 zu löschen, wobei Knoten 1 nicht erhalten bleiben würde. Alternativ kann geprüft werden, ob ein solcher Konflikt auftritt. Fällt eine solche Überprüfung positiv aus, so wird die Anwendung der Produktion verweigert.

In Abbildung 2.4 entsteht der Konflikt durch die Löschung eines Knotens, an dem eine Kante grenzt, welche nicht Teil der Regel ist. In Abbildung 2.4 betrifft dies Kante 7 und Knoten 2. Das Ergebnis einer solchen Produktion wäre kein korrekter Graph mehr, da eine Kante mit einem Knoten verbunden ist, der nicht mehr existiert. Das Problem solcher "baumelnden" Kanten kann durch Löschen dieser gelöst werden. Eine Vorhersage über das Ergebnis der Anwendung einer solchen Produktion wird dadurch allerdings erschwert, da nun unabhängig von der definierten Produktion Kanten gelöscht werden. Auch hier kann alternativ die Anwendung der Produktion verweigert werden.

In den Abbildungen 2.3 und 2.4 wurden im Konfliktfall Knoten und Kanten gelöscht und nicht erhalten. Die beiden algebraischen Ansätze *single pushout approach* (SPO siehe 2.2.2) und *double pushout approach* (DPO siehe 2.2.3), die im Folgenden erläutern werden, unterscheiden sich neben ihrer Definition einer Produktion vor allem in der Art und Weise, wie sie die erwähnten Konflikte lösen. Beide Ansätze verkleben dabei die in der Regel angegebenen Graphen miteinander unter Verwendung von *pushouts* basierend auf dem kategorientheoretischen Ansatz. Bei der Kategorie, die dafür verwendet wird, sind die Objekte Graphen, die Pfeile Homomorphismen.

2.2.1 Bestimmung eines pushout

Ein *pushout* wird konstruiert, wenn zwei kategorientheoretische Pfeile (f, g) "gleichzeitig" ausgeführt werden sollen. Man muss dazu zwei Pfeile f^* und g^* finden, so dass aus $A \xrightarrow{f} B \xrightarrow{g^*} D$ und $A \xrightarrow{g} C \xrightarrow{f^*} D$ das gleiche Objekt D resultiert, wie es in Abbildung 2.5 dargestellt ist. Dazu seien zunächst folgende Definitionen gegeben:

Definition 2 (pushout) *Für zwei Pfeile $f : A \to B$ und $g : A \to C$ bezeichnet man das Tripel $(D, g^* : B \to D, f^* : C \to D)$ als pushout und D als pushout Objekt von (f, g), wenn es folgende Bedingungen erfüllt:*

1. $g^* \circ f = f^* \circ g$
2. *Für alle anderen Objekte E mit den Pfeilen $f' : C \to E$ und $g' : B \to E$, die die erste Bedingung erfüllen, gibt es einen Pfeil $h : D \to E$ mit $h \circ g^* = g'$ und $h \circ f^* = f'$.*

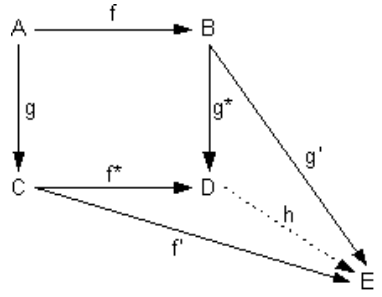


Abbildung 2.5: Ein vollständiges *pushout* Diagramm

Definition 3 (pushout Komplement) *Für zwei Pfeile $f : A \to B$ und $g^* : B \to D$ bezeichnet man das Tripel $(C, g : A \to C, f^* : C \to D)$ als pushout Komplement von (f, g^*), wenn (D, g^*, f^*) ein pushout von (f, g) ist.*

Um die erste Bedingung zu erfüllen, muss in die Konstruktion von g^* (bzw. f^*) g (bzw. f) einfließen. Nur dadurch wird sichergestellt, dass die Abbildung von A auf das *pushout* Objekt unabhängig vom genommenen Weg (über f oder über g) ist und damit sowohl f als auch g umfasst.

Die zweite Bedingung stellt sicher, dass der *pushout* - bis auf Isomorphie - eindeutig ist, wenn er existiert. Ohne diese Einschränkung kann es beliebig viele *pushouts* (D, g^*, f^*) für (f, g) geben, die zwar korrekt wären, aber unterschiedlich aussagekräftig sind. Sei erneut das Anfangsbeispiel aus Abbildung 2.2 betrachtet. Es könnte m^* und p^* so gewählt werden, dass jeder Knoten auf ein und denselben Knoten in H und jede Kante auf einen Loop an diesem Knoten abgebildet wird. Das Ergebnis wäre, dass H nur einen einzelnen Knoten mit einem Loop enthielte. Dieser Graph hat jedoch wenig Aussagekraft, da er das *pushout* Objekt für jedes Paar von Abbildungen (f, g) wäre, die jeweils mindestens auf einen Knoten und eine Kante abbilden. Deshalb wird nur nach dem *pushout* Objekt gesucht, das sich auf alle möglichen Alternativen abbilden lässt und dadurch das aussagekräftigste Ergebnis erreicht.

Für beliebige Kategorien kann es vorkommen, dass zwei Pfeile (f, g) kein *pushout* besitzen. Die Kategorie, die für Graphtransformationen verwenden wird (auch Kategorie **Graph** genannt), eignet sich jedoch für die Formalisierung von Regeln, da die Konstruktion eines *pushout* immer möglich ist. Das liegt daran, dass die Konstruktion des *pushouts* für die Knoten- und Kantenmenge eines Graphen getrennt berechnet werden kann und somit das Problem auf die Kategorie der Mengen (auch Kategorie **Set** genannt) reduziert wird. Auch für diese Kategorie lässt sich immer ein *pushout* bestimmen. Dies geschieht für zwei Abbildungen $f : A \to B$ und $g : A \to C$ durch die disjunkte Vereinigung von B und C. Um die *pushout* Kriterien zu erfüllen, muss jedoch jedes Paar aus Elementen von B und C, mit demselben Urbild in A, dasselbe Bild in D haben,

wobei D das *pushout* Objekt bezeichnet (siehe auch Abbildung 2.5). Dies geschieht unter Einbeziehung der Äquivalenzrelation $\equiv$, indem die Äquivalenzklassen der disjunkten Vereinigung von B und C bezüglich $\equiv$ gebildet werden. $\equiv$ ist dabei für zwei Elemente $b \in B$ und $c \in C$ definiert durch: $b \equiv c$, wenn $\exists a \in A : f(a) = b \wedge g(a) = c$. Dieses Vorgehen wird auch als Verkleben von B und C entlang A bezeichnet und geschrieben als $D = B +_A C$.

Für weitergehende Betrachtungen sei an dieser Stelle auf [1] verwiesen.

2.2.2 Single pushout approach (SPO)

Definition 4 (SPO Regel) *Seien L, R und G Graphen. Eine SPO Regel ist dann ein Paar (p, m), wobei $p : L \to R$ eine SPO Produktion und $m : L \to G$ ein Matching ist. p ist dabei ein partieller und m ein totaler Graph-Homomorphismus.*

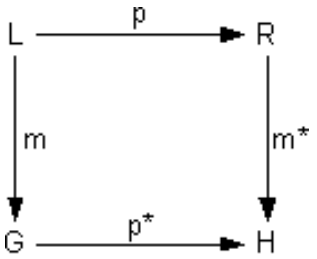


Abbildung 2.6: *pushout* Diagramm des SPO Ansatzes (entnommen aus [5])

Der SPO Ansatz funktioniert wie im Beispiel am Anfang dieses Kapitels erläutert. Die Bezeichnungen seien dabei wie in Definition 4 und Abbildung 2.6 beschrieben. Eine Regel muss sonst keine zusätzlichen Eigenschaften erfüllen, jede Regel ist daher per se anwendbar.

Tritt ein Konflikt wie auf Seite 12 beschrieben auf, so wird bei diesem Ansatz das Löschen von Elementen dem Erhalten von Elementen vorgezogen. Müsste ein Teil des Graphen laut Regel gelöscht werden und erhalten bleiben, so wird er gelöscht. Ist in der Produktion ein Knoten zum Löschen markiert, so werden implizit auch alle Kanten gelöscht, die an diesem Knoten anliegen, selbst wenn sie nicht in der Produktion aufgeführt sind. Das kann Vorteile bieten, aber auch zu ungewollten Nebeneffekten führen. Bildet m zwei Knoten von L auf denselben Knoten in G ab und genau einer der zwei Knoten soll gelöscht werden, so wird bei diesem Ansatz der Knoten aus G gelöscht. m^* wird dann zu einem partiellen Homomorphismus, da der Knoten, der erhalten bleiben sollte, aber gelöscht wurde, nun kein Bild mehr in H besitzt.

Eine Anwendung einer Regel des SPO Ansatzes entspricht der Konstruktion eines *pushouts*.

2.2.3 Double pushout approach (DPO)

Definition 5 (DPO Regel) *Seien I, L, R, G, C und H Graphen. Eine DPO Regel ist dann ein Paar $((l, r), m)$ aus einer DPO Produktion (l, r) und einem Matching $m : L \to G$. Die DPO Produktion besteht aus zwei totalen Graph Homomorphismen $l : I \to L$ und $r : I \to R$, wobei I der sogenannte Interface Graph ist. l wird als linke Regelseite und r als rechte Regelseite bezeichnet. Das Matching m ist ein totaler Graph Homomorphismus.*

Da der SPO Ansatz in dieser Arbeit nur eine nachgeordnete Rolle spielt, werden DPO Produktionen im Folgenden nur als Produktion bezeichnet. Wie bereits durch die Bezeichnung suggeriert und in Abbildung 2.7 dargestellt, benutzt der DPO Ansatz zwei *pushout* Diagramme um eine Regel zu realisieren. Ziel dieses Ansatzes ist es, mögliche Konflikte zu erkennen und ggf.

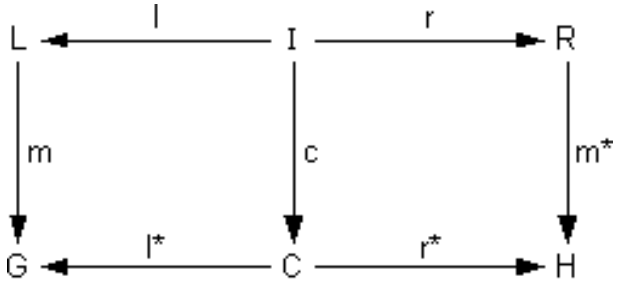


Abbildung 2.7: *pushout* Diagramm des DPO Ansatzes (entnommen aus [5])

die Anwendung der Regel zu verweigern. Die Sicherheit im Sinne des Verhinderns ungewollter Nebeneffekte wird dadurch erhöht. Der Sicherheitsgewinn ist in diesem Zusammenhang gleich bedeutend mit einer höheren Kontrolle über das Ergebnis einer Regelanwendung.

Anders als beim SPO Ansatz werden beim DPO Ansatz die Löschung und das Einfügen von Knoten bzw. Kanten voneinander getrennt. Der Interface Graph I definiert alle Elemente, die erhalten bleiben sollen. Aus G werden die Bilder aller Elemente aus L gelöscht, die kein Urbild in I besitzen. Analog werden in G alle Knoten und Kanten aus R eingefügt, die kein Urbild in I besitzen.

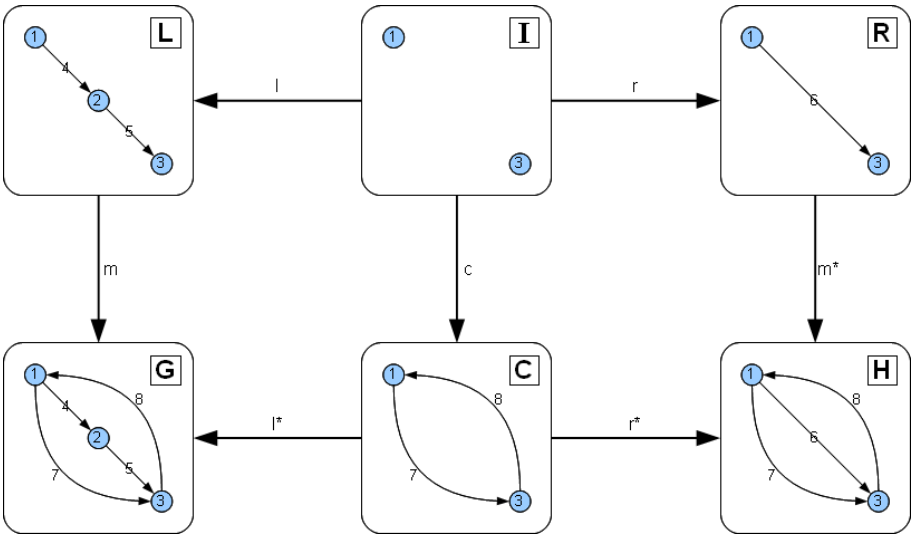


Abbildung 2.8: Die Regel aus Abbildung 2.2 geschrieben als DPO Regel

Abbildung 2.8 zeigt, wie eine Regel des DPO Ansatzes aussehen kann. Im ersten Schritt werden die Bilder der beiden Kanten und des Knoten 2 in L aus G gelöscht und es resultiert daraus der Kontext Graph C. In einem zweiten Schritt wird die Kante 6 in C eingefügt, da sie kein Urbild in I besitzt. Sowohl das Einfügen als auch das Löschen wird mittels *pushouts* realisiert. Dabei ist zu beachten, dass beim Einfügen der *pushout* von (r, c) berechnet werden muss, um H zu erhalten. Beim Löschen von Elementen muss jedoch das *pushout* Komplement von (l, m) bestimmt werden, um C zu erhalten. Wie später gezeigt wird, existiert das *pushout* Komplement nicht in jedem Fall. Falls ein *pushout* Komplement existiert, ist es nicht zwangsläufig eindeutig.

Zusätzlich zur Trennung der Regelseiten im Aufbau der Produktion, wird eine Bedingung eingeführt, die erkenntlich macht, ob es bei der Anwendung der Regel zu einem Konflikt kommen wird, wie in Abschnitt 2.2 beschrieben ist.

Definition 6 (gluing condition) *Seien* $I = (V_I, E_I, s_I, t_I)$, $L = (V_L, E_L, s_L, t_L)$ *und* $G = (V_G, E_G, s_G, t_G)$ *Graphen. Zwei Graph-Homomorphismen* $l : I \to L$ *und* $m : L \to G$ *erfüllen die gluing condition genau dann, wenn folgende Bedingungen erfüllt sind:*

$$\neg\exists e \in (E_G \backslash m(E_L)) \quad : \quad s_G(k) \in m(V_L \backslash l(V_I)) \vee t_G(k) \in m(V_L \backslash l(V_I)) \tag{2.1}$$

$$\neg\exists x, y \in (V_L \cup E_L) \quad : \quad x \neq y \wedge m(x) = m(y) \wedge x \notin l(V_I \cup E_I) \tag{2.2}$$

Die *gluing condition* besteht aus der *dangling condition* (Bed. 2.1 aus Definition 6) und der *identification condition* (Bed. 2.2 aus Definition 6).

Die *dangling condition* besagt, dass für jeden Knoten, der gelöscht werden soll, auch explizit alle Kanten, die mit ihm verbunden sind, gelöscht werden müssen. Durch diese Bedingung ist eindeutig festgelegt, welche Kanten gelöscht werden. Die *indentification condition* besagt, dass Knoten und Kanten aus G nicht gelöscht werden dürfen, wenn sie mehr als ein Urbild in L haben. Hierdurch wird verhindert, dass ein Element aus G laut Produktion gleichzeitig gelöscht werden und erhalten bleiben muss.

Regeln, die die *gluing condition* nicht erfüllen, werden nicht ausgeführt. Erfüllt eine Regel die *gluing condition*, so ist sicher gestellt, dass zum Einen genau die Elemente gelöscht werden, die die Produktion explizit zur Löschung vorsieht und zum Anderen genau die Elemente eingefügt werden, welche die Produktion zur Einfügung vorsieht. Hieraus resultiert eine höhere Sicherheit bei der Anwendung der Regel auf Kosten der Anwendbarkeit, da nicht alle Regeln automatisch korrekt und damit anwendbar sind. Eine weitere Eigenschaft des DPO Ansatzes ist, dass man zu gegebenen Produktionen leicht eine inverse Produktion erzeugen kann, wenn die verwendeten Abbildungen injektiv sind. In diesem Fall reicht es, l und r zu vertauschen, um die Produktion zu invertieren.

Probleme bei der Bestimmung des pushout Komplements

Wie bereits in Abschnitt 2.2.1 erläutert, existiert für jedes Paar von Graph-Homomorphismen ein eindeutiger *pushout* (siehe Beweis in [1] und [5]). Die rechte Seite einer DPO Produktion ist daher klar definiert. Zur Berechnung der linken Seite muss das *pushout* Komplement für (l, m) bestimmt werden. Dieses existiert genau dann, wenn die *gluing condition* erfüllt ist. Alle korrekten Regeln sind daher anwendbar. Sind l und m injektiv, so ist das *pushout* Komplement eindeutig. Ist dies nicht der Fall, gibt es mehrere korrekte *pushout* Komplemente, wobei jedes ein unterschiedliches DPO Diagramm bildet.

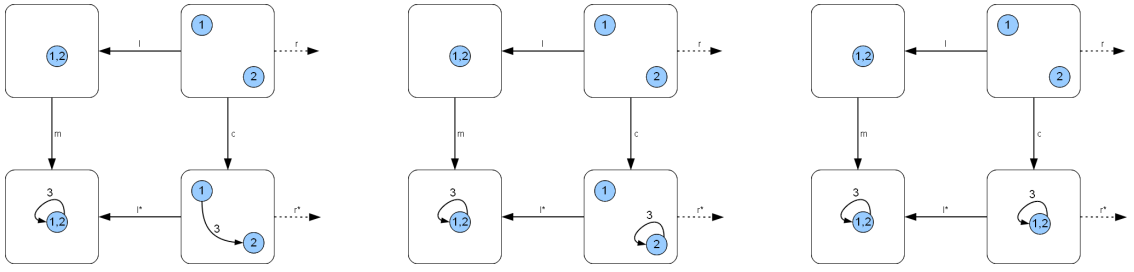


Abbildung 2.9: Drei Möglichkeiten, ein *pushout* Komplement zu bilden

Sei dies verdeutlicht an der Abbildung 2.9, in der l als nicht injektiv angenommen sei. Sei außerdem r die Identität, woraufhin r^* ebenfalls die Identität ist und m^* mit c übereinstimmt. Das *pushout* Komplement Objekt C ist daher automatisch bereits der Ergebnis Graph H, da $r^* : C \to H$ den Graphen nicht verändert. Gegeben sind l und m, gesucht ist ein Pushout Komplement $(C, c : I \to C, l^* : C \to G)$, so dass (G, m, l^*) ein pushout von (l, c) ist.

Seien nun die drei gegebenen Alternativen in Abbildung 2.9 betrachtet. Alle drei Beispiele erfüllen die geforderten Bedingungen, wobei alle drei unterschiedliche Ergebnisse liefern. l sorgt dafür, dass die Knoten 1 und 2 in jedem Fall fusioniert werden müssen. Unabhängig davon welchen Start- und Zielknoten die Kante 3 in C hat (sie könnte auch von 2 nach 1 oder von 1 nach 1 gehen), durch die Fusion der Knoten 1 und 2 entsteht grundsätzlich G. Im dritten Fall ist c die Konkatenation von l und m und l^* ist die Identität. Die Konkatenation kann immer gebildet werden, da das Bild von l komplett enthalten ist im Definitionsbereich von m.

Wegen der fehlenden Eindeutigkeit werden oft nicht-injektive Abbildungen in der Konstruktion von Regeln ausgeschlossen. Wird Injektivität nicht gefordert, so muss ein Formalismus bereitgestellt werden, der entscheidet, ob und wie betroffene Knoten aufgespaltet werden. Der einfachste Weg ist Knoten nicht zu spalten, da bei der Aufspaltung von Knoten die Anzahl der möglichen Partitionen sehr stark ansteigt, wenn die Zahl der Knoten und Kanten steigt.

2.2.4 SPO und DPO im Vergleich

Der Hauptunterschied zwischen SPO und DPO ist ihr Umgang mit Konflikten. Beim SPO Ansatz ist jede Regel korrekt, da keine *gluing condition* existiert. Dem entsprechend hat der SPO Ansatz eine höhere Aussagekraft. Möchte man beispielsweise einen Knoten mit all seinen Kanten löschen, so reicht eine Regel, die einen Knoten auf den eigentlichen Graphen, aber nicht auf die rechte Regelseite abbildet. Um dasselbe im DPO Ansatz zu realisieren, müsste man den zu löschenden Knoten mitsamt aller seiner Kanten und an diesen Kanten anliegenden Knoten in den linken Graphen L aufnehmen. Der Interface Graph und die rechte Seite bestehen dann nur noch aus allen Knoten, die mit dem zu löschenden Knoten verbunden waren. In diesem Fall ist es ein Nachteil, dass beim DPO Ansatz jede Lösch-Aktion explizit definiert sein muss. Demgegenüber bietet der DPO Ansatz eine höhere Kontrolle darüber, auf welche Graphen eine Produktion angewandt werden darf. Diese Sicherheit kann der SPO Ansatz nicht bieten. Mit dem DPO Ansatz ist i.d.R. leichter umzugehen, da seine Sicherungsmaßnahmen die nicht vorgesehenen Nebeneffekte des SPO Ansatzes verhindern. Werden injektive Abbildungen verwendet, so bietet der DPO Ansatz außerdem eine einfache Möglichkeit, der Invertierung von Produktionen.

Ein detaillierter Vergleich beider Ansätze, mit Fokus auf ihre Eigenschaften, findet sich in [7].

2.3 Anwendung des DPO Ansatzes auf P/T Netzen

Dieser Abschnitt umfasst die Anwendung der Konzepte aus 2.2, speziell den DPO Ansatz aus 2.2.3, auf Petri Netze. Diese Anwendung basiert auf dem Ansatz von Ehrig aus [8] und [9]. An dieser Stelle sei aus Gründen der Begriffsbildung auf eine Definition von P/T Netzen zurückgegriffen, die auch Ehrig verwendet. Als Einführung in Petri Netze sei auf [2] verwiesen. Dort wird auch auf das Markenspiel eines P/T Netzes eingegangen, das an dieser Stelle nicht von vorrangigem Interesse ist.

Definition 7 (Stellen/Transition Netz (P/T Netz)) *Ein P/T Netz ist ein Graph definiert als das Tupel $(P, T, pre, post)$. Jedes $p \in P$ wird dabei als Stelle und jedes $t \in T$ als Transition bezeichnet. Die Abbildungen $pre : T \to P^{\oplus}$ und $post : T \to P^{\oplus}$ definieren die Kanten des Netzes. $P^{\oplus}$ ist die Menge aller endlichen Multimengen, gebildet aus Elementen von P.*

2.3.1 Bestimmung des pushouts

Definition 8 (Abbildungen auf P/T Netzen) *Ein Homomorphismus $f : N_1 \to N_2$ auf zwei P/T Netzen $N_1 = (P_1, T_1, pre_1, post_1)$ und $N_2 = (P_2, T_2, pre_2, post_2)$ ist das Tupel (f_P, f_T),*

mit $f_P : P_1 \to P_2$ und $f_T : T_1 \to T_2$. Für alle $t \in T_1$ müssen f_P und f_T folgende Bedingungen erfüllen:

$$\begin{aligned} pre_2(f_T(t)) &= f_{P^\oplus}(pre_1(t)) \\ post_2(f_T(t)) &= f_{P^\oplus}(post_1(t)) \end{aligned}$$

$f_{P^\oplus} : P_1^\oplus \to P_2^\oplus$ ist dabei definiert als die Multimenge aller Bilder von Elementen der angegebenen Multimenge bezüglich f_P.

Die Abbildung $f_{P^\oplus}$ in Definition 8 dient dazu den *pre*- bzw. *post*-Bereich einer Transition eines Netzes auf ein anderes Netz abzubilden. Dazu wird das Bild aller Stellen der abzubildenden Multimenge bezüglich f_P bestimmt und die Multimenge dieser Bilder gebildet. Ist f_P nicht injektiv, so kann das Bild einer gegebenen Multimenge bezüglich $f_{P^\oplus}$ zwar weniger verschiedene Elemente als die gegebene Multimenge enthalten, die Anzahl der Elemente der beiden Multimengen ist jedoch immer gleich.

Ein Homomorphismus f ist auf zwei Netzen N_1 und N_2 zerlegbar in (f_P, f_T). Dies ermöglicht es, die Menge der Stellen und die Menge der Transitionen jeweils isoliert zu transformieren. Die in Definition 8 gestellten Bedingungen sorgen hierbei für die strukturerhaltende Eigenschaft des Morphismus. Mit solchen Morphismen ist es nicht möglich, den *pre*- oder *post*-Bereich einer Transition zu verändern. Eine Fusion von Stellen ist möglich, die Anzahl der eingehenden und ausgehenden Kanten einer Transition ist allerdings unveränderlich. Diese restriktive Definition schränkt die Menge der möglichen Produktionen ein, da eine Veränderung des *pre*- oder *post*-Bereichs einer Transition durch Löschen und Neuerstellen dieser Transition erfolgen muss.

Durch die Aufteilung der Abbildung $f = (f_P, f_T)$ lässt sich die Erzeugung des *pushouts* von P/T Netzen zurückführen auf die Konstruktion von *pushouts* auf den Mengen P_i und T_i $(i \in \{1, 2\})$.

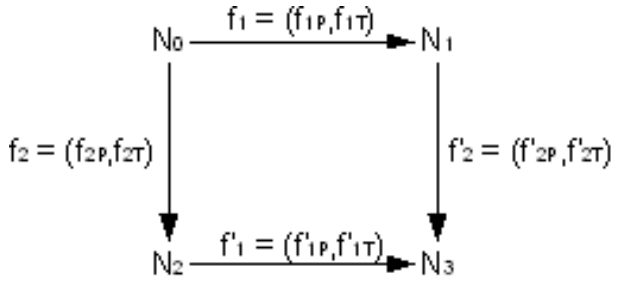


Abbildung 2.10: *pushout* Diagramm für ein P/T Netz (entnommen aus [9])

Seien $f_1 : N_0 \to N_1$ und $f_2 : N_0 \to N_2$ mit $N_i = (P_i, T_i, pre_i, post_i)$, $i \in \{0, 1, 2, 3\}$ die Morphismen, für die ein Netz (das *pushout* Objekt) N_3 bestimmt werden soll. Nach Definition 8 werden die Morphismen zerlegt in $f_1 = (f_{1P}, f_{1T})$ und $f_2 = (f_{2P}, f_{2T})$ (dargestellt in Abbildung 2.10).

Im Folgenden sei zuerst die Erzeugung von P_3 betrachtet. Dazu reicht es aus, den *pushout* von f_{1P} und f_{2P} in der Kategorie der Mengen zu bestimmen. Wie bereits in Abschnitt 2.2.1 erläutert, geschieht dies durch $P_3 = P_1 +_{P_0} P_2$, dem Verkleben von P_1 und P_2 entlang P_0. Analog zur Menge der Stellen wird auch der *pushout* für die Menge der Transitionen auf diese Weise bestimmt durch $T_3 = T_1 +_{T_0} T_2$.

Zusätzlich müssen auch die Abbildungen pre_3 und $post_3$ erzeugt werden, indem auf pre_1 und pre_2 zurückgegriffen wird. Es ergibt sich:

$$pre_3(t) = \begin{cases} f'_{2P^\oplus}(pre_1(t_1)) & ; \quad falls\ f'_{2T}(t_1) = t \\ f'_{1P^\oplus}(pre_2(t_2)) & ; \quad falls\ f'_{1T}(t_2) = t \end{cases}$$

Hier sei angemerkt, dass es durch die Bedingungen in Definition 8 unwichtig ist, welches t_1 bzw. t_2 gewählt wird, da aus allen die gleiche Abbildung pre_3 resultiert. Analog ist auch $post_3$ definiert als:

$$post_3(t) = \begin{cases} f'_{2P^\oplus}(post_1(t_1)) & ; \quad falls\ f'_{2T}(t_1) = t \\ f'_{1P^\oplus}(post_2(t_2)) & ; \quad falls\ f'_{1T}(t_2) = t \end{cases}$$

Der Unterschied zwischen diesem Ansatz und dem in Abschnitt 2.2 behandelten allgemeinen Ansatz besteht vor allem in der Verarbeitung der Kanten. Im Gegensatz zu der allgemeinen Definition von Graphen (Definition 1), sind die Kanten in der hier gewählten P/T Netz Definition nur implizit über *pre* und *post* festgelegt. Eine Veränderung dieser Abbildungen findet auch bei der Konstruktion des pushouts nicht direkt, sondern nur über eine Veränderung der Transitionen und Stellen statt. Löschen (und auch Einfügen) einer Kante kann nicht geschehen ohne die zugehörige Transition mit einzubeziehen.

2.3.2 Bestimmung des pushout Komplement

Im Folgenden wird die in Kapitel 2.2.3 beschriebene *gluing condition* für P/T Netze formuliert. Seien die benutzten Morphismen wie in Abbildung 2.7 gegeben mit $L = (P_L, T_L, pre_L, post_L)$, $I = (P_I, T_I, pre_I, post_I)$ und $G = (P_G, T_G, pre_G, post_G)$. Die rechte Produktionsseite R ist für die Bestimmung der *gluing condition* nicht relevant, da sie keinen Einfluss auf die Löschung von Elementen des Netzes hat. Sei desweiteren eine Menge GP definiert, welche alle Stellen und Transitionen in L enthält, die nicht gelöscht werden. Um die *identification condition* zu erfüllen, dürfen keine Stellen oder Transitionen in L gelöscht werden, die zusammen mit einer anderen Stelle oder Transition auf dasselbe Objekt abgebildet werden. Sei IP die Menge der Stellen und Transitionen, die mehrfach abgebildet werden. Die *identification condition* ist genau dann erfüllt, wenn $IP \subseteq GP$ gilt. Um die *dangling condition* zu erfüllen, darf keine Stelle gelöscht werden, die in G mit einer Transition verbunden ist, auf die von L nicht abgebildet wird. Sei daher DP die Menge dieser Stellen. Die *dangling condition* ist genau dann erfüllt, wenn $DP \subseteq GP$ gilt. Hier sei angemerkt, dass dies nicht für Transitionen gefordert werden muss, da durch die Definition via *pre* und *post* das Löschen einer Transition automatisch das Wegfallen aller Kanten dieser Transition zur Folge hat. Sei $l = (l_P, l_T)$ die linke Regelseite, $r = (r_P, r_T)$ die rechte Regelseite, sowie $m = (m_P, m_T)$ das Matching, so sind GP, IP und DP definiert als:

$$\begin{aligned} GP &= l(P_I \cup T_I) \\ IP &= \{p \in P_L | \exists p' \in P_L : p \neq p' \wedge m_P(p) = m_P(p')\} \\ &\quad \cup \{t \in T_L | \exists t' \in T_L : t \neq t' \wedge m_T(t) = m_T(t')\} \\ DP &= \{p \in P_L | \exists t \in (T_G \backslash m_T(T_L)) : m_P(p) \in pre_G(t) \vee m_P(p) \in post_G(t)\} \end{aligned}$$

Gilt die *gluing condition*, so ist das *pushout* Komplement Objekt C gegeben durch:

$$\begin{aligned} P_C &= (P_G \backslash m_P(P_L)) \cup m_P(l_P(P_I)) \\ T_C &= (T_G \backslash m_T(T_L)) \cup m_T(l_T(T_I)) \end{aligned}$$

Die Abbildung pre_C und $post_C$ ergeben sich durch Reduktion des Definitionsbereichs von pre_G und $post_G$ auf T_C. Sei $l^* : C \to G$ der parallele Morphismus zu l, so ist $pre_C(t_C) = pre_G(l^*_T(t_C))$ und $post_C(t_C) = post_G(l^*_T(t_C))$.

2.3.3 Probleme in der Anwendung

Der dargestellte Ansatz ist korrekt und anwendbar, schränkt jedoch die Möglichkeiten der Definition von Produktionen stark ein. Die in Abschnitt 2.3.1 beschrieben Einschränkung der Morphismen erschwert die Veränderung von vorhandenen Stellen und Transitionen. Soll eine Transition eine zusätzlich eingehende oder ausgehende Kante erhalten, so geschieht dies, indem die Transition vollständig gelöscht und eine neue Transition hinzugefügt wird, die alle gewünschten Kanten besitzt.

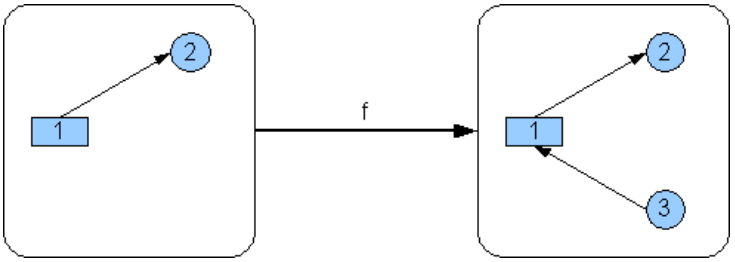


Abbildung 2.11: Die Abbildung f ist laut Definition 8 nicht erlaubt

Das Problem sei verdeutlicht anhand der Abbildung $f : N_1 \rightarrow N_2$ in Abbildung 2.11. Im linken Netz ist $post_1(1) = \{2\}$ und $pre_1(1) = \varnothing$ und im rechten Netz ist $post_2(1) = \{2\}$ und $pre_2(1) = \{3\}$. Laut Definition muss $pre_2(f_T(t)) = f_{P^\oplus}(pre_1(t))$ gelten, allerdings ergibt sich:

$$
\begin{aligned}
& pre_2(f_T(1)) = f_{P^\oplus}(pre_1(1)) \\
\Leftrightarrow \quad & pre_2(1) = f_{P^\oplus}(\varnothing) \\
\Leftrightarrow \quad & \{3\} = \varnothing \qquad \Rightarrow \text{Widerspruch!}
\end{aligned}
$$

f ist daher als rechte Seite (und auch linke) der Produktion nicht erlaubt. Diese Einschränkung dient dazu, die Wirkungsweise von Transitionen unveränderbar zu machen. Im oben genannten Beispiel kann die Transition im ursprünglichen Netz immer feuern, nach der Transformation allerdings ausschließlich, wenn Stelle 3 ein Token zu Verfügung stellt. Die Wirkungsweise ist damit verändert worden. Soll eine Transition verändert werden, so muss diese mitsamt *pre-* und *post-*Bereich in die Produktion aufgenommen werden. Wie bereits beim DPO Ansatz für allgemeine Graphen wird hier die Sicherheit bei der Anwendung der Regel auf Kosten der Anwendbarkeit erhöht. Das Hinzufügen einer einzelnen Kante zu einer Transition kann bereits eine umfangreiche Produktion erfordern, wenn die Transition umfangreich ist.

Durch die implizite Behandlung von Kanten wird zusätzlich die Übertragbarkeit dieses Ansatzes auf speziellere Arten von Petri Netzen erschwert, bei denen Kanten semantische Informationen tragen, die für den Transformationsprozess relevant sind. Eine mögliche Lösung kann durch die explizite Behandlung von Kanten realisiert werden.

2.4 Erweiterung des DPO Ansatzes auf farbige Petri Netze

Im Folgenden wird (größtenteils informell) beschrieben, ob und wie sich die bisher behandelten Konzepte auf farbige Petri Netze übertragen lassen. Dazu sei auf die Definition farbiger Petri Netze von Jensen in [12] verwiesen.

Der Fokus soll hier nicht auf einer formalen Erweiterung des bereits diskutierten Ansatzes liegen, sondern auf einer möglichst allgemeinen Ebene das Zusammenspiel von Farben und Produktionen behandeln. Dazu seien farbige Petri Netze wie folgt definiert:

Definition 9 (farbige Petri Netze) *Ein farbiges Petri Netz ist ein Tupel* $(P, T, A, node, In, incr)$. *Hierbei ist* P *die Menge der Stellen,* T *die Menge der Transitionen und* A *die Menge der Kanten, wobei* $T \cap P = T \cap A = P \cap A = \varnothing$ *gilt. Die Abbildung* $node : A \to (P \times T \cup T \times P)$ *legt die Start- und Zielknoten jeder Kante fest. In ist die Menge der inscriptions, die mittels* $incr : (P \cup T \cup A) \to In$ *den Stellen, Transitionen und Kanten zugeordnet werden.*

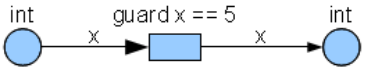


Abbildung 2.12: Beispiel eines farbigen Petri Netzes

Inscription meint hier (boolesche) Ausdrücke, Typen oder ähnliches. Wie in Abbildung 2.12 gezeigt, können *inscriptions* von Stellen den Typ der Tokens dieser Stellen beschreiben. *Inscriptions* von Transitionen können Bedingungen beschreiben, unter denen die Transition feuert. In diesem Fall sind alle Tokens der Stellen ganze Zahlen und die Transition kann nur feuern, wenn ein Token in der eingehenden Stelle den Wert 5 hat.

2.4.1 Inscriptions und Matching

Wird der in den vorigen Abschnitten vorgestellte Ansatz der Graphtransformation direkt auf farbige Petri Netze übertragen, so müssen die in den Produktionen verwendeten Netze auch *inscriptions* enthalten. Auf der einen Seite erfordert die Theorie, dass die in der Produktion verwendeten Petri Netze vom gleichen Typ wie die transformierten Petri Netze sind. Auf der anderen Seite muss es in Produktionen mindestens möglich sein, farbige Knoten zu erzeugen, da sonst kein farbiges Petri Netz aufgebaut werden kann. Beim Matching m der linken Regelseite auf das zu verändernde Netz ($m : L \to G$) muss auch die Identität der *inscriptions* überprüft werden. Sind die *inscriptions* nicht identisch, ist m nicht korrekt. Dies ist deshalb wichtig, weil durch *inscriptions* Stellen, Transitionen und Kanten nicht mehr per se gleich behandelt werden können, wenn sie nur vom Typ identisch sind. Eine Transition, die ganze Zahlen verarbeitet, soll möglichst nicht an eine Stelle angefügt werden können, die nur `Strings` enthalten darf.

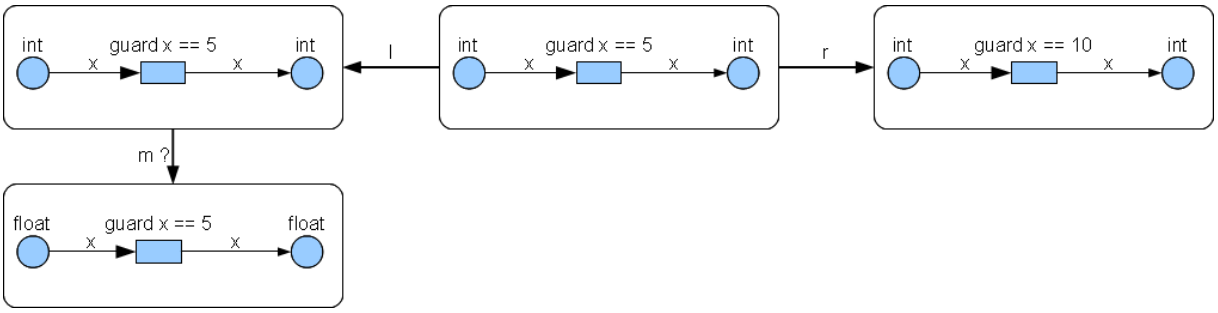


Abbildung 2.13: Anwendung einer Regel unter Verwendung einer Ordnungsrelation

Alternativ kann auch eine (partielle) Ordnungsrelation $\sqsubseteq$ auf der Menge der *inscriptions* definiert und zum Matching verwendet werden. Die Abbildung m ist dann korrekt, wenn

$$\forall x \in (P_L \cup T_L \cup A_L) : incr_L(x) \sqsubseteq incr_G(m(x))$$

gilt. Dies macht je nach Definition des farbigen Petri Netzes nur bedingt Sinn, wie Abbildung 2.13 zeigt. Nur wenn alle Operationen auf dem Typ `int` auch auf dem Typ `float` ausführbar

sind, kann die Abbildung m korrekt sein. In diesem Fall ist die Verallgemeinerung möglich, allerdings erfordert dies zusätzliche Korrektheitsprüfungen. Würde die angegebene Produktion die *inscription* `guard x == 5` durch `guard ggT(x,5) == 1` ersetzen, dürfte die Produktion nicht ohne weiteres auf das zu transformierende Netz angewandt werden, da `ggT(a,b)` nur für ganze Zahlen definiert ist.

Gleiche Einschränkungen können auch von l und r gefordert werden, diese beeinflussen aber maßgeblich die Veränderbarkeit von *inscriptions*.

2.4.2 Regeln ohne Inscriptions

Denkbar ist es, Produktionen zu definieren, die keine *inscriptions* enthalten. Deren Anwendung auf Netze, die *inscriptions* enthalten, könnte durch Ignorieren dieser gewährleistet werden. Damit können Regeln, für die die *inscriptions* nebensächlich sind (z.B. Löschen von Stellen), allgemeiner definiert werden. Bei dieser Art von Regel ergeben sich allerdings Schwierigkeiten bei der Fusion und dem Einfügen von Knoten in das Netz. Das Einfügen neuer Stellen, Transitionen oder Kanten in das Netz ist mit solchen Regeln nur dann sinnvoll, wenn der verwendete Petri Netz-Formalismus es zulässt, dass ein Objekt auch keine *inscription* (oder eine leere) enthält. D.h. In muss eine leere *inscription* beinhalten. Bei der Fusion von Objekten ist das Wissen über die *inscriptions* unabdingbar, da sonst nicht entscheidbar ist, welche *inscription* der zu fusionierenden Objekte verwendet oder wie diese kombiniert werden sollen. Nur durch Einbringung von *inscriptions* in den Regelformalismus kann die Fusion von Objekten eindeutig definiert werden.

2.4.3 Veränderung von Inscriptions

Um die Semantik farbiger Petri Netze nicht nur auf der Ebene der Stellen, Transitionen und Kanten zu verändern, sondern auch auf der Ebene der *inscriptions*, müssen *inscriptions* in den Regelformalismus miteinbezogen werden. Dies kann durch unterschiedliche Konzepte realisiert werden.

Verändern durch Löschen und Neuerzeugen

Das Verändern von *inscriptions* z.B. einer Stelle ist in jedem Falle möglich, indem die Stelle gelöscht und eine neue Stelle mit allen vorher vorhandenen Kanten und der neuen *inscription* erzeugt wird. Im DPO Ansatz bedeutet dies, dass die Stelle im linken Netz, aber nicht im Interface Netz enthalten ist. Dieser Ansatz funktioniert zwar, ist allerdings in der Praxis sehr umständlich, da ähnliche Effekte auftreten, wie bereits in 2.3.3 beschrieben. Um die *inscription* einer Stelle zu verändern, müssen alle mit dieser Stelle verbundenen Transitionen und Kanten in die linke Seite der Produktion aufgenommen werden - und damit auch bekannt sein. Da so die *inscriptions* fest mit den Objekten des Netzes verbunden sind, sorgt die *identification condition* der *gluing condition* dafür, dass für *inscriptions* keine Konflikte bei nicht injektiven Abbildungen entstehen können. Bei diesem Ansatz wird gefordert, dass ein Objekt des Netzes I und sein Bild bezüglich l bzw. r die gleiche *inscription* besitzen. Dadurch ist sichergestellt, dass keine *inscriptions* außer den Gewollten, die indirekt durch Löschen und Neuerzeugen von Objekten definiert sind, geändert werden kann.

Verändern durch direkte Manipulation

Sei zunächst angenommen, dass die in der Produktion verwendeten Morphismen l und r injektiv seien. Da bei DPO Produktionen alle Elemente des Interface Netzes nicht gelöscht werden,

muss der Formalismus der farbigen Petri Netze erlauben, dass Objekte leere *inscriptions* haben. Analog zum Verhalten des DPO Ansatzes bei Stellen, Transitionen und Kanten wird eine *inscription* gelöscht, wenn das Objekt in L diese *inscription* besitzt, sein Urbild bezüglich l in I jedoch nicht. Wiederum analog erhält jedes Objekt in I ohne *inscription* die *inscription*, die sein Bild bezüglich r in R besitzt. Ist l nicht injektiv, so ist es nötig die *identification condition* auf die *inscriptions* zu erweitern.

Es muss gefordert werden, dass für jedes Objekt in L alle seine Urbilder bezüglich l die gleichen *inscriptions* besitzen. Dies kann auch als *identification condition* für *inscriptions* bezeichnet werden, da ein Nichterfüllen bedeutet, dass eine *inscription* gleichzeitig gelöscht werden und erhalten bleiben müsste. Ist r nicht injektiv, so muss auch hier gefordert werden, dass für jedes Objekt in R alle sein Urbilder bezüglich r in I die gleiche *inscription* oder eine leere *inscription* besitzen. Wird diese Bedingung nicht erfüllt, ist es möglich, dass die Abbildung r eine *inscription* "überschreibt" und damit die alte löscht, was im DPO Ansatz auf der rechten Seite nicht geschehen darf.

Erlaubt die verwendete Formalisierung farbiger Petri Netze keine leeren *inscriptions*, so muss die direkte Änderung der *inscription* mit Hilfe von r realisiert werden. Für l wird gefordert, dass die *inscription* jedes Objekts in I und seines Bildes in L identisch sein müssen, damit sich l bezüglich der *inscriptions* neutral verhält. r unterliegt keinerlei Einschränkungen, da eine Veränderung der *inscriptions* nun explizit durch die Abbildung eines Objekts aus I auf ein Objekt aus R mit einer veränderten *inscription* geschieht. Gleichzeitig wird auf diese Weise in der Regel definiert wie die *inscription* verändert wird, wenn Objekte fusioniert werden.

Vergleich der Ansätze

Die Übertragung des DPO Ansatzes auf farbige Petri Netze bewirkt keine Veränderung des allgemeinen Verfahrens in Bezug auf das Löschen und Einfügen von Knoten und Kanten, wie es in Abschnitt 2.2.3 beschrieben ist. Allerdings müssen *inscriptions* bei der Definition der Regel beachtet werden. Dazu seien drei in diesem Abschnitt betrachtete Ansätze in Tabelle 2.1 verglichen. Wenn nicht anders angegeben, ist $x \in I$ und $\varnothing \in In$ die leere *inscription*.

2.5 Beispiel: Needham-Schroeder Schlüsselaustausch-Protokoll

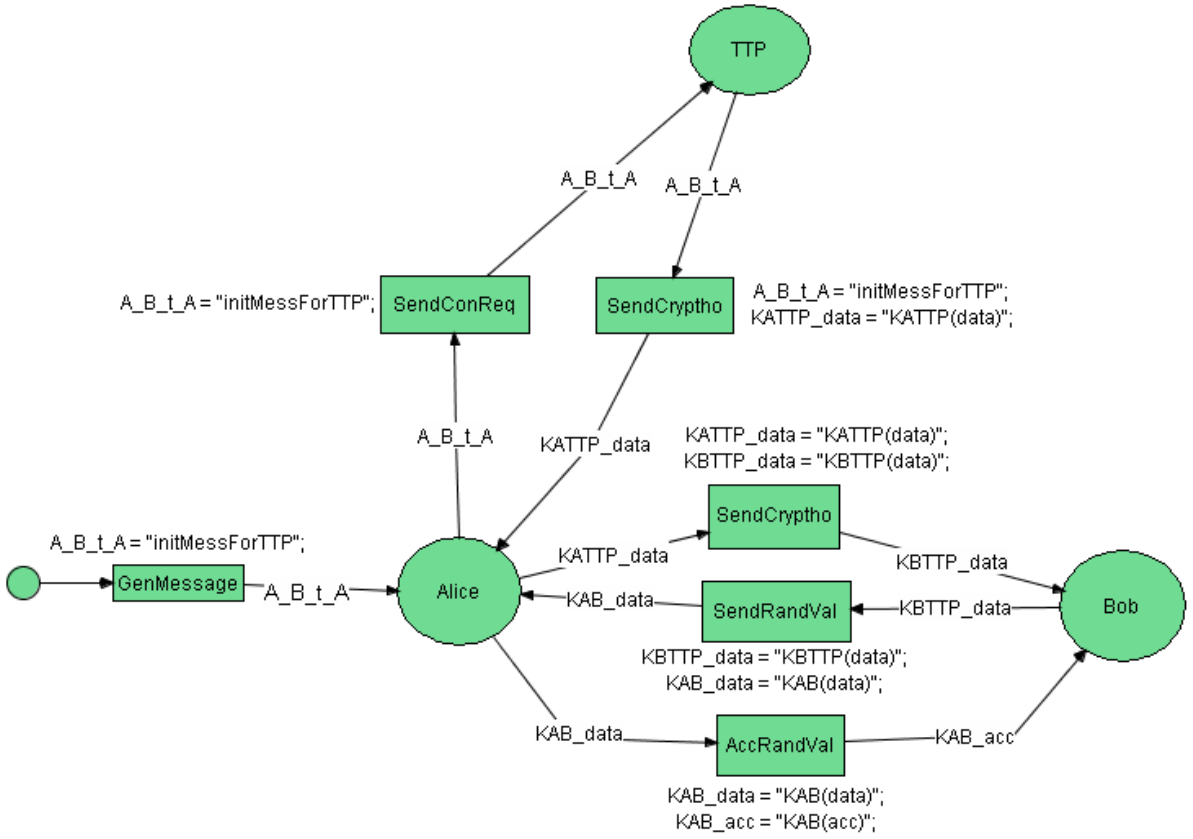
2.5.1 Verfeinerung der Protokoll Modellierung

Im Folgenden soll anhand des Needham-Schroeder Protokolls aus der Kryptographie exemplarisch gezeigt werden, wie mit Hilfe von Graphersetzungssystemen das Modell eines Algorithmus verfeinert werden kann. Dazu ist eine Variante des Needham-Schroeder Protokoll als farbiges Petri Netz modelliert, dargestellt in Netz 2.1.

Das Needam-Schroeder Protokoll ist ein Schlüsselaustausch-Protokoll für zwei Teilnehmer Alice und Bob unter Zuhilfenahme einer vertrauenswürdigen dritten Instanz, der TTP (trusted third party). Alice und Bob besitzen bereits jeweils ein Schlüssel für die Kommunikation mit der TTP, den jedoch nur sie und die TTP kennen. Das Protokoll eignet sich für den Schlüsselaustausch von zwei Kommunikationspartnern, die selber nicht in der Lage sind gute Sitzungsschlüssel zu generieren, da die Schlüsselgenerierung auf Seiten der TTP stattfindet.

Der Ablauf des Protokolls lässt sich in folgende Schritte aufteilen, welche auch in Abbildung 2.14 dargestellt sind:

1. Alice generiert einen Zeitstempel und schickt diesen zusammen mit ihrer und der Identifizierung von Bob an die TTP.



Netz 2.1: Das Needham-Schroeder Protokoll vor der ersten Verfeinerung

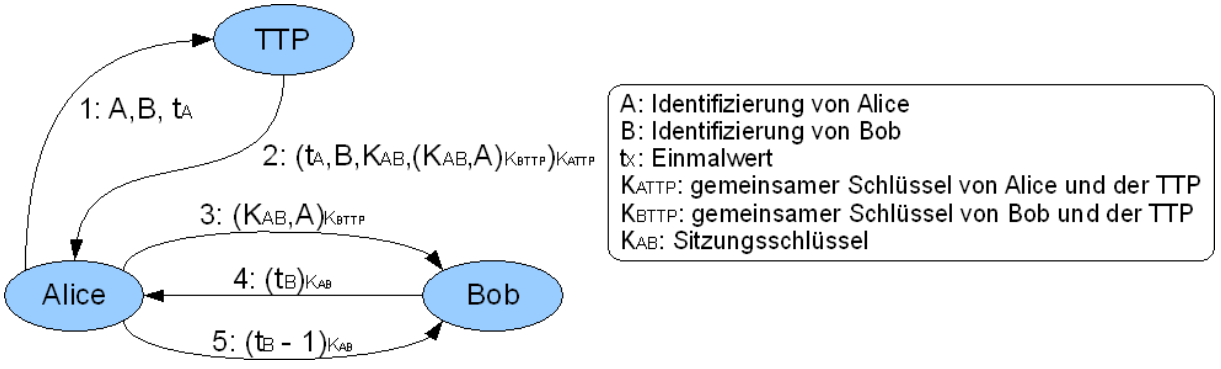


Abbildung 2.14: Ablauf des Needham-Schroeder Protokolls

Eigenschaften	Löschen und Neuerzeugen	direkte Manipulation mit leeren *inscriptions*	direkte Manipulation ohne leere *inscriptions*
Einschränkungen von m ($x \in L$)	$incr(x) = incr(m(x))$ $\vee incr(x) = \varnothing$	$incr(x) = incr(m(x))$	$incr(x) = incr(m(x))$
Einschränkungen von l	$incr(x) = incr(l(x))$	$incr(x) = incr(l(x))$ $\vee incr(x) = \varnothing$	$incr(x) = incr(l(x))$
Einschränkungen von r	$incr(x) = incr(r(x))$	$incr(x) = incr(r(x))$ $\vee incr(x) = \varnothing$	keine
Löschen von *inscriptions*	Knoten bzw. Kante löschen und ohne *inscription* einfügen	$incr(x) = \varnothing$ $\wedge incr(l(x)) \neq \varnothing$ $\wedge incr(r(x)) = \varnothing$	nicht möglich, da $\varnothing \notin In$
Einfügen von *inscriptions*	Knoten bzw. Kante löschen und mit *inscription* einfügen	$incr(x) = \varnothing$ $\wedge incr(l(x)) = \varnothing$ $\wedge incr(r(x)) \neq \varnothing$	identisch mit der Ersetzung von *inscriptions*
Ersetzen von *inscriptions*	Knoten bzw. Kante löschen und mit neuer *inscription* einfügen	$incr(x) = \varnothing$ $\wedge incr(l(x)) \neq \varnothing$ $\wedge incr(r(x)) \neq \varnothing$	$incr(x) = incr(l(x))$ $\wedge incr(x) \neq incr(r(x))$
identification condition für *inscriptions*	durch Einschränkungen von l und m immer erfüllt	$\forall x_1, x_2 \in (P_I \cup T_I \cup A_I)$: $l(x_1) = l(x_2) \Rightarrow$ $incr(x_1) = incr(x_2)$	durch Einschränkungen von l und m immer erfüllt

Tabelle 2.1: Unterschiede der Einbeziehung von *inscriptions* in Regeln

2. Die TTP generiert den Sitzungsschlüssel für die spätere Kommunikation zwischen Alice und Bob. Dieser wird mit dem gemeinsamen Schlüssel von Bob und der TTP zusammen mit der Identifizierung von Alice verschlüsselt. Das so entstandene Kryptogramm, der Sitzungsschlüssel, der Zeitstempel von Alice und die Identifizierung von Bob werden wiederum mit dem gemeinsamen Schlüssel von Alice und der TTP verschlüsselt und an Alice gesandt.

3. Alice entschlüsselt die Nachricht und überprüft, ob der Zeitstempel korrekt ist. Alice hat nun den Sitzungsschlüssel und ein Kryptogramm, das nur Bob und die TTP lesen können. Dieses leitet Alice an Bob weiter.

4. Bob entschlüsselt das Kryptogramm und ist nun ebenfalls im Besitz des Sitzungsschlüssels. Nun generiert Bob einen Einmalwert, verschlüsselt diesen mit dem Sitzungsschlüssel und sendet ihn an Alice. Dieser Einmalwert ermöglicht es Bob zu prüfen, ob auch Alice den Sitzungsschlüssel kennt.

5. Alice entschlüsselt die Nachricht und dekrementiert den enthaltenen Einmalwert. Der entstehende Wert wird mit dem Sitzungsschlüssel verschlüsselt und an Bob geschickt.

6. Bob entschlüsselt die Nachricht und überprüft, ob der erhaltene Wert korrekt ist. Ist er korrekt, so weiß Bob, dass auch Alice den Sitzungsschlüssel kennt.

Die Verfeinerung der Needham-Schroeder Protokolls findet hier in zwei Schritten statt. Auf das Netz 2.1 wird als erstes die Produktion in Abbildung 2.15 angewandt, woraus das Netz 2.2 resultiert. Auf dieses Netz wiederum wird die Produktion aus Abbildung 2.16 angewandt, die

das Netz 2.3 erzeugt. Die für die Anwendung benötigten Matching Abbildungen m_1 und m_2 sind hierbei eindeutig durch die Namen der Knoten und die *inscriptions* der Kanten bestimmt.

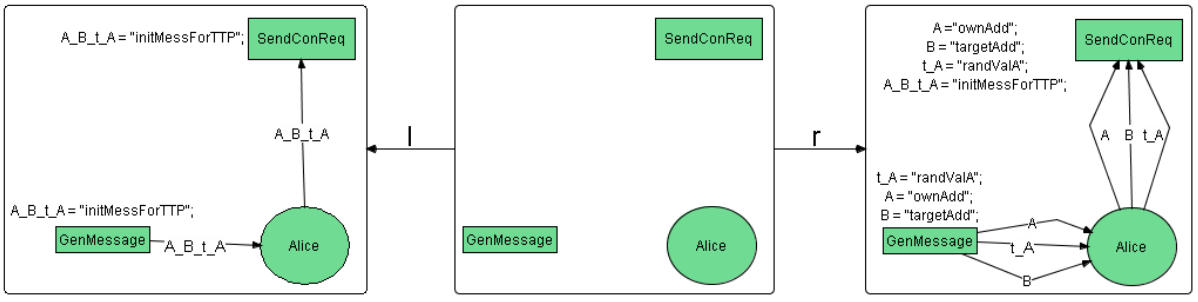


Abbildung 2.15: Erste Stufe der Verfeinerung des Needham-Schroeder Protokolls

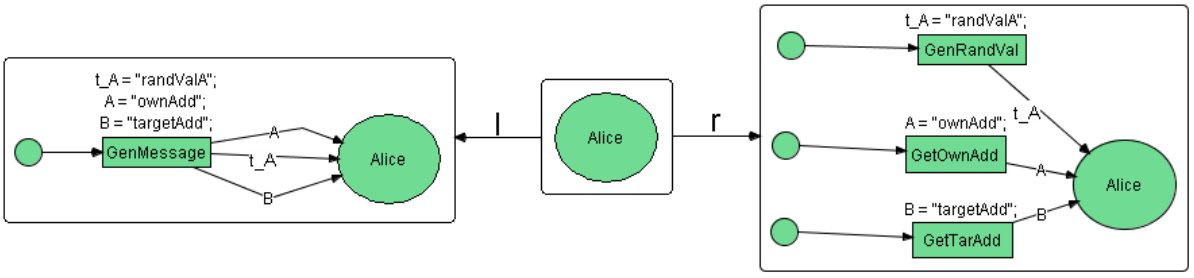
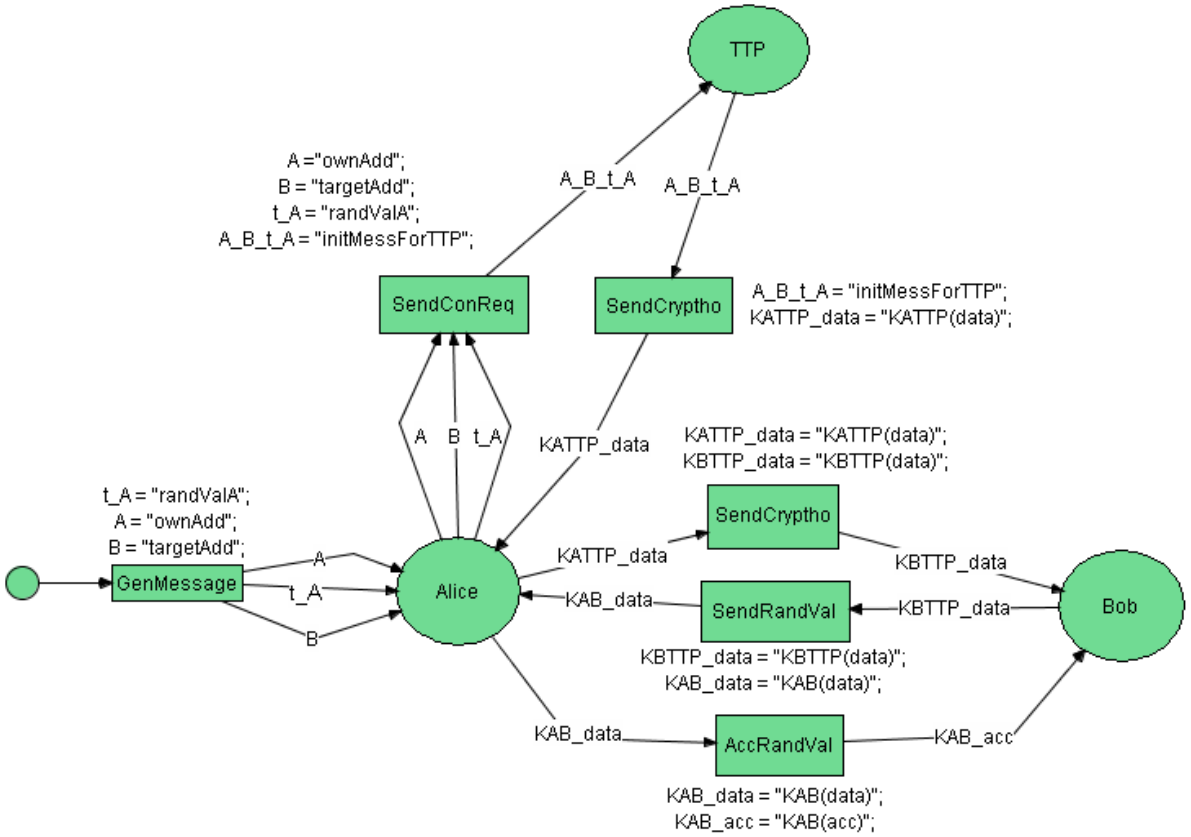


Abbildung 2.16: Zweite Stufe der Verfeinerung des Needham-Schroeder Protokolls

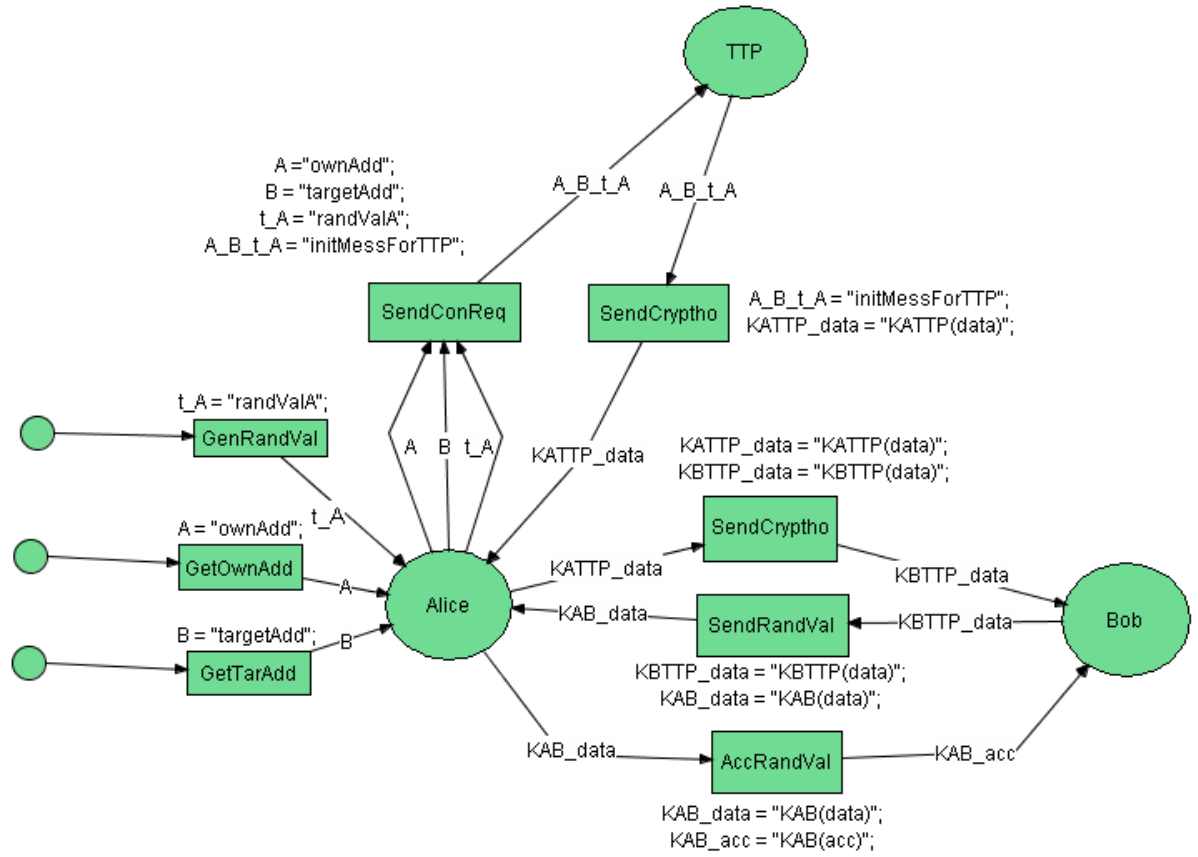
Die erste Verfeinerung (Abbildung 2.15) spaltet die Erzeugung der `ConnectionRequest`-Nachricht auf. Die Abbildungen der linken und rechten Seite sind gegeben durch die Namen der Knoten. Vor der Transformation generiert die Transition `GenMessage` die komplette Nachricht und `SendConReq` leitet diese nur weiter. Nach der Transformation generiert `GenMessage` die drei benötigten Werte getrennt und `SendConReq` setzt diese zu einer Nachricht zusammen und verschickt diese. Bei der Definition dieser Regel wurde angenommen, dass der Formalismus keine *inscriptions* erzwingt, dem entsprechend ist es möglich, die *inscriptions* direkt zu manipulieren. Bei der Anwendung der linken Seite werden beide Kanten und die *inscriptions* der Transitionen gelöscht. Auf der rechten Seite werden sechs neue Kanten mitsamt der angegebenen *inscriptions* erzeugt. Den Transitionen werden die angegebenen *inscriptions* zugeordnet.

Die zweite Verfeinerung (Abbildung 2.16) spaltet die Erzeugung der einzelnen Parameter des `ConnectionRequest` weiter auf. Die Abbildung der linken und rechten Seite bestehen dabei nur aus der Abbildung der Stelle `Alice` des Interface Netzes auf die entsprechenden Stellen im linken und rechten Netz. Im ersten Schritt wird alles außer der Stelle `Alice` gelöscht und im zweiten Schritt die entsprechenden neuen Stellen, Transitionen und Kanten mitsamt *inscriptions* erzeugt. Würde diese Transformation in die andere Richtung verlaufen, so kämen Fusionen von Knoten zum Einsatz. Die drei unbeschrifteten Stellen, ihre jeweiligen angeschlossenen Kanten sowie die drei Transitionen würden auf eine leere Stelle, eine Kante und eine Transition abgebildet. Die nicht injektiven Abbildung treten auf der rechten Seite auf und sind dadurch eindeutig auflösbar. Würde diese Regel durch eine nicht injektive Abbildung auf der linken Seite definiert, so besteht das in Abschnitt 2.2.3 beschriebene Problem. Das *pushout* Komplement ist hier nicht eindeutig und die Regel resultiert in unterschiedliche Ergebnisse, je nach konkreter Abarbeitung. Durch geschickte Definition der Produktion lässt sich das Problem in diesem Fall lösen, da die



Netz 2.2: Das Needham-Schroeder Protokoll nach der ersten und vor der zweiten Verfeinerung

aufzulösenden Knoten des Netzes mit keinem anderen Knoten verbunden sind, der nicht Teil der Regel ist.



Netz 2.3: Das Needham-Schroeder Protokoll nach beiden Verfeinerungen

2.5.2 Intruder Problem

Das Needham-Schroeder Protokoll hat eine seit längerem bekannte Schwachstelle. Das von der TTP erzeugte Kryptogramm, das Alice an Bob weiterleitet, enthält keinen Zeitstempel. Dadurch kann dieses Kryptogramm von einem unbekannten Dritten gespeichert und zur Wiedereinspielung verwendet werden. Schafft es der unbekannte Dritte sich als Alice auszugeben, so kann er nur durch Einspielen der letzten drei im Protokoll verwendeten Nachrichten eine Verbindung mit Bob aufnehmen. Der Unbekannte kennt dann zwar den Sitzungsschlüssel noch nicht, hat sich aber erfolgreich bei Bob als Alice ausgegeben.

Um diese Schwachstelle zu korrigieren, ist es notwendig, dass Bob einen Zeitstempel generiert. Bei erhaltenen Kryptogrammen kann er so prüfen, ob das Kryptogramm aktuell ist oder wieder eingespielt wurde. Zu diesem Zweck werden folgende zwei zusätzlichen Schritte durchgeführt, bevor das Protokoll beginnt.

1. Alice sendet eine Nachricht an Bob mit der Bitte einen Zeitstempel zu generieren.

2. Bob generiert einen Zeitstempel und schickt ihn an Alice.

Der von Bob generierte Zeitstempel muss von Alice an die TTP weitergeleitet werden. Diese verschlüsselt ihn zusätzlich in dem Kryptogramm, dass Alice an Bob weiterleiten wird. Bob kann so die Frische des Kryptogramms prüfen.

Dieser zusätzliche Ablauf soll nun in das Netz 2.3 mit Hilfe von Graphtransformation eingefügt werden. Dazu werden die beiden zusätzlichen Schritte in Form von zwei neuen Transitionen `AskForRandVal` und `SendRandVal` in das Netz eingefügt. Die Transition `SendConReq` und ihre *inscription* muss aktualisiert werden, so dass der von Bob generierte Zeitstempel mit in die Nachricht an die TTP einfließt. Eine Regel, die das Netz in dieser Weise umformt ist in Abbildung 2.17 dargestellt. Aus der Anwendung dieser Regel resultiert das Netz 2.4. Hier sei angemerkt, dass die drei Kanten zwischen der Stelle `Alice` und der Transition `SendConReq` nicht in der Produktion vorkommen müssen. Werden die Kanten dennoch aufgenommen, so muss das Matching m diese auf das Netz abbilden, was die Sicherheit der Anwendung der Regel erhöht.

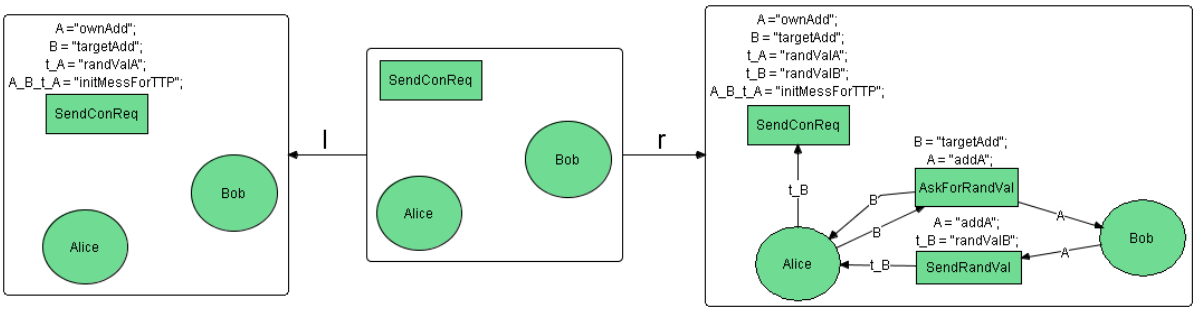
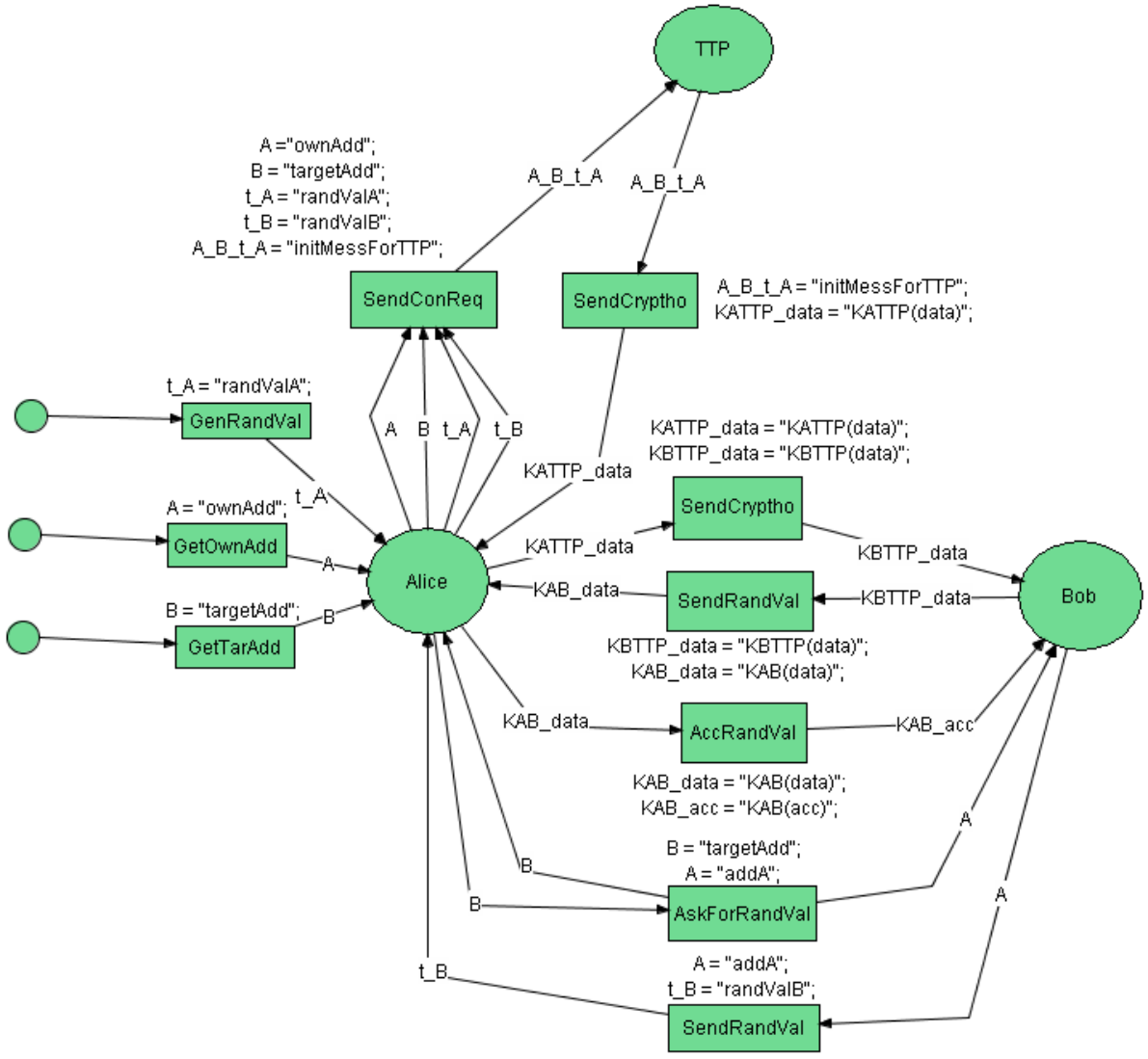


Abbildung 2.17: Regel zur Beseitigung des Intruder Problems



Netz 2.4: Needham-Schroeder Protokoll nach Beseitigung des Intruder Problems

3 Architektur und Implementierung

Zur Implementierung eines Graphtransformationssystems für Petri Netze auf Basis der in Kapitel 2 diskutierten theoretischen Grundlagen ist es zunächst notwendig, einen Transformationsprozess zu identifizieren und zu modellieren (3.1). Dieser Prozess realisiert dabei die Anwendung der in Kapitel 2 beschriebenen Regeln. Er umfasst die Definition und Instantiierung einer Produktion sowie die Anwendung dieser auf ein Petri Netz unter Verwendung eines Matchings. Darauf aufbauend müssen geeignete Datenstrukturen hergeleitet, bzw. vorhandene Standards sinnvoll in der Form erweitert werden (3.2), dass sowohl Petri Netze als auch Regeln beschrieben und schließlich auf der Ebene der Implementierung verarbeitet werden können. In diesem Zusammenhang müssen Möglichkeiten der Syntax- und Semantikprüfung betrachtet und gegebenenfalls integriert werden. Desweiteren ist es Ziel, Module der zu entwickelnden Software im Transformationsprozess zu identifizieren. Für diese Module muss zum Einen bestimmt werden, welche formale Grundlagen, z.B. eine Produktionsdefinition in XML, sie erfüllen (3.3) und zum Anderen, welche Software spezifischen Eigenschaften und Schnittstellen sie besitzen müssen (3.4). Desweiteren soll auf Probleme der automatisierten Erzeugung von Matchings eingegangen werden (3.5). Zusätzlich soll die Verwendung der Implementierung erläutert (3.6) und die durch die Implementierung abbildbaren Regeln betrachtet (3.7), sowie ein kurzer Ausblick auf mögliche Erweiterungen gegeben werden.

3.1 Konzeptuelle Prozessmodellierung

Der Aufbau eines Systems zur Graphtransformation kann auf den in Abbildung 3.1 beschriebenen Prozess abgebildet werden. Bei diesem steht dabei die Regelanwendung im Zentrum mit dem Ziel, ein gegebenes Petri Netz zu transformieren. Basierend auf den Grundlagen aus Kapitel 2 wird dazu eine Regel benötigt bestehend aus Produktion und Matching. Die Anwendung der Regel kann jedoch nur stattfinden, wenn die Produktion instantiiert ist, daher in eine Form gebracht wurde, die das System direkt verarbeiten kann. Dies ist notwendig, da vom Anwender nicht verlangt werden kann, Produktion bereits instantiiert zu definieren, da dies Kenntnis von der Implementierung des Transformationsprozesses erfordert. Bei der Instantiierung wird die vom Anwender erstellte Definition der Produktion umgewandelt in eine instantiierte Produktion, die eine Beschreibung der Schritte darstellt, die für eine Transformation durchgeführt werden müssen. Eine einmal instantiierte Produktion kann durch Kombination mit unterschiedlichen Matchings ohne erneute Instantiierung auf unterschiedliche Netze angewandt werden. Eine Produktion kann auch ohne explizite Instantiierung bei der Anwendung der Regel verwendet werden unter der Bedingung, dass die Instantiierung ein Teil der Regelanwendung darstellt. Durch die Trennung zwischen Instantiierung und Regelanwendung kann die Laufzeit des Gesamtsystems verbessert werden, da unter Anderem eine instantiierte Produktion mehrfach verwendet werden kann. Im Folgenden sei auf die einzelnen Module des in Abbildung 3.1 dargestellten Prozesses näher eingegangen.

Produktionsdefinition Am Anfang des Ablaufs steht die Produktionsdefinition, welche vom Anwender zu erzeugen ist. Dies kann manuell geschehen, falls dem Anwender ein fester

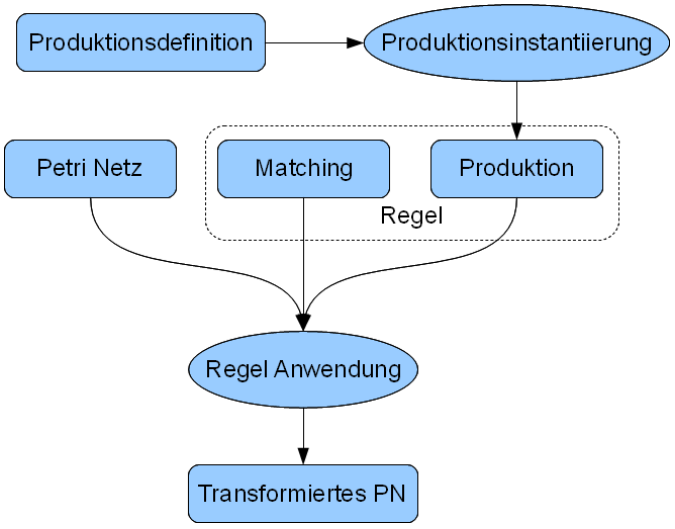


Abbildung 3.1: Konzeptueller Transformationsprozess eines Systems zur Graphtransformation

Satz von Produktionen genügt. Alternativ kann die Produktionsdefinition automatisch zur Laufzeit erzeugt werden, was eine weitaus höhere Flexibilität gewährleistet, allerdings auch weitergehende Implementierungen nach sich zieht.

Die Produktionsdefinition umfasst je nach verwendetem Formalismus der Transformation (beschrieben in Kapitel 2) mehrere Petri Netze sowie Abbildungen zwischen diesen Netzen. Das zu entwickelnde System muss dem Anwender daher eine Plattform anbieten, die ihm eine einfache Erstellung von Produktionen ermöglicht. Diese Plattform sollte dabei unabhängig von der verwendeten Programmiersprache sein. Ferner soll die Plattform frühstmöglich inkorrekte Definitionen unterbinden und möglichst einfach strukturiert sein. Im Idealfall ist die Plattform durch den Nutzer erweiterbar und würde somit die Möglichkeit bieten, an unterschiedliche Formalismen von Petri Netzen anpassbar zu sein.

Produktionsinstantiierung Bei der Instantiierung der Produktion wird aus der Produktionsdefinition eine in der Programmiersprache verarbeitbare Form der Produktion erzeugt (ab hier einfach Produktion genannt). Hierzu soll die zu implementierende Software folgende zwei Schritte sequentiell abarbeiten:

a) Im ersten Schritt wird die Produktion auf Korrektheit überprüft. Hierbei soll die Syntax der Produktion sowie die in der Produktion benutzten Netze und Abbildungen überprüft werden. Ist ein Netz syntaktisch nicht korrekt oder erfüllt eine der Abbildungen die strukturellen Bedingungen des gewählten Regelformalismus nicht, so ist der Instantiierungsprozess an dieser Stelle abzubrechen. Dazu muss der verwendete Parser der Produktionsdefinition die Schwere des Fehlers bestimmen und gegebenenfalls die Instantiierung der Produktion unterbinden.

b) Ist die Prüfung im ersten Schritt erfolgreich verlaufen, so wird in einem zweiten Schritt aus der Definition der Produktion die anwendbare Produktion erzeugt. Da die Instantiierung einmalig durchgeführt wird, soll die Instantiierungskomponente in diesem Schritt möglichst viele der in der Regel erforderlichen Berechnungen durchführen um die Performanz zu erhöhen. Welche Berechnungen an dieser Stelle bereits durchgeführt werden können, hängt dabei stark vom verwendeten Petri Netz und Regelformalismus ab. Es ist beispielsweise möglich, die Menge der zu löschenden Elemente

des Netzes zu bestimmen, so dass bei der Anwendung der Regel die Bilder dieser Elemente bezüglich des Matchings direkt gelöscht werden können. Es sind jedoch nur solche Vorausberechnungen möglich, die kein Wissen über das zu transformierende Netz benötigen. Es kann beispielsweise bestimmt werden, welche Knoten bei der Regelanwendung zu fusionieren sind, allerdings nicht, wie dies geschehen soll, da die Produktion nicht zwangsläufig über ausreichende Informationen zur Umsetzung verfügt. Der Produktion ist meist nicht einmal bekannt, wie viele Kanten an die zu fusionierenden Elemente grenzen, folglich kann keine der erwähnten Vorausberechnung, wie die zu verändernden Kanten, stattfinden.

Produktion Die Produktion ist eine anwendbare Instanz einer Produktionsdefinition. Dazu soll die Produktion als eine eigenständige Klasse von Objekten definiert werden. Die Produktion soll so unabhängig wie möglich von dem verwendeten Petri Netz-Formalismus instantiierbar sein, damit dieser austauschbar und damit veränderbar ist. Der gewählte Petri Netz-Formalismus ist hier jedoch ausschlaggebend für die Möglichkeiten der Transformation durch die Produktion, da verschiedene Petri Netz-Formalismen verschiedene Elemente besitzen, welche transformiert werden können.

Matching Eine Produktion bildet zusammen mit einem Matching eine anwendbare Regel. Unabhängig vom verwendeten Regelansatz wird ein Matching benötigt, dass angibt, auf welche Knoten bzw. Kanten des Netzes die Produktion angewandt wird. Das Matching beschreibt daher eine Zuordnung von Elementen eines Netzes aus der Produktion auf Elemente des zu verändernden Netzes. Welche Eigenschaften diese Abbildung haben muss, ist abhängig von dem gewählten Regelformalismus. Das zu entwickelnde System muss dabei sicherstellen, dass diese Eigenschaften gelten oder durch eine Prüfung inkorrekte Matchings erkannt werden können. Inkorrekte Matchings dürfen nicht Teil einer anwendbaren Regel sein, da diese während der Regelanwendung entweder unlösbare Konflikte oder inkorrekte, transformierte Petri Netze erzeugen kann.

Petri Netz Im skizzierten Prozess stellt das Petri Netz das Netz dar, welches durch die Regelanwendung transformiert werden soll. Für das Netz muss ein geeigneter Formalismus und eine Datenstruktur ausgewählt werden. Es stehen dabei eine Vielzahl von Formalismen zur Verfügung, wie etwa für Stellen/Transition Netze oder farbige Petri Netze. Der verwendete Formalismus muss in Einklang mit dem in der Produktionsdefinition verwendeten Regelformalismus stehen, da sonst das bei der Regelanwendung entstehende, transformierte Petri Netz mit großer Wahrscheinlichkeit nicht korrekt ist. Eine Regel, welche für Stellen/Transition Netze definiert ist, kann ein farbiges Petri Netz unter Umständen nicht korrekt transformieren, da die Farbe nicht abgebildet werden kann. Werden bei der Anwendung dieser Regel neue Knoten oder Kanten erzeugt, so besitzen diese keine Farbe, was nach der Definition von farbigen Petri Netzen (z.B. aus [12]) gegebenenfalls nicht erlaubt ist.

Die Wahl der Datenstruktur zur Darstellung eines Petri Netzes kann sowohl abhängig, als auch unabhängig von der gewählten Programmiersprache erfolgen. Falls vorhanden, sollte in jedem Fall auf einen anerkannten Standard zurückgegriffen werden, um eine erhöhte Kompatibilität mit anderen Systemen zu gewährleisten, die diesen Standard verwenden.

Regelanwendung Die Regelanwendung stellt die eigentliche Transformation dar. Dazu wird die Produktion, das Matching und ein zu transformierendes Netz als Eingabe benötigt. Zur Regelanwendung sollen folgende zwei Schritte sequentiell durchgeführt werden:

a) Im ersten Schritt muss geprüft werden, ob das Matching in Verbindung mit der Produktion korrekt ist. Dabei soll es optional sein zu prüfen, ob das Matching korrekt ist in Bezug auf die vom Regelformalismus geforderten Einschränkungen für Abbildungen zwischen Netzen. Ist dies anderweitig sichergestellt, so kann hierdurch ein Performanzgewinn im Bezug auf die Laufzeit der zu entwickelnden Software erzielt werden, da die Korrektheitsprüfungen nicht durchgeführt werden müssen. Diese anderweitige Sicherstellung der Korrektheit kann beispielsweise bei einer automatischen Erzeugung von Regeln stattfinden.

   Es ist obligatorisch zu prüfen, ob die vom Regelformalismus geforderten Einschränkungen gelten, unter denen eine Regel anwendbar ist. Sind diese Einschränkungen, wie beispielsweise die *gluing condition* des DPO Ansatzes (Kapitel 2.2.3) nicht erfüllt, so muss die Regelanwendung unter Rückgabe eines Fehlers abgebrochen werden.

b) Ist der erste Schritt der Regelanwendung erfolgreich verlaufen, so wird in einem zweiten Schritt die Regel ausgeführt. Dazu sollen alle Löschungen, Einfügungen und Fusionen von Knoten oder Kanten gemäß des genutzten Regelformalismus unter Verwendung der Produktion und des Matchings auf dem gegebenen Petri Netz durchgeführt werden.

Transformiertes PN Das transformierte Petri Netz stellt die Ausgabe des Systems dar und resultiert aus der Regelanwendung. Es sollte sichergestellt sein, dass dieses Netz ein korrektes Netz im Sinne des verwendeten Petri Netz-Formalismus ist. Dies bedeutet insbesondere, dass das Netz keine Kanten oder Knoten besitzen darf, die nach dem verwendeten Petri Netz-Formalismus nicht erlaubt sind. Dieses Ziel wurde bereits durch die Korrektheitsprüfungen bei der Produktionsinstantiierung und der Regelanwendung verfolgt, da fehlerhafte Regeln fehlerhafte Netze zur Folge haben können.

Das Ergebnisnetz sollte in der gleichen Datenstruktur wie das Eingabenetz vorliegen, um das sequentielle Ausführen einer Menge von Regeln zu ermöglichen.

3.2 Datenstrukturen für Petri Netze

Vor der Konstruktion der Architektur des Systems zur Graphtransformation muss eine Datenstruktur für Petri Netze gewählt werden. Die richtige Wahl hängt stark von der Zielsetzung des Systems ab. Soll ein bestehendes System um Graphtransformation erweitert werden, so ist die bereits verwendete Datenstruktur von Petri Netzen die geeignetste. Wird mehr Wert auf Anpassbarkeit und Übertragbarkeit gelegt, so sind auf XML basierende Markup Sprachen geeigneter.

Im Folgenden wird auf Anpassbarkeit Wert gelegt und deshalb ein XML-basierter Ansatz verfolgt. Dazu wird PNML verwendet, eine XML Definition für Petri Netze, die im wissenschaftlichen Bereich einen Standard darstellt. Die Konzepte von PNML sind unter anderem beschrieben in [3], und Definitionsdateien werden auf der Internetseite der Humboldt Universität Berlin[1] bereitgestellt.

[1] http://www2.informatik.hu-berlin.de/top/pnml/

3.2.1 PNML

Das Wurzelelement eines PNML Dokuments ist `pnml`, welches eine Menge von Netzen repräsentiert. Daher kann `pnml` mehrere Kind-Knoten vom Typ `net` besitzen. Ein `net` Knoten repräsentiert ein Netz, dessen Struktur durch die Knoten `transition`, `place` und `arc` definiert wird. PNML ermöglicht es, diesen Elementen weitere Eigenschaften zuzuweisen, wie einen Namen oder initiale Token-Belegung bei Stellen. Optional kann in PNML auch das visuelle Layout des Netzes durch Angabe von Koordinaten, Größen und Ähnlichem beschrieben werden. Die Definition von PNML ist darauf ausgelegt, dass Teile des Netzes durch zusätzliche Eigenschaften erweitert werden können. Schreibt ein Petri Netz-Formalismus beispielsweise vor, dass eine Kante einem bestimmten Typ zugeordnet sein muss, so kann die Definition des Knoten `arc` ohne großen Aufwand erweitert werden, indem diesem weitere Kindknoten oder Attribute zugewiesen werden. Desweiteren ist es möglich, Knoten neu zu definieren, falls weitreichende Änderungen der Definition vorgenommen werden sollen. Aus diesen Gründen eignet sich PNML insbesondere für die Definition eines erweiterten Petri Netz-Formalismus. Ein Beispiel eines PNML Dokuments, welches ein einzelnes Netz beschreibt, ist in Listing 3.1 dargestellt. Dieses Netz ist ein Stellen/Transition System nach der `ptNetb.pntd` Definition der HU Berlin, die auch auf ihrer Internetseite zu finden ist. Das Netz hat den Namen `simple_net` und besitzt eine Stelle mit der ID `p1`, eine Transition mit der ID `t1` und eine Kante mit der ID `a1`, die von der Stelle zu Transition führt. IDs werden von PNML benutzt, um Elemente von Netzen und Netze selber eindeutig zu kennzeichnen. Das Layout der Elemente des Netzes kann mit `graphics` Knoten beschrieben werden. In diesem Fall umfassen die `graphics` Knoten ausschließlich die Position (`position` und `offset`) und die Größe (`dimension`) der Elemente. Es sind jedoch auch weitere Eigenschaften konfigurierbar, wie Farbe, Schriftart, etc.. Weiterhin ist der Stelle `p1` eine initiale Markierung bestehen aus zwei Tokens durch den Knoten `initialMarking` zugeordnet.

3.2.2 RelaxNG

Um PNML nutzen und validieren zu können, muss die Struktur von PNML mit Hilfe einer XML Definitionssprache festgelegt werden. Dazu wird im Folgenden die im wissenschaftlichen Bereich verbreitete Sprache RelaxNG[2] verwendet. Diese zeichnet sich durch einen einfachen Aufbau aus, der an eine formale Grammatik angelegt ist, wobei eine RelaxNG Definition selbst wiederum ein XML Dokument ist. RelaxNG bietet alternativ auch die Möglichkeit einer so genannten kompakten Definition ohne Verwendung von XML an. Diese wird im Folgenden jedoch nicht verwendet, da sie zwar kürzer, aber durch die verwendete Schreibweise aufwändiger zu lesen ist und an dieser Stelle eine auf XML basierende Syntax verwendet werden soll. Für RelaxNG existiert bereits eine Definition der grundlegenden Struktur von PNML durch die HU Berlin[3]. Desweiteren gibt es für RelaxNG eine gute Software Unterstützung zur Definition, Validierung und Umwandlung in andere Formate wie XML Schema oder DTD. RelaxNG wird hier anderen Sprachen vor allem wegen der vorhandenen PNML Umsetzung und des einfachen Aufbaus vorgezogen.

Eine RelaxNG Definition besteht aus einem Startsymbol und mehreren Referenzelementen. Innerhalb des Startsymbols und der Referenzelemente wird die Struktur des zu beschreibenden XML Dokuments festgelegt. Dazu können XML Knoten mit Name und Attributen definiert und auf Referenzelemente verwiesen werden. Mit RelaxNG ist es zudem möglich festzulegen, wie oft Elemente vorkommen können bzw. müssen und ob die Reihenfolge der Elemente relevant ist.

[2] http://relaxng.org/
[3] http://www2.informatik.hu-berlin.de/top/pnml/pnml.html

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb" id="n1">
    <name>
      <text>simple_net</text>
    </name>
    <place id="p1">
      <name>
        <text>p1</text>
        <graphics>
          <offset x="10" y="20"/>
        </graphics>
      </name>
      <graphics>
        <position x="30" y="10"/>
        <dimension x="40" y="40"/>
      </graphics>
      <initialMarking>
        <text>2</text>
      </initialMarking>
    </place>
    <transition id="t1">
      <name>
        <text>t1</text>
        <graphics>
          <offset x="10" y="120"/>
        </graphics>
      </name>
      <graphics>
        <position x="30" y="110"/>
        <dimension x="40" y="40"/>
      </graphics>
    </transition>
    <arc id="a1" source="p1" target="t1"/>
  </net>
</pnml>
```

Listing 3.1: Beispiel eines Netzes als PNML und als Zeichnung

Zusätzlich ist es möglich, mehrere RelaxNG Definitionen zu fusionieren und dadurch komplexere Definitionen zu generieren. Listing 3.2 zeigt, wie eine RelaxNG Definitionsdatei aussehen kann.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
     datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <start>
    <choice>
      <ref name="one.element"/>
      <ref name="more.elements"/>
    </choice>
  </start>

  <define name="one.element">
    <element name="emptyElement">
      <attribute name="someID">
        <data type="ID"/>
      </attribute>
    </element>
  </define>
```

```
  <define name="more.elements">
    <element name="nonEmptyElement">
      <oneOrMore>
        <ref name="one.element"/>
      </oneOrMore>
    </element>
  </define>

</grammar>
```

Listing 3.2: Beispiel einer RelaxNG Definition

Die Nähe des Aufbaus von RelaxNG zu Grammatiken zeigt das Wurzelelement `grammar`. Der `start` Knoten gibt an, wie das Wurzelelement des definierten XML Dokuments aussehen muss. Hier wird durch das Element `choice` angegeben, dass ein Dokument korrekt ist, wenn es sich aus einem der beiden in `choice` enthaltenen Alternativen herleiten lässt. Statt `choice` kann auch `interleave` oder `group` verwendet werden, wenn alle enthaltenen Elemente im definierten XML Dokument vorkommen sollen. Bei `interleave` ist, im Gegensatz zu `group`, die Reihenfolge der enthaltenen Elemente nicht von Belang. Im speziellen Fall des Startknotens ist die Verwendung von `interleave` oder `choice` jedoch nicht möglich, da dadurch das Wurzelelement des definierten XML Dokuments nicht eindeutig wäre. Innerhalb von `choice` werden durch `ref` Knoten auf Referenzelemente verwiesen. Diese sind durch das Schlüsselwort `define` gekennzeichnet. Das Referenzelement `one.element` definiert durch das Schlüsselwort `element` einen XML Knoten des resultierenden Dokuments. Dieser heißt `emptyElement` und besitzt ein Attribut mit dem Namen `someID` vom Typ `ID`, angegeben durch das Schlüsselwort `attribute`. Das Referenzelement `more.elements` definiert einen Knoten mit dem Namen `nonEmptyElement`, der wiederum mindestens einen Kindknoten des Typs `emptyElement` enthält, da hier auf das Referenzelement `one.element` verwiesen wird.

Zur Validierung von Dokumenten mittels RelaxNG steht eine Vielzahl an Software Bibliotheken für unterschiedliche Programmiersprachen zur Verfügung, wie beispielsweise Jing für Java, Libxml2 für C und Tenuto für .NET bzw. C#. Im Folgenden wird als RelaxNG Validierer Jing[4] verwendet, da Jing eine Open Source Implementierung auf Java Basis ist, welche außerdem von der HU Berlin empfohlen wird. Zur Validierung von PNML Dokumenten stehen auf der Internetseite der HU Berlin RelaxNG Definitionen zur Verfügung. Darunter ist eine Definition für Stellen/Transition Netze (verwendet in Listing 3.1), sowie die Basisdefinition `basicPNML.rng` zu finden. Das Erstellen von eigenen Definitionen ist durch die Ableitung der Basisdefinition möglich, ohne dass die zu Grunde liegende Struktur in der Basisdefinition von neuem definiert werden muss.

3.2.3 Farbige Petri Netze mit PNML

Im Folgenden wird die PNML Definition auf farbige Petri Netze erweitert, indem von der PNML Basisdefinition `basicPNML.rng` abgeleitet wird. Dies geschieht unter Verwendung der allgemeinen Definition farbiger Petri Netze aus Kapitel 2.4. Das vollständig resultierende RelaxNG Schema ist in Anhang A zu finden.

Zur Erzeugung eines neuen PNML Netztyps muss in der Basisdefinition das Referenzelement `nettype.uri` überschrieben werden, um den neuen Netztyp von anderen Netztypen unterscheidbar zu machen. Dies geschieht durch Definition des Elements direkt innerhalb des `include` Knotens, wie in Listing 3.3 dargestellt. Hierdurch wird die ursprüngliche Definition von `nettype.uri`

[4] http://code.google.com/p/jing-trang/

bei der Einbeziehung der Basisdefinition, vollständig ersetzt. Alle anderen in der Basisdefinition definierten Referenzelemente (und das Startsymbol) sind durch den `include` Knoten auch in der abgeleiteten Definition vorhanden.

```
<include href="basicPNML.rng">
  <define name="nettype.uri" combine="choice">
    <value>refNet</value>
  </define>
</include>
```

Listing 3.3: Ersetzen des Referenzelements `nettype.uri`

Darüber hinaus sollen die Elemente `net.labels`, `arc.labels`, `place.labels` und `transition.labels` spezifiziert werden. Diese geben an, welche Kindknoten die XML Knoten `net`, `arc`, `place` und `transition` zusätzlich zu den Knoten der allgemeinen Struktur enthalten dürfen, wie beispielsweise `graphics`. Die Spezifikation geschieht durch die Definition der Referenzelemente unter Verwendung des Schlüsselworts `define`. Diese Elemente sind jedoch bereits durch das Einbinden der Basisdefinition definiert. Diese doppelte Definition muss durch Angabe des Attributs `combine` aufgelöst werden. Der `interleave` Knoten (3.2.2), bewirkt hier eine Konkatenation beider Definitionen in beliebiger Reihenfolge. Da die Definitionen dieser Elemente in der Basisdefinition leer sind, bewirkt `interleave`, dass die leeren Definitionen ignoriert werden und nur die neuen auschlaggebend sind.

```
<define name="net.labels" combine="interleave">
  <interleave>
    <optional>
      <element name="name">
        <ref name="text.node"/>
      </element>
    </optional>
    <optional>
      <element name="declaration">
        <ref name="inscription.content"/>
      </element>
    </optional>
  </interleave>
</define>

<define name="arc.labels" combine="interleave">
  <interleave>
    <optional><ref name="inscription.label"/></optional>
    <optional><ref name="arc.type"/></optional>
  </interleave>
</define>

<define name="place.labels" combine="interleave">
  <optional><ref name="inscription.label"/></optional>
</define>

<define name="transition.labels" combine="interleave">
  <optional><ref name="inscription.label"/></optional>
</define>
```

Listing 3.4: Definition zusätzlicher Kindknoten von `net`, `place`, `transition` und `arc`

Einem Netz soll ein Name in Form eines `name` Knotens zugeordnet werden. Zusätzlich soll ein Netz eine *inscription* enthalten, die auch `declaration node` genannt wird. Dieser legt globale Eigenschaften von Variablen fest, die in den übrigen *inscriptions* der Knoten und Kanten gelten.

Laut Definition 9 in Kapitel 2.4 besitzen alle Stellen, Transitionen und Kanten eine *inscription*. Das hier definierte farbige Petri Netz soll allerdings leere *inscriptions* erlauben, weshalb das Vorhandensein eines `inscription` Knotens optional ist. Ist dieser Knoten vorhanden, so muss er einen Text enthalten, der die eigentliche *inscription* darstellt. Optional kann hier das visuelle Layout der *inscription* angegeben werden. Listing 3.5 zeigt die Definition eines `inscription` Knotens.

Im Zusammenhang mit farbigen Petri Netzen werden oft besondere Typen von Kanten verwendet, wie beispielsweise *inhibitor arc*, *test arc* oder *reserve arc*. Diese Kantentypen werden bei der Simulation eines Petri Netzes unterschiedlich verarbeitet. Die Kanten *test arc* und *reserve arc* legen beispielsweise ein eingelesenes Token nach dem Feuern der mit ihnen verbundenen Transition wieder in die mit ihnen verbundene Stelle zurück. Um dies abzubilden, können Kanten zusätzlich zur *inscription* einen `type` Knoten enthalten, der angibt, um welche Art von Kante es sich handelt. Ist der Typ einer Kante nicht explizit angegeben, so ist die Kante eine Standardkante.

```
<define name="inscription.label">
  <element name="inscription">
    <ref name="inscription.content"/>
  </element>
</define>

<define name="inscription.content">
  <interleave>
    <optional>
      <element name="graphics">
        <ref name="annotationgraphics.content"/>
      </element>
    </optional>
    <ref name="text.node"/>
  </interleave>
</define>
```

Listing 3.5: Definition der *inscription* als XML Knoten

3.3 Produktionsdefinition

Als Regelformalismus wird im Folgenden der DPO Ansatz aus Kapitel 2.2.3 verwendet. Dies geschieht in erster Linie auf Grund des Verhaltens dieses Ansatzes in Konfliktfällen. Durch die Prüfung der *gluing condition* (Definition 6) wird der Anwender in einem gewissen Rahmen auf eine fehlerhafte Definition der Produktion oder des Matchings aufmerksam gemacht, was ihm die Möglichkeit der Korrektur bietet. Zudem ist die *gluing condition* ein einfach zu implementierender Sicherheitsmechanismus. Im Gegensatz dazu würde beim SPO Ansatz (2.2.2) die Regel ohne Angabe eines Fehlers ausgeführt und der Anwender würde zu einem deutlich späteren Zeitpunkt bemerken, dass das Ergebnis der Transformation nicht seinen Erwartungen entspricht. Dies resultiert direkt daraus, dass jede Regel des SPO Ansatzes anwendbar ist. Ein etwaiger Fehler äußert sich meist durch das Fehlen von Elementen im resultierenden Netz, da beim Auftreten eines Konflikts das Löschen von Elementen dem Erhalten von Elementen vorgezogen wird.

Ein anderer Vorteil des DPO Ansatzes ist die Aufteilung der Produktion in einen rechten und linken Teil und die damit verbundene Trennung der Definition von Löschungen und Einfügungen. Beim SPO Ansatz werden alle Elemente des linken Netzes gelöscht, die kein Bild im rechten Netz besitzen und damit auch Knoten, die bei der Definition der Produktion vergessen wurden abzubilden. Durch die Verwendung von totalen Homomorphismen, kann dies beim DPO Ansatz

nicht geschehen. Dieser Fehler wird bereits bei der Instantiierung der Produktion erkannt. Zusammengefasst liegt die Stärke des DPO Ansatzes in der frühzeitigen Erkennung von einfachen Fehlern. Der größte Nachteil dieses Ansatzes liegt jedoch in der fehlenden Eindeutigkeit bei der Berechnung des *pushout* Komplements (siehe Kapitel 2.2.3). Daher muss bei der Implementierung der Regelanwendung eine geeignete, eindeutige Verarbeitung festgelegt werden.

Im Folgenden soll eine Datenstruktur geschaffen werden, mittels derer DPO Produktionen definiert werden können. Eine Produktion des DPO Ansatzes umfasst dabei drei Netze, das Interface Netz I, die linke Seite der Produktion L und die rechte Seite der Produktion R. Darüber hinaus werden zwei Abbildungen $l : I \to L$ und $r : I \to R$ benötigt.

An dieser Stelle wird ein der formalen Beschreibung naher Ansatz verfolgt, bei dem Netze und Abbildungen explizit in der entwickelten Datenstruktur angegeben werden. Die Definition der Datenstruktur geschieht in den folgenden Kapiteln unter Verwendung von RelaxNG und der bereits oben angesprochenen PNML Definition.

Alternativ ist auch ein Ansatz denkbar, bei dem die Netze und Abbildungen der Produktion nicht explizit definiert werden. Es ist beispielsweise möglich, nur das Netz der linken Seite L und die Abbildungen zu definieren, so dass sich die anderen Netze implizit ergeben. l ist dann keine Abbildung nach L mehr, sondern bildet jedes Element aus L auf einen Wahrheitswert ab, der angibt, ob das Element gelöscht werden soll oder nicht. Nicht-Injektivität der linken Seite muss zusätzlich gesondert berücksichtigt werden. r ist keine Abbildung mehr im mathematischen Sinne, sondern gibt, z.B. durch Kommandos an, welche Elemente eingefügt oder fusioniert werden sollen. Bei Benutzung einer solchen Definition muss auch die Implementierung der Prüfungsroutine der *gluing condition* entsprechend der anderen Datenstruktur angepasst werden.

3.3.1 Generische Produktionsdefinition

Um noch mehr Anpassbarkeit zu bieten, wird an dieser Stelle die Struktur der Produktionsdefinition so unabhängig von der Petri Netz-Definition wie möglich gestaltet. Dazu wird zunächst eine generische Produktionsdefinition erzeugt. Diese Definition ist insofern generisch, als sie die Struktur der Produktion vorgibt, allerdings ohne die Petri Netz-Definition vorzuschreiben. Eine vollständige Produktionsdefinition ergibt sich aus der Kombination einer generischen Produktionsdefinition und einer Petri Netz-Definition. Prinzipiell kann hier auch eine Petri Netz-Definition verwendet werden, die nicht auf PNML aufbaut, solange die Elemente des Netzes über IDs gekennzeichnet sind. Bei der späteren Interpretation der Regel muss die Petri Netz-Struktur jedoch bekannt sein, da sonst nicht sichergestellt werden kann, dass das resultierende Petri Netz in jedem Fall korrekt ist. Sowohl die generische, als auch die vollständige Produktionsdefinition sind in Anhang A mit zusätzlichen Kommentaren aufgeführt und werden im Folgenden ausschnittweise erläutert.

```
<define name="trans.rule">
  <element name="rule">
    <interleave>
      <ref name="trans.left"/>
      <ref name="trans.interface"/>
      <ref name="trans.right"/>
      <ref name="trans.mapping"/>
    </interleave>
  </element>
</define>
```

```
<start>
  <ref name="trans.rule"/>
</start>
```

Listing 3.6: Definition der groben Struktur einer Produktion

Das Wurzelelement der generischen Produktionsdefinition ist der Knoten `rule`, wie in Listing 3.6 dargestellt. Dieser enthält die vier Kindknoten `interface`, `deleteNet`, `insertNet` und `mapping` in beliebiger Reihenfolge, die eine Produktion vollständig bestimmen. Der Knoten `interface` beschreibt das Interface Netz, der Knoten `deleteNet` beschreibt die linke Produktionsseite L und der Knoten `insertNet` beschreibt die rechte Produktionsseite R. Jeder dieser Knoten verweist aus das Referenzelement `net.definition`. Dieses Element muss bei der Generierung einer konkreten Produktionsdefinition auf eine Petri Netz-Definition verweisen. In der generischen Produktionsdefinition ist dieses Referenzelement jedoch leer, wie in Listing 3.7 dargestellt. Durch diese Konstruktion wird bei der Korrektheitsprüfung der Produktion automatisch die Korrektheit der verwendeten Netze geprüft.

```
<define name="trans.left">
  <element name="deleteNet">
    <ref name="net.definition"/>
  </element>
</define>

<define name="trans.interface">
  <element name="interface">
    <ref name="net.definition"/>
  </element>
</define>

<define name="trans.right">
  <element name="insertNet">
    <ref name="net.definition"/>
  </element>
</define>

<define name="net.definition">
  <empty/>
</define>
```

Listing 3.7: Definition der drei in der Produktion verwendeten Netze

Zusätzlich zu den drei Netzen muss in der Produktion auch die Abbildung des Interface Netzes I auf die linke Produktionsseite L und auf die rechte Produktionsseite R definiert sein. Dies geschieht durch einen `mapping` Knoten, dargestellt in Listing 3.8. Der `mapping` Knoten enthält eine beliebige Anzahl von `mapElement` Knoten. Für jede im Interface Netz benutzte ID muss ein `mapElement` Knoten existieren, dessen `interfaceID` Attribut dieser ID entspricht. Die Werte der Attribute `deleteID` und `insertID` stellen das Bild der `interfaceID` in L und R dar. Dabei ist `deleteID` das Bild der `interfaceID` auf der linken Produktionsseite und `insertID` das Bild der `interfaceID` auf der rechten Produktionsseite.

```
<define name="trans.mapping">
  <element name="mapping">
    <zeroOrMore>
      <ref name="trans.mapping.element"/>
    </zeroOrMore>
  </element>
</define>
```

```
<define name="trans.mapping.element">
  <element name="mapElement">
    <attribute name="interfaceID">
      <data type="ID"/>
    </attribute>
    <attribute name="deleteID">
      <data type="ID"/>
    </attribute>
    <attribute name="insertID">
      <data type="ID"/>
    </attribute>
  </element>
</define>
```

Listing 3.8: Definition der beiden in der Produktion verwendeten Abbildungen

3.3.2 Konkrete Produktionsdefinition

Um aus der generischen eine konkrete Produktionsdefinition zu erzeugen, müssen die Petri Netz-Definition und die generische Produktionsdefinition fusioniert werden. Dabei müssen zwei Dinge beachtet werden:

a) Zum Einen muss das Startsymbol der Petri Netz-Definition durch ein leeres Element überschrieben werden, da beide fusionierten Dokumente Startsymbole besitzen und daher sonst kein eindeutiges Startsymbol definiert wäre.

b) Zum Zweiten muss die Petri Netz-Definition der Produktion `net.definition` auf die verwendete Petri Netz-Definition verweisen. Dazu wird hier auf `net.element` verwiesen, was in der Petri Netz-Definition den `net` Knoten definiert. Die Erzeugung einer vollständigen Produktionsdefinition erfolgt in Listing 3.9 unter Verwendung der in Abschnitt 3.2.3 erläuterten Petri Netz-Definition (`refPNML.rng`) und der generischen Produktionsdefinition (`generic_rule.rng`).

```
<include href="refPNML.rng">
  <start combine="interleave">
    <empty/>
  </start>
</include>

<include href="generic_rule.rng"/>

<define name="net.definition" combine="interleave">
  <ref name="net.element"/>
</define>
```

Listing 3.9: Konkretisierung der generischen Produktionsdefinition für farbige Petri Netze

Das hier verwendete Prinzip der generischen Produktionsdefinition ermöglicht eine einfache Anpassung der Produktionsdefinition auf beliebige Petri Netztypen. Die Implementierung der Regelanwendung kann allgemein stattfinden, so dass alle auf diese Weise definierten vollständigen Produktionsdefinitionen unterstützt werden.

Beispiel

Im Folgenden soll der Aufbau einer Produktion anhand eines Beispiels einer Produktion verdeutlicht werden, welches graphisch in Abbildung 3.2 dargestellt ist. Die Abbildungen l und

r sind hier nicht explizit aufgeführt, da sie sich vollständig aus der Struktur der verwendeten Netze ergeben. Selbst die Eindeutigkeit der Abbildung der oberen Stelle und der dazugehörigen Kante des Interface Netzes ergeben sich aus den verwendeten *inscriptions*.

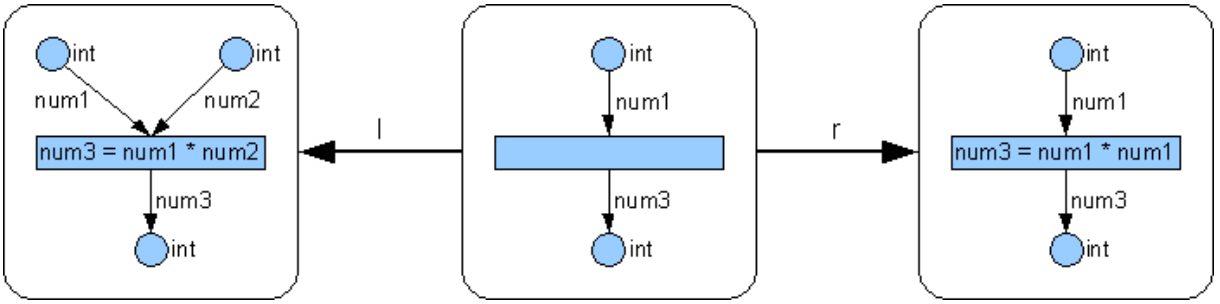


Abbildung 3.2: Grafische Darstellung der im Folgenden verwendete Produktion

Als Produktionsdefinition wird die in Abschnitt 3.3.2 definierte, konkrete Produktionsdefinition verwendet. Zur Erstellung einer Produktion müssen daher vier XML Knoten definiert werden, jeweils einer für jedes der drei verwendete Netze und einer für beide Abbildungen.

Dazu seien zunächst die Abbildungen definiert, wie in Listing 3.10 gezeigt. Der Einfachheit halber werden in allen drei verwendeten Netzen die gleichen IDs verwendet, so dass die ID eines Elements des Interface Netzes mit den IDs seiner Bilder in der linken und rechten Produktionsseite übereinstimmen. Dies ist möglich, da die Produktion keine Fusionen von Elementen vorsieht und damit die Abbildungen injektiv sind. Der `mapping` Knoten ist hier ausreichend definiert, da er ein `mapElement` Kindknoten für jede ID des Interface Netzes enthält.

```
<mapping>
  <mapElement interfaceID="id1" deleteID="id1" insertID="id1"/>
  <mapElement interfaceID="id2" deleteID="id2" insertID="id2"/>
  <mapElement interfaceID="id9" deleteID="id9" insertID="id9"/>
  <mapElement interfaceID="id13" deleteID="id13" insertID="id13"/>
  <mapElement interfaceID="id14" deleteID="id14" insertID="id14"/>
</mapping>
```

Listing 3.10: Die in der Produktion verwendeten Abbildungen

Das umfangreichste der drei Netze ist das Netz der linken Produktionsseite, welches durch den `deleteNet` Knoten beschrieben wird. Dieser enthält einen in Listing 3.11 dargestellten `net` Knoten, dessen Struktur in der Petri Netz-Definition festgelegt ist. Dieses Netz wird bei der Regelanwendung durch das Matching auf das zu transformierende Netz abgebildet und muss daher sowohl die zu löschenden Elemente enthalten, als auch Elemente, welche das zu transformierende Netz besitzen muss. Dies gilt auch für Kindknoten der Elemente des Netzes. Kindknoten beschreiben in diesem Fall nur die *inscriptions* von Elementen (`inscription`) und die Typen der Kanten (`type`). Es wurde bewusst auf Layout Knoten wie `graphics` verzichtet, um die Produktion unabhängig von dem Layout des zu transformierenden Netzes zu definieren.

```
<deleteNet>
  <net id="net1" type="refNet">
    <transition id="id1">
      <inscription><text>num3 = num1 * num2</text></inscription>
    </transition>
    <place id="id2">
      <inscription><text>int</text></inscription>
    </place>
    <place id="id3">
```

```
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id9" source="id2" target="id1">
      <inscription><text>num1</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
    <arc id="id10" source="id3" target="id1">
      <inscription><text>num2</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
    <place id="id13">
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id14" source="id1" target="id13">
      <inscription><text>num3</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
  </net>
</deleteNet>
```

Listing 3.11: Die in der Produktion verwendete linke Seite

Das Interface Netz (`interface`) entspricht dem Netz der linken Seite nach Löschung der zu löschenden Elemente, wie in Listing 3.12 zu sehen ist. Die zu löschenden Elemente des Netzes sind hier die Stelle `id3` und die Kante `id10`, da kein Element des Interface Netzes auf sie abgebildet wird. Zusätzlich wird die *inscription* der Transition `id1` gelöscht, da sie nicht im Urbild der Transition `id1` vorkommt. Alle anderen Knoten des Interface Netzes sind mit ihren Bildern im linken Netz identisch und werden daher nicht verändert.

```
<interface>
  <net id="net1" type="refNet">
    <transition id="id1"/>
    <place id="id2">
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id9" source="id2" target="id1">
      <inscription><text>num1</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
    <place id="id13">
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id14" source="id1" target="id13">
      <inscription><text>num3</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
  </net>
</interface>
```

Listing 3.12: Das in der Produktion verwendete Interface Netz

Der einzige Unterschied des Netzes der rechten Produktionsseite (`insertNet`), dargestellt in Listing 3.13, und des Interface Netzes liegt in der *inscription* der Transition. Diese ist nur auf der rechten Produktionsseite definiert und wird daher neu erzeugt.

```
<insertNet>
  <net id="net1" type="refNet">
    <transition id="id1">
      <inscription><text>num3 = num1 * num1</text></inscription>
    </transition>
```

```
    <place id="id2">
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id9" source="id2" target="id1">
      <inscription><text>num1</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
    <place id="id13">
      <inscription><text>int</text></inscription>
    </place>
    <arc id="id14" source="id1" target="id13">
      <inscription><text>num3</text></inscription>
      <type><text>ordinary</text></type>
    </arc>
  </net>
</insertNet>
```

Listing 3.13: Die in der Produktion verwendete rechte Seite

Mehrere in der Produktion angegebene Knoten sind in allen drei Netzen identisch, darunter z.B. die Stelle `id13` und die Kante `id14`. Diese Elemente werden während der Regelanwendung nicht verändert und können daher weggelassen werden. Dadurch, dass sie in der Produktion aufgeführt sind, stellen sie jedoch eine zusätzliche Bedingung an Netze, auf welche diese Produktion angewandt werden kann, da das Matching diese Elemente abbilden muss. Durch das Vorhandensein der Stelle `id13` und der Kante `id14` wird beispielsweise erzwungen, dass die Transition des zu transformierenden Netzes mindestens eine ausgehende Kante besitzt.

3.4 Programmarchitektur

Nach der Festlegung einer Datenstruktur und einer Plattform zur Definition von Produktionen, ist es nun Ziel, den in Abschnitt 3.1 beschriebenen konzeptuellen Prozess zu implementieren. Dazu wird mit der objektorientierten Programmiersprache Java[5] gearbeitet. Die Verarbeitung von XML Dokumenten geschieht dabei auf Basis der Standardbibliothek von Java, während zu Validierung von XML Dokumenten auf Jing zurückgegriffen wird. Zur Realisierung der Implementierung ist es erforderlich, Klassen für die unterschiedlichen Module und Teilprozesse zu definieren. Dazu gehören Klassen zur Definition von Abbildungen zwischen Netzen (`Mapper`, `ID`, etc.), zur Fehlerbehandlung (`RuleRelatedException`, etc.), zur Instantiierung von Produktionen (`RuleBuilder`) und zur Repräsentation einer instantiierten Produktion (`Rule`). Die Implementierung der Regelanwendung findet dabei innerhalb der `Rule` Klasse statt. Die Abbildung der wichtigsten Klassen der Implementierung auf den in Abschnitt 3.1 vorgestellten Prozess ist in Abbildung 3.3 dargestellt. Der Aufbau und die Wirkungsweise dieser Klassen wird im Folgenden näher erläutert.

Abbildung 3.4 zeigt die drei wichtigsten Klassen der Programmarchitektur. Die Klasse `Rule` ist entgegen ihrer Bezeichnung keine vollständige Regel, sondern repräsentiert das Ergebnis einer Instantiierung einer Produktion. Die Instantiierung dieser Klasse geschieht durch die Klasse `RuleBuilder`. Dazu muss der Klasse `RuleBuilder` durch Aufruf der Methode `setRule(String)` eine Regel zugeordnet werden. Dies geschieht durch die Angabe einer URL, die auf eine Produktionsdefinition verweist, wie sie in Abschnitt 3.3 definiert ist. Durch den Aufruf der `buildRule()` Methode wird das `Rule` Objekt generiert.
Die Regelanwendung geschieht hier durch den Aufruf der Methode `apply(String,Mapper)` des

[5] http://www.sun.com/java/

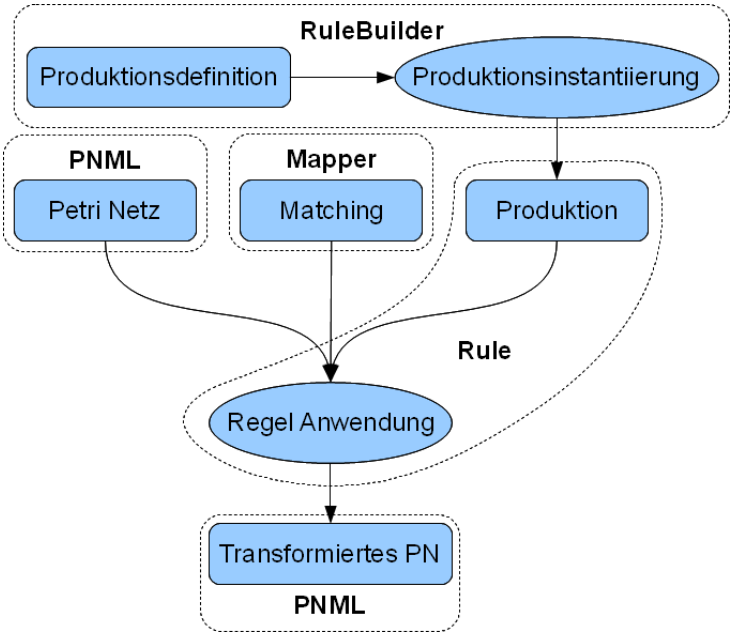


Abbildung 3.3: Abbildung der Implementierung auf den in Abschnitt 3.1 vorgestellten Prozess

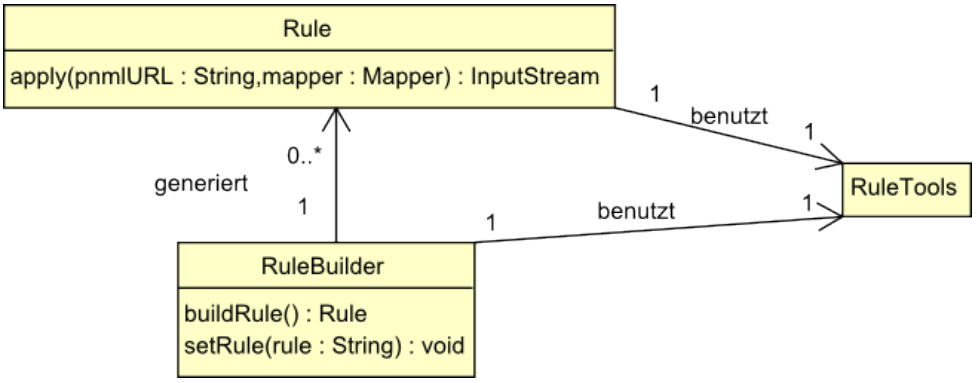


Abbildung 3.4: Überblick über den Kern der Programmarchitektur

`Rule` Objekts. Hierfür muss eine URL eines PNML Dokuments, sowie ein Matching, realisiert durch die Klasse `Mapper` (3.4.1), angegeben werden. Das Resultat ist ein `InputSteam` Objekt, welches zur Rückgabe des transformierten Petri Netzes verwendet wird.

Die Klasse `RuleTools` stellt dem gesamten System eine Auswahl von statischen Methoden zur Verfügung. Diese werden vor allem in den Klassen `Rule` und `RuleBuilder` zur Instantiierung der Produktion und zur Anwendung der Regel verwendet. Die bereitgestellten Methoden sind allgemein implementiert und dienen meist der Vereinfachung der Verarbeitung von XML Dokumenten.

3.4.1 Abbildungen zwischen Netzen

Für die Abbildungen zwischen den Netzen, die bei der Produktionsdefinition und dem Matching verwendet werden, wird eine geeignete Implementierung in Java benötigt. Diese wird durch das `Mapper` Interface realisiert, wie in Abbildung 3.5 dargestellt. Dazu wird die Eigenschaft der Definition von `place`, `transition` und `arc` Knoten eines PNML Dokuments genutzt, dass jedes dieser Elemente eine eindeutige ID besitzt. Die Methode `transformID(String)` bildet dabei die ID eines Elements des Ursprungsnetzes auf eine ID eines Elements des Bildnetzes ab. Nicht jedes instantiierte `Mapper` Objekt ist jedoch eine korrekte Abbildung, da der verwendete Regelformalismus Bedingungen an die Eigenschaften von Abbildungen stellt. Diese Bedingungen ergeben sich zum Einen aus der Definition des DPO Ansatzes für allgemeine Graphen (2.2.3) und zum Anderen aus den Überlegungen zur Erweiterung des Ansatzes auf farbige Petri Netze (2.4.3).

Im Folgenden wird die Veränderung von *inscriptions* durch direkte Manipulation realisiert, da dieser Ansatz wenig Löschungen von Knoten benötigt. In der Anwendung ist dies von Vorteil, da dadurch die Veränderung von *inscriptions* unabhängig von anderen Eigenschaften, wie dem graphischen Layout geschehen kann. Um dies umzusetzen, wird der Ansatz der Ordnungsrelation auf *inscriptions* aus Kapitel 2.4.1 aufgegriffen. Dazu sei eine **Teil von** Beziehung zwischen zwei XML Knoten wie folgt definiert:

Definition 10 (Teil von) *Ein XML Knoten A ist ein **Teil von** einem XML Knoten B, genau dann, wenn folgende zwei Bedingungen gelten:*

1. *B besitzt jedes Attribut, welches A besitzt und diese Attribute haben identische Werte.*
2. *Für jeden Kindknoten A' von A existiert ein Kindknoten B' von B, sodass A' und B' identisch sind oder A' ist ein **Teil von** B'.*

Die Ordnungsrelation auf *inscriptions* wird mit Hilfe dieser Definition auf alle Kindknoten von Elementen des Netzes ausgeweitet. ID Attribute müssen an dieser Stelle ausgenommen sein, da sie zur Identifizierung von Elementen der Netze dienen und damit im Regelfall keine identischen Werte besitzen. Eine Abbildung ist nach diesem Ansatz genau dann korrekt, wenn jedes Element des Ursprungsnetzes ohne seine ID Attribute ein **Teil von** seinem Bild ist. Das ID Attribut eines Knotens ist `id`. Die ID Attribute von Kanten umfassen jedoch zusätzlich zum `id` Attribut auch die Attribute `target` und `source`.

Die Verwendung der **Teil von** Beziehung bewirkt eine hohe Flexibilität der Implementierung, da die Ordnungsrelation ohne Wissen über den verwendeten Regelformalismus arbeitet. Dieser muss dem System nur bekannt gemacht werden, um eine syntaktische Korrektheitsprüfung der Produktion zu ermöglichen.

Die hier verwendete direkte Manipulation von *inscriptions* (2.4.3) geschieht unter Verwendung von leeren *inscriptions*, da bei diesem Ansatz das Löschen und Einfügen von *inscriptions*

konsistent zu der Löschung und Einfügung von Knoten und Kanten durchgeführt wird. Auch die Löschungen von *inscriptions* sollen daher nur durch die linke Seite einer Produktion definiert werden. Analog soll das Einfügen von *inscriptions* nur durch die rechte Produktionsseite definiert werden. Leere *inscriptions* sind dabei realisiert durch das Fehlen des `inscription` Knotens.

Die direkte Manipulation von *inscriptions* kann alternativ an dieser Stelle auch ohne Verwendung der leeren *inscription* realisiert werden. Die Veränderung von *inscriptions* würde dann ausschließlich auf Basis der rechten Abbildung der Produktion stattfinden (2.4.3). Dieser Ansatz wurde hier jedoch nicht verwendet, da die Verarbeitung von *inscriptions* nicht konsistent zum Rest des DPO Ansatzes ist. Desweiteren wurde nicht auf die indirekte Manipulation von *inscriptions* durch Löschen und neu Einfügen von Knoten und Kanten zurückgegriffen, da Löschungen von Elementen des Netzes weitest möglich vermieden werden sollen, wenn Veränderungen ausreichend sind.

Folgende Bedingungen muss ein `Mapper` Objekt erfüllen, damit es eine korrekte Abbildung darstellt und damit es bei der Bearbeitung der Regel verwendet werden kann:

1. Jede im Ursprungsnetz verwendete ID muss auf eine ID des Bildnetzes abgebildet werden. Für jede mögliche ID, die nicht im Ursprungsnetz verwendet wird, muss die `transformID(String)` Methode des `Mapper` Objekts daher `null` zurückgeben. Ferner müssen die hier verwendeten IDs auf existierende XML Knoten verweisen.

2. Für jede ID des Ursprungsnetzes und die ihr zugeordnete ID des Bildnetzes müssen die von ihnen referenzierten XML Knoten vom gleichen Typ sein. Das heißt, dass diese Knoten beide `place`, `transition` oder `arc` Knoten sein müssen.

3. Wie in den theoretischen Grundlagen in Kapitel 2.2.3 erläutert, muss die Abbildung der IDs einer Kante strukturerhaltend sein. Für jede ID, welche auf einen `arc` Knoten verweist, müssen daher zusätzliche Bedingungen gelten. Das Bild der ID des `target` Attributs eines `arc` Knotens muss identisch sein mit der ID des `target` Attributs des `arc` Knotens, den das `Mapper` Objekt mit dem ursprünglichen `arc` Knoten assoziiert. Gleiches gilt für das `source` Attribut eines `arc` Knotens.

4. Die Urbilder jedes Knotens des Bildnetzes müssen eine **Teil von** diesem sein (siehe Definition 10). Die Attribute `id`, `target` und `source` sind dabei zu vernachlässigen.

Zusätzlich zur Transformation von IDs bietet das Java-Interface `Mapper` die Methode `getSourceIDs()` an, welche eine Liste aller IDs zurückgibt, die ein Bild besitzen. Die Methode `getTargetIDs()` gibt eine Liste aller IDs zurück, auf die abgebildet werden.

Zur besseren Verwaltung der IDs wird eine Klasse `ID` definiert, die eine Verknüpfung aus ID Wert (hier ID Name genannt) und dem Typ des Knotens darstellt, welcher von dieser ID referenziert wird. Diese Klasse spezifiziert durch den in ihr enthaltenen Aufzählungstyp `TYPES` die drei möglichen Typen `PLACE` (Stelle), `TRANSITION` (Transition) und `ARC` (Kante).

Als weitere Klasse zur Verwaltung der IDs wurde die Klasse `IDSpace` implementiert. Diese Klasse bietet die Möglichkeit, alle IDs eines Netzes geordnet zu speichern. Ein Objekt dieser Klasse speichert daher drei unabhängige Listen für die Stellen, Transitionen und Kanten eines Netzes, auf die durch die in Abbildung 3.5 dargestellten Methoden `getPlaceIDs()`, `getArcIDs()` und `getTransitionIDs()` unabhängig voneinander zugegriffen werden kann.

3.4.2 Fehlerbehandlung

Während der Bearbeitung von Regeln der Transformation können Fehler auftreten, wie beispielsweise fehlerhaft spezifizierte Produktionen in XML oder Matchings, die die in Abschnitt 3.4.1

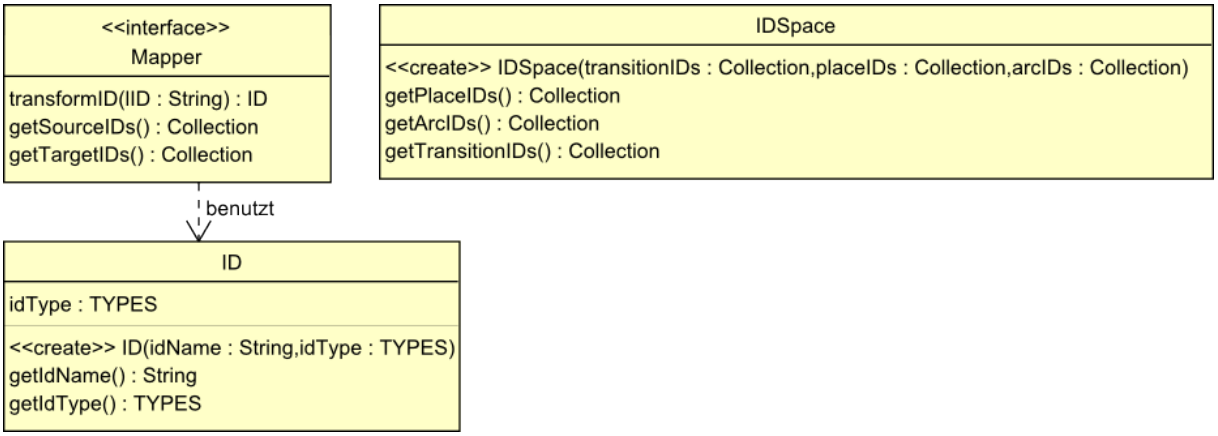


Abbildung 3.5: Klassenstruktur zur Realisierung von Abbildungen

geforderten Einschränkungen nicht erfüllen. Um diese Fehler abzufangen und für den Anwender transparent zu machen, sind die `Exceptions` `RuleRelatedException`, `RuleBuildingException` und `RuleApplicationException` definiert. Die `RuleRelatedException` bildet dabei die Oberklasse der anderen beiden `Exceptions` und beinhaltet von beiden verwendete Fehlercodes, wie in Abbildung 3.6 dargestellt. Beide abgeleiteten `Exceptions` können durch Angabe eines Fehlercodes oder durch die Kapselung einer Fehlermeldung einer anderen `Exception` erzeugt werden. Die `RuleBuildingException` kann bei der Instantiierung der Produktion auftreten, während die `RuleApplicationException` bei der Anwendung der Regel geworfen werden kann. Die von den `Exceptions` verwendeten Fehlercodes beschreiben Fehler, wie das Verstoßen gegen die an `Mapper` Objekte gestellten Bedingungen (3.4.1), dem Verstoßen gegen die *gluing condition* oder auch allgemeinen Problemen bei der Verarbeitung von XML Dokumenten. Die Gründe, welche das Werfen dieser `Exceptions` verursachen, werden in den folgenden Kapiteln zum Zeitpunkt ihres Auftretens erläutert.

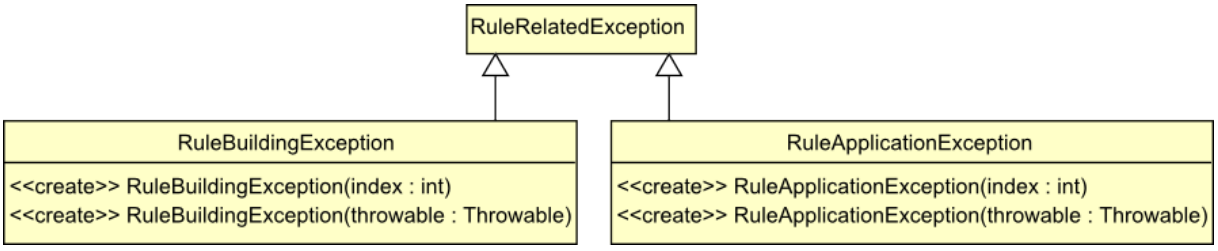


Abbildung 3.6: Aufbau der im System verwendeten `Exceptions`

3.4.3 Produktionsinstantiierung

Die Instantiierung der Produktion erfolgt durch ein `RuleBuilder` Objekt (siehe Abbildung 3.7). Dazu muss die `RuleBuilder` Klasse selbst instantiiert werden. Bei dieser Instantiierung oder durch einen Aufruf der `setRule(String)` Methode muss dem `RuleBuilder` Objekt eine Produktion zugewiesen werden. Die beim Aufruf der `setRule(String)` Methode oder der Erzeugung des `RuleBuilder` Objekts angegeben URL muss auf eine Produktion in Form eines XML Dokuments (3.3) verweisen. Eine Unterstützung anderer Datenstrukturen für Petri Netze ist hier zunächst nicht vorgesehen. Wenn die `buildRule()` Methode aufgerufen wird, ohne vorher eine

Produktion angegeben zu haben, wird eine `RuleBuildingException` geworfen.

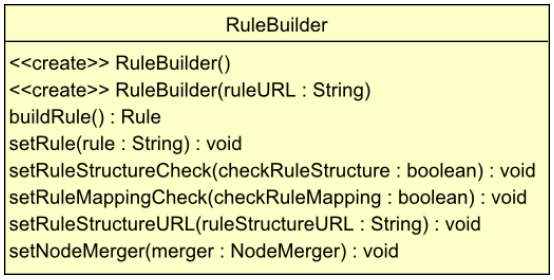


Abbildung 3.7: Die wichtigsten Methoden der `RuleBuilder` Klasse

Zur Erzeugung des `Rule` Objekts wird folgender Algorithmus ausgeführt:

1. Zu Beginn wird geprüft, ob eine Produktion als URL angegeben ist. Ist dies nicht der Fall, so wird eine `RuleBuildingException` geworfen.

2. Eine syntaktische Prüfung der Produktion wird durchgeführt, um fehlerhaft spezifizierte Produktionen zu erkennen. Dazu wird die Produktion geladen und anhand der in Abschnitt 3.3 vorgestellten RelaxNG Definition einer vollständigen Produktion validiert. Soll nicht auf die vorgestellte RelaxNG Definition eines farbigen Petri Netzes zurückgegriffen werden, so kann durch die Methode `setRuleStructureURL(String)` eine eigene Produktionsdefinition angegeben werden. Dazu muss eine URL zu einer korrekten RelaxNG Definition, die auf der generischen Produktionsdefinition basiert (siehe Abschnitt 3.3.1), angegeben werden. Eine inkorrekte Produktionsdefinition führt zu einer `RuleBuildingException` bei dem Versuch eine Produktion zu validieren, da die Produktionsdefinition von Jing zurückgewiesen wird. Auch der Versuch eine inkorrekte Produktion zu instantiieren, wird eine `RuleBuildingException` verursachen, die den Grund der Inkorrektheit genau spezifiziert.

   Dieser komplette Schritt wird nicht ausgeführt, wenn im Vorhinein die Methode `setRuleStructureCheck(boolean)` mit dem Wert `false` aufgerufen wurde.

3. Die drei in der Produktion angegebenen Netze werden extrahiert. Dafür wird aus der angegebenen Produktion eine DOM (Document Object Model) Repräsentation generiert und die Knoten `interface`, `deleteNet` und `insertNet` extrahiert.

4. Aus jedem im vorigen Schritt extrahierten Netz wird ein `IDSpace` Objekt erzeugt (siehe Abschnitt 3.4.1). Dieses Objekt wird vor allem bei der Regelanwendung benötigt, wird jedoch aus Performanzgründen bereits hier generiert.

5. Die in der Produktion angegebenen Abbildungen der linken und rechten Produktionsseite l und r werden ausgelesen und zwei `Mapper` Objekte erzeugt, die diese Abbildungen realisieren (siehe 3.4.1).

6. Die Korrektheit der im vorigen Schritt erzeugten `Mapper` Objekte wird unter Berücksichtigung der in Abschnitt 3.4.1 gestellten Bedingungen für die Korrektheit von `Mapper` Objekten überprüft. Die Überprüfung findet dabei durch folgende Prozedur statt, welche auf jede ID des Interface Netzes angewandt wird:

   a) Das Bild der gegebenen ID bezüglich des `Mapper` Objekts wird bestimmt. Ist dies `null`, so wird abgebrochen.

b) Die referenzierten Knoten der gegebenen ID (Ursprungsknoten) und ihres Bildes (Bildknoten) werden aus dem Netz extrahiert. Kann einer dieser Knoten nicht gefunden werden, so wird abgebrochen.

c) Die Typen (`place`, `transition` oder `arc`) des Ursprungsknotens und des Bildknotens werden auf Gleichheit geprüft. Sind sie nicht gleich, so wird abgebrochen.

d) Wurde im vorigen Schritt festgestellt, dass beide Knoten vom Typ `arc` sind, so werden die `source` und `target` Attribute beider Knoten extrahiert. Entspricht die ID des `source` bzw. `target` Attributs des Bildknotens nicht dem Bild der ID des `source` bzw. `target` Attributs des Ursprungsknotens, so wird abgebrochen.

e) Es wird überprüft ob der Ursprungsknoten ein **Teil von** dem Bildknoten ist (siehe Definition 10), wobei die Attribute `id`, `source` und `target` nicht beachtet werden. Dazu wird bestimmt, ob für jedes Attribut des Ursprungsknotens ein Attribut des Bildknotens existiert, welches den gleichen Namen und Wert besitzt. Ist dies nicht der Fall, so wird abgebrochen. Anschließend werden die Kindknoten des Ursprungs- und des Bildknotens nach Namen gruppiert. Für jeden Kindknoten des Ursprungsknotens wird nun geprüft, ob er ein **Teil von** einem Kindknoten des Bildknotens ist, der den gleichen Namen besitzt. Dies geschieht rekursiv. Ist ein Kindknoten des Ursprungsknotens nicht ein **Teil von** einem der Kindknoten des Bildknotens, so wird abgebrochen.

Wird bei der Überprüfung der Bedingungen vorzeitig abgebrochen, so wird eine `RuleBuildingException` geworfen, die angibt, welche der Bedingungen verletzt wurde und die Instantiierung wird abgebrochen.

Dieser komplette Schritt wird nicht ausgeführt, wenn im Vorhinein die Methode `setRuleMappingCheck(boolean)` mit dem Wert `false` aufgerufen wurde.

7. Sind alle vorhergehenden Schritte erfolgreich durchgeführt worden, so wird abschließend das `Rule` Objekt mit den Ergebnissen der durchgeführten Berechnungen instantiiert und zurückgegeben.

An dieser Stelle ist bereits die Konfiguration der Fusion von Elementen des Netzes durch die Methode `setNodeMerger(NodeMerger)` möglich. Dies wird im folgenden Kapitel näher erläutert.

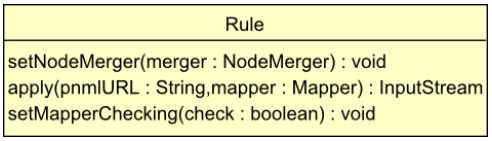


Abbildung 3.8: Die wichtigsten Methoden der Klasse `Rule`

3.4.4 Regelanwendung

Ist durch die Instantiierung erfolgreich ein `Rule` Objekt (siehe Abbildung 3.8) generiert worden, so kann dieses direkt angewandt werden, ohne weitere Konfigurationen vornehmen zu müssen. Dazu wird die Methode `apply(String,Mapper)` verwendet unter Angabe eines Netzes sowie eines `Mappers`. Der erste Parameter muss eine URL sein, die auf ein PNML Dokument verweist. Die Definition dieses PNML Dokuments muss identisch sein mit der in der Produktionsdefinition

verwendeten PNML Definition. Das angegebene `Mapper` Objekt muss alle IDs des linken Netzes L der Produktion auf das gegebene Petri Netz abbilden. Dabei müssen alle in Abschnitt 3.4.1 angegebenen Bedingungen für die Korrektheit eines `Mappers` gelten.

Wird die Methode `apply(String,Mapper)` aufgerufen, so wird folgender Algorithmus ausgeführt:

1. Das durch die URL angegebene Netz wird geladen und daraus eine DOM Repräsentation erzeugt. Zusätzlich wird ein `IDSPace` Objekt für das gegebene PNML Dokument generiert.

2. In diesem Schritt wird die Korrektheit des angegebenen `Mapper` Objekts überprüft. Dazu wird durch die bereits bei der Produktionsinstantiierung (3.4.3) beschriebenen Prozedur die Bedingungen aus Abschnitt 3.4.1 überprüft und bei einem negativen Ergebnis eine `RuleApplicationException` erzeugt.
   Dieser komplette Schritt wird nicht ausgeführt, wenn im Vorhinein die Methode `setMapperChecking(boolean)` mit dem Wert `false` aufgerufen wurde.

3. Die in Kapitel 2.2.3 erläuterte *gluing condition* wird überprüft. Diese Bedingung setzt sich zusammen aus den drei Unterbedingungen *dangling condition*, *identification condition* und *identification condition* für *inscriptions* (siehe Kapitel 2.4.3). Ist eine dieser Bedingungen nicht erfüllt, so verbietet der DPO Formalismus die Anwendung der Regel, weshalb eine `RuleApplicationException` geworfen wird. Die Prüfung dieser Bedingungen findet durch folgenden Algorithmus statt:

   a) Es wird die Menge aller Knoten des linken Netzes L der Produktion bestimmt, welche ein Urbild im Interface Netz I besitzen. Die Berechnung geschieht dabei durch die Bildung einer Menge aus allen Bildern von Knoten des Interface Netzes I. Sei diese Menge mit GP bezeichnet.

   b) Um die *dangling condition* zu prüfen, werden zuerst alle Kanten des zu verändernden Netzes ausgelesen und mit dem Aspekt gespeichert, welche IDs sie verbinden. Danach werden die IDs aller Knoten des zu verändernden Netzes bestimmt, die ein Urbild bezüglich des als Matching angegebenen `Mapper` Objekts besitzen. Seien diese als "bekannte" IDs bezeichnet, da diese IDs in der Produktion explizit bekannt sind und durch diese verarbeitetet werden (ggf. ohne verändert zu werden). Zuletzt wird die Menge DP bestimmt, die jede ID des linken Netzes enthält, dessen Bild im zu transformierenden Netz verbunden ist mit einer ID, welche nicht zu den bekannten IDs gehört. Dies geschieht unter Verwendung der zu Anfang bestimmten Verbindungen von IDs.

      Die *dangling condition* ist genau dann erfüllt, wenn $DP \subseteq GP$ gilt. Ist dies nicht der Fall, wird eine `RuleApplicationException` geworfen.

   c) Zur Prüfung der *identification condition* müssen alle Knoten des zu transformierenden Netzes bestimmt werden, die mehr als ein Urbild bezüglich des als Matching angegebenen `Mapper` Objekts besitzen, da diese nicht gelöscht werden dürfen. Dies geschieht durch Abbildung aller IDs des linken Netzes auf das zu transformierende Netz und durch Zählen aller Vorkommen jeder abgebildeten ID. Mit diesem Wissen wird die Menge IP bestimmt, die alle IDs des linken Netzes enthält, die auf eine ID abgebildet werden, die mehr als ein Urbild besitzt.

      Die *identification condition* ist genau dann erfüllt, wenn $IP \subseteq GP$ gilt. Ist dies nicht der Fall, wird eine `RuleApplicationException` geworfen, die dies beschreibt.

d) Für die Prüfung der *identification condition* für *inscriptions* wird das `Mapper` Objekt der linken Produktionsseite und das als Matching angegebene `Mapper` Objekt konkateniert und die inverse Abbildung der entstehenden Abbildung bestimmt. Für jede ID des zu transformierenden Netzes, welche eine Bildmenge bezüglich der inversen Abbildung besitzt, werden alle von den IDs der Bildmenge referenzierten Knoten auf Identität geprüft. Zur Prüfung der Identität zweier Knoten wird bestimmt, ob der erste Knoten ein **Teil von** dem zweiten Knoten ist und ob der zweite Knoten ein **Teil von** dem ersten Knoten ist. Die Attribute `id`, `target` und `source` werden bei dieser Prüfung nicht berücksichtigt. Sind nicht alle Knoten identisch, so wird eine `RuleApplicationException` geworfen.

4. Es wird ermittelt, welche Knoten und Kanten des zu transformierenden Netzes gelöscht werden müssen. Anschließend wird diese Löschung durchgeführt. Dazu wird der Bildbereich des `Mapper` Objekts der linken Produktionsseite bestimmt. Danach wird für jede ID des linken Netzes geprüft, ob sie in diesem Bildbereich liegt. Jede ID, die nicht in diesem Bildbereich liegt, wird auf das zu transformierende Netz abgebildet und der dort referenzierte Knoten gelöscht.

5. Nach der Löschung von Knoten und Kanten werden in einer ähnlichen Weise *inscriptions* und andere Kindknoten der Elemente des Netzes gelöscht. Dazu wird der Unterschied zwischen jedem Element des Interface Netzes (kleiner Knoten genannt) und seinem Bild in der linken Produktionsseite (großer Knoten genannt) bestimmt und dieser Unterschied im zu transformierenden Netz entfernt. Die Bezeichnung "kleiner Knoten" und "großer Knoten" rührt daher, dass durch die Forderungen am `Mapper` Objekte (3.4.1) der Knoten des Interface Netzes ein **Teil von** seinem Bild ist und deshalb nach der **Teil von** Ordnungsrelation kleiner ist als sein Bild. Die Bestimmung des Unterschieds zwischen dem Knoten des Interface Netzes und seinem Bild wird durch die folgende rekursive Prozedur erreicht:

   a) Jedes Attribut des großen Knotens wird gelöscht, wenn der kleine Knoten kein Attribut desselben Namens enthält.

   b) Jeder Kindknoten des großen Knotens wird gelöscht, wenn der kleine Knoten keinen Kindknoten desselben Namens besitzt.

   c) Diese Prozedur wird rekursiv entlang der XML Baumstruktur für jeden Kindknoten des großen Knotens ausgeführt, wenn der kleine Knoten einen Kindknoten desselben Namens besitzt.

   Wird ein direkter oder indirekter Kindknoten eines Elements des linken Netzes als zu löschender Knoten erkannt, so gibt es einen Pfad in der DOM Baumstruktur, welcher vom Knoten des Elements zu dem zu löschenden Knoten führt. Dieser Pfad wird im zu transformierenden Netz nachvollzogen, und an der entsprechenden Stelle der Baumstruktur wird jeder Knoten gelöscht, wenn der zu löschende Knoten des linken Netzes ein **Teil von** diesem Knoten ist. Es wird daher jeder Knoten des zu transformierenden Netzes gelöscht, der dem zu löschenden Knoten entspricht. Durch die an das verwendete `Mapper` Objekt gestellten Bedingungen ist sichergestellt, dass es mindestens einen Knoten gibt, der diese Bedingung erfüllt.

6. Analog zu Löschung von *inscriptions* erfolgt auch das Einfügen von neuen *inscriptions*. Der Unterschied zwischen jedem Element des Interface Netzes (kleiner Knoten genannt) und seinem Bild in der rechten Produktionsseite (großer Knoten genannt) wird bestimmt

und in das zu transformierende Netz eingefügt. Die Bestimmung des Unterschieds wird durch die folgende rekursive Prozedur erreicht:

a) Jedes Attribut des großen Knotens wird mit seinem Wert neu eingefügt, wenn der kleine Knoten kein Attribut desselben Namens enthält.

b) Jeder Kindknoten des großen Knotens wird neu eingefügt, wenn der kleine Knoten keinen Kindknoten desselben Namens besitzt.

c) Diese Prozedur wird rekursiv für jeden Kindknoten des großen Knotens ausgeführt, wenn der kleine Knoten einen Kindknoten desselben Namens besitzt.

Für das Einfügen von Knoten wird der Pfad, der vom Element des rechten Netzes zum einzufügenden Kindknoten führt, bestimmt. Dieser Pfad wird im zu transformierenden Netz nachvollzogen und der einzufügende Knoten an der entsprechenden Stelle eingefügt.

7. Nach der Einfügung von *inscriptions* werden auch Knoten und Kanten in das zu transformierende Netz eingefügt. Dazu werden die IDs des `IDSpace` Objekts des rechten Netzes durchlaufen und alle IDs bestimmt, die nicht im Bildbereich des auf der rechte Seite verwendeten `Mapper` Objekts liegen. Alle XML Knoten, welche von den so bestimmten IDs referenziert sind, werden in das zu transformierende Netz eingefügt.

8. Ist die rechte Produktionsseite nicht injektiv, so finden Fusionen von Elementen des zu transformierenden Netzes G statt. Dazu wird für jedes Element des Netzes der rechten Produktionsseite R die Menge der Urbilder im Interface Netz I bestimmt. Alle Bilder in G der Elemente dieser Menge werden fusioniert. Dazu wird ein `NodeMerger` Objekt verwendet, wie es in Abbildung 3.9 dargestellt ist. Bei Verwendung des Standard `NoKnowledgeNodeMerger` Objekts enthält das Ergebnis jeden Kindknoten beider zu fusionierenden Objekte. Ist dies nicht erwünscht, so kann eine eigene Implementierung des Interfaces erstellt werden. Dazu muss die Methode `mergeNode(Node,Node)` implementiert werden. Diese wird im Falle einer Fusion mit zwei `transition`, `place` oder `arc` Knoten aufgerufen. Der erste angegebene Knoten (`main`) muss bei Aufruf der `mergeNode(Node,Node)` Methode so verarbeitet werden, dass er nach der Verarbeitung die Fusion beider angegebenen Knoten darstellt. Die Attribute `id`, `target` und `source` dürfen dabei nicht verändert werden, da die Aktualisierung der IDs bereits an anderer Stelle durchgeführt wird. Veränderungen des zweiten angegebenen Knotens (`drained`) wirken sich nicht aus. Zur Verwendung eines eigenen `NodeMerger` Objektes muss die Methode `setNodeMerger(NodeMerger)` mit dem zu verwendenden Objekt aufgerufen werden.

9. Das veränderte PNML Dokument kann durch ein zurückgegebenes `InputStream` Objekt ausgelesen werden.

Das bei der Regelanwendung resultierende PNML Dokument kann nach erfolgreichem Durchlaufen aller aufgeführten Schritte erneut transformiert, serialisiert oder in anderen Bereichen der Software, die das Graphtransformationssystem benutzt, verwendet werden.

Das in Kapitel 2.2.3 angesprochene Problem der fehlenden Eindeutigkeit des *pushout* Komplements bei einer nicht-injektiven linken Produktionsseite ist hier ohne das Teilen von Elementen des zu transformierenden Netzes gelöst. Durch diese Konvention wird Eindeutigkeit erlangt. Eine nicht-injektive linke Produktionsseite kann die Regelanwendung jedoch auf andere Weise beeinflussen. Dies ist in Kombinationen mit einer nicht-injektiven rechten Produktionsseite der Fall. Zwei zu fusionierende Elemente des Interface Netzes werden nicht fusioniert, wenn ihr Bilder im zu transformierenden Netz bereits identisch sind. Ferner wird der Schritt der Einfügung von *inscriptions* für ein Element des zu transformierenden Netzes mehrfach durchgeführt, wenn dieses

mehrere Urbilder besitzt. Definiert eine Produktion beispielsweise für zwei Knoten des Interface Netzes eine Einfügung von *inscriptions*, so müssen beide Einfügungen stattfinden, unabhängig davon ob sie auf dem gleichen Knoten des zu transformierenden Netzes ausgeführt werden oder nicht, da die Regel sonst nicht vollständig angewendet würde. Eine mehrfache Einfügungen von *inscriptions* kann jedoch dazu führen, dass ein Element des transformierten Netzes eine Art von Kindknoten nach der Regelanwendung mehrfach besitzt, obwohl der verwendete Petri Netz-Formalismus dies verbietet.

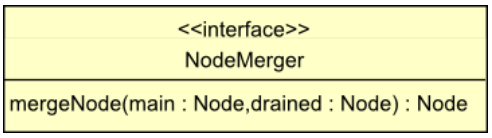


Abbildung 3.9: Das `NodeMerger` Interface

3.4.5 Alternative Lösung durch XSLT

Die Veränderung des PNML Dokuments geschieht in der Implementierung durch die direkte Veränderung der DOM Repräsentation. Da die verwendete Datenstruktur für Petri Netze auf XML basiert, liegt es jedoch auch nahe, eine XML Transformationssprache zu verwenden. Daher soll im Folgenden diese alternative Lösungsmöglichkeit unter Verwendung von XSLT betrachtet werden. XSLT ist eine XML-basierte Markupsprache zur Transformation von XML Dokumenten. Sie gehört zur XSL Familie[6] des W3C. Zur Nutzung von XSLT in Java stellt die Standard Java API bereits eine Schnittstelle zu Verfügung, welche hier verwendet werden kann.

Soll eine Regel durch XSLT realisiert werden, so muss ein XSLT-Stylesheet generiert werden. Ein XSLT-Stylesheet besteht dabei aus mehreren so genannten *templates*, wie beispielhaft in Listing 3.14 dargestellt. Ein `template` Knoten besitzt ein `match` Attribut, welches einen XPath[7] Ausdruck enthält, der angibt, auf welche Knoten des zu transformierenden XML Dokuments das *template* angewandt werden soll. XPath ist eine Abfragesprache für XML Dokumente, die ebenfalls vom W3C entwickelt wurde. Sie wird dazu verwendet, XML Knoten durch Angabe von Pfaden, Knoten oder Attributen zu identifizieren. Die Veränderungen, die ein *template* bewirkt, werden durch den Inhalt des `template` Knotens definiert.

Die Anwendung eines XSLT-Stylesheets auf ein XML Dokument beginnt mit dem Wurzelelement. Für dieses werden anwendbare *templates* gesucht und das *template* ausgeführt, dass in seinem `match` Attribut die stärksten Einschränkungen spezifiziert. Bei der Transformation von PNML Dokumenten umfassen die Einschränkungen vor allem die Namen und IDs von Elementen des Netzes, um diese eindeutig zu kennzeichnen. Das *template* aus Listing 3.14 kann auf jeden Knoten und jedes Attribut angewandt werden und wird damit immer verwendet, wenn kein anderes *template* anwendbar ist. Der Inhalt des *templates* bewirkt, dass das aktuell verarbeitete Objekt unverändert kopiert und die Verarbeitung der Transformation auf alle Kindknoten ausweitet wird.

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

Listing 3.14: Ein XSLT *template*, dass Knoten ohne Veränderung kopiert

[6] http://www.w3.org/Style/XSL/
[7] http://www.w3.org/TR/xpath/

Wird XSLT als Basis der Transformation von PNML Dokumenten verwendet, so besteht die Regelanwendung im Wesentlichen aus der Anwendung eines XSLT-Stylesheets auf ein PNML Dokument. Es ist dabei möglich das XSLT-Stylesheet bereits teilweise bei der Instantiierung der Produktion zu generieren. Ohne Wissen über das zu verändernde Netz können die *templates* zur Löschung und Einfügung von Knoten generiert werden.

Das Löschen eines Elements des Netzes kann einfach durch ein leeres *template* realisiert werden, dessen `match` Bedingung nur von dem zu löschenden Knoten erfüllt wird. Die Identifizierung bestimmter Elemente des Netzes geschieht durch das Vorkommen ihrer IDs im `match` Attribut. Wie auch in der durchgeführten Implementierung (3.4) wird hier die Eindeutigkeit der IDs ausgenutzt. Ist dieses Attribut beispielsweise definiert als `*[@id='t1']`, so wird das dazugehörige *template* nur auf die Knoten angewandt, die ein Attribut `id` mit dem Wert `t1` besitzen.

Da ein XSLT-Stylesheet ebenfalls ein XML Dokument ist, kann das Einfügen von Elementen realisiert werden, indem der gesamte einzufügende Knoten in das *template* eingefügt wird, dass den `net` Knoten verarbeitet.

Schwieriger zu realisieren ist die Veränderung von *inscriptions* und die Fusion von Knoten. Die Schwierigkeit der Veränderung von *inscriptions* liegt darin, dass nur bestimmte Kindknoten eines Elements des Netzes entfernt oder eingefügt werden. Diese genau zu identifizieren ist bei Verwendung von XSLT schwierig zu realisieren, da sie keine eindeutigen Merkmale wie IDs besitzen.

Da die Abarbeitung eines XSLT-Stylesheets die Knoten des XML Dokuments getrennt betrachtet, wird auch die Realisierung der Fusion von Knoten des Netzes erschwert. Die einfachste Möglichkeit ist die Löschung der zu fusionierenden Knoten und Einfügung eines neuen Knotens, der das Ergebnis der Fusion repräsentiert. Damit dabei keine Kindknoten der zu fusionierenden Knoten gelöscht werden, müssen diese Knoten und damit das zu transformierende Netz bei der Erzeugung des zugehörigen *templates* bekannt sein. Das *template* kann daher frühestens im Schritt der Anwendung der Regel vollständig erzeugt werden. Dies bedeutet, dass die Regelanwendung nicht ausschließlich aus einer Anwendung eines XSLT-Stylesheet bestehen kann, obwohl ein großer Teil der notwendigen *templates* bereits bei der Produktionsinstantiierung erzeugt werden können.

Die Realisierung der Regelanwendung ist folglich auf Basis von XSLT möglich, allerdings sind Veränderungen von *inscriptions* und Fusionen aufwendig zu realisieren.

3.4.6 Besonderheiten der Implementation

Im Folgenden werden weitergehende Aspekte erläutert, die sich aus dem theoretischen (2.2.3 sowie 2.4.3) wie architektonischen Ansatz (3.4) ergeben und in der Implementierung betrachtet werden müssen.

Verwendung von IDs in XML Dokumenten

Die Definition der XML-Struktur von Petri Netzen ist durch die Verwendung von RelaxNG ohne weiteres möglich, wie in Kapitel 3.2.1 gezeigt. Bei einer Validierung eines PNML Dokuments wird diese Struktur überprüft, allerdings ohne die Korrektheit von IDs zu überprüfen. Daher werden Netze nicht als ungültig erkannt, wenn sie mehrere Elemente mit der gleichen ID enthalten. Auch Kanten, die IDs verwenden, für die es keine tatsächlichen Knoten gibt, werden bei einer Validierung nicht erkannt. Selbst eine einzelne Kante (ohne Knoten) wird als korrektes Netz validiert. Desweiteren kann eine Kante zwei Stellen, zwei Transitionen oder sogar zwei Kanten verbinden, ohne das eine Validierung fehlschlägt. Diese Probleme rühren daher, dass eine

Korrektheitsprüfung durch RelaxNG nur auf Basis der Syntax und nicht auf Basis der Semantik stattfindet. Diese ist durch RelaxNG nicht definierbar und damit auch nicht validierbar.

Das gleiche Problem ergibt sich bei der Produktionsdefinition in Kapitel 3.3. Zum Einen können die verwendeten Netze problematische IDs enthalten, wie z.B. IDs ohne referenzierten Knoten, da die Petri Netz-Definition auf PNML aufbaut. Zum Anderen können bei der Definition der verwendeten Abbildungen nicht vorhandene IDs oder Knoten unterschiedlichen Typs auf einander abgebildet werden.

Diese Probleme können durch zusätzliche Prüfungen nach der Validierung der Struktur der Produktion bzw. des PNML Dokuments gelöst werden. Jede dieser Prüfungen steigert jedoch die Laufzeit der Produktionsinstantiierung und Regelanwendung. Die Nutzung solcher Prüfungen ist daher nur zu empfehlen, wenn in der Anwendung vermehrt mit derartigen inkorrekten Produktionen bzw. Netzen zu rechnen ist. Im Fall der in Kapitel 3.4 beschriebenen Implementierung wurde daher auf die Prüfung der Korrektheit der IDs bei PNML Dokumenten verzichtet und dem Anwender die Verantwortung der Sicherstellung von semantischer Korrektheit von Netzen übertragen. Findet eine Regelanwendung auf einem semantisch inkorrekten Netz statt, so kann nicht erwartet werden, dass ein korrektes Netz resultiert. Für die Abbildungen innerhalb der Produktionen wird eine Prüfung der IDs im Standardfall durchgeführt. Diese Überprüfung kann jedoch aus den oben genannten Gründen der Performanz deaktiviert werden, um die benötigte Laufzeit zu verkürzen.

Flexibler Regelformalismus

Um die Implementierung möglichst anpassbar an unterschiedliche Netzformalismen zu gestalten, wurde ein Ansatz auf Basis von PNML gewählt. Zusätzlich wurden die Bedingungen der Abbildungen zwischen Netzen bewusst flexibel gestaltet. Eine Abbildung eines XML Knotens auf einen zweiten ist korrekt, wenn der zweite Knoten jeden Kindknoten des ersten Knotens enthält (siehe Definition 10 in Kapitel 3.4.1). Dies bewirkt zwar eine inhärente Unsicherheit, ermöglicht jedoch eine allgemeinere Produktionsdefinition. Besitzt ein Netz beispielsweise sowohl Knoten, die die Semantik beeinflussen (*inscriptions*), als auch Knoten, die die visuelle Darstellung des Netzes beschreiben, so können die beiden Typen von Knoten unabhängig voneinander transformiert werden. Bei der Transformation der *inscriptions* ist kein Wissen über die visuelle Darstellung von Nöten und diese wird nicht beeinflusst. Ohne diese Flexibilität müsste eine Produktion, welche *inscriptions* transformiert, um die Knoten der Darstellung erweitert werden, obwohl diese nicht verändert werden und eigentlich für die Produktion keine Relevanz haben, da für das Matching nur die verwendeten *inscriptions* zu beachten sind. Eine Veränderung der visuellen Darstellung sollte auch wiederum möglich sein, ohne die Semantik des Netzes zu beachten.

Inhärente Unsicherheit durch inscriptions

Wie auch bereits in Kapitel 2.4 beschrieben existiert eine inhärente Unsicherheit bei der Verwendung von *inscriptions* in Produktionen, wenn die Existenz von *inscriptions* nicht streng vorgeschrieben ist. Verdeutlicht sei dies durch die Produktion in Abbildung 3.10, die einer Stelle die *inscription* `int` zuordnet. Der in der Implementierung verwendete Regelformalismus lässt es zu, dass die Stelle des linken Netzes L auf eine Stelle des zu verändernden Netzes G abgebildet wird, die bereits eine *inscription* besitzt. Dies ist der Fall, da an die verwendeten Abbildungen die Bedingung gestellt wird, dass für jedes abgebildete Element e die *inscription* von e ein Teil der *inscription* des Bildes von e ist (Definition 10). Aus praktischen Gründen wird hier nicht

gefordert, dass die *inscriptions* von e und seinem Bild identisch sein müssen. Dies hat den Vorteil, dass die Implementierung zu beliebigen PNML Definitionen kompatibel ist. Die so erzeugte inhärente Unsicherheit kann verhindert werden, indem für das Matching m der linken Produktionsseite auf das zu verändernde Netz gefordert wird, dass jedes Element des Netzes und sein Bild bezüglich m die gleiche *inscription* haben.

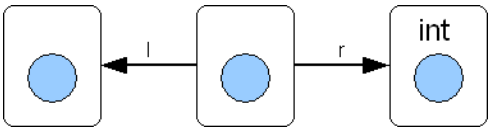


Abbildung 3.10: Beispiel für inhärente Unsicherheit einer Produktion

Eine zweite Quelle inhärenter Unsicherheit ist die Fusion von Elementen des Netzes, da die Fusion durch ein Hinzufügen von *inscriptions* realisiert ist. Daher kann auch diese durch die oben beschriebene Forderung an das Matching verhindert werden.

Explizite Definition von Kanten

Im Gegensatz zu dem Ansatz von Ehrig, wie er in Kapitel 2.3 vorgestellt wurde, sind bei der Implementierung durch die Verwendung von PNML Kanten explizit definiert. Dies erleichtert sowohl das Löschen und Hinzufügen von Kanten, als auch das Löschen, Hinzufügen und Verändern von *inscriptions*. Durch diese Definition ist es notwendig, Kanten explizit auch bei Abbildungen zwischen Netzen abzubilden.

Mögliche Erweiterungen der Implementierung

Die in Kapitel 3.4 vorgestellte Implementierung kann in ihrer vorliegenden Form an zwei Stellen erweitert werden.

PNML Definitionen, bei denen Elemente des Netzes mehrere gleichnamige Kindknoten besitzen können, werden nicht unterstützt. Dies liegt an der Schwierigkeit der Zuordnung dieser Kindknoten in Kombination mit Abbildungen zwischen Netzen. Sei dies verdeutlicht durch den Ausschnitt einer Produktion in Listing 3.15. Dafür sei angenommen, dass die verwendeten Netze nach einer nicht näher erläuterten PNML Definition korrekt sind.

Laut verwendetem Regelformalismus muss zu dem `inscription` Knoten der Transition des Interface Netzes ein `inscription` Knoten der Transition des rechten Netzes existieren, der alle Kindknoten des ursprünglichen `inscription` Knotens enthält. Dies ist jedoch der Fall für zwei `inscription` Knoten. Da es sich hier um die rechte Seite der Produktion handelt, müsste der Unterschied zwischen `inscription` Knoten des Interface Netzes und `inscription` Knoten des rechten Netzes in den `inscription` Knoten des zu verändernde Netz eingefügt werden. Je nach Wahl der `inscription` des rechten Netzes ist daher `<text>def</text>` bzw. `<text>ghi</text>` einzufügen. Der jeweils andere `inscription` Knoten muss als neuer Kindknoten der Transition in das zu transformierende Netz eingefügt werden. Da jedoch der `inscription` Knoten des tatsächlichen Netzes mehr Kindknoten enthalten kann als der Produktion bekannt ist, macht es einen Unterschied, welcher der `inscription` Knoten der rechten Seite gewählt wird. Da dies jedoch nicht in der gewählten Datenstruktur konfigurierbar ist, wäre eine Erweiterung der Datenstruktur nötig, um dies zu unterstützen.

Eine mögliche Lösung dieses Problems ist das Erweitern der im Regelformalismus verwendeten Abbildungen, so dass diese zusätzlich eine Zuordnung der `inscription` Knoten enthalten. Dazu müssen diese Knoten mit eindeutigen IDs versehen werden. Dies kann durch das Einfügen

eines `id` Attributs in die Definition geschehen. Alternativ kann auf den `toolspecific` Knoten zurückgegriffen werden, der in der PNML Basisdefinition bereits definiert ist und dazu dient, verschiedenen Programmen die Möglichkeit zu bieten, Informationen, die sie zur Verarbeitung des Netzes benötigen, direkt im PNML Dokument zu speichern. Unabhängig von der Umsetzung muss die Unterscheidung von gleichnamigen Knoten auf jeder Ebene der XML Baumstruktur möglich sein, da die Mehrdeutigkeit an jeder Stelle auftreten kann.

```
<interface>
  <net id="n1" type="refNet">
    <transition id="t1">
      <inscription>
        <text>abc</text>
      </inscription>
    </transition>
  </net>
</interface>
<insertNet>
  <net id="n1" type="refNet">
    <transition id="t1">
      <inscription>
        <text>abc</text>
        <text>def</text>
      </inscription>
      <inscription>
        <text>abc</text>
        <text>ghi</text>
      </inscription>
    </transition>
  </net>
</insertNet>
```

Listing 3.15: Ausschnitt einer problematischen Produktion

Ein ähnlich gelagertes Problem tritt bei der Fusion von Elementen des Netzes auf, wenn gleichzeitig Kindknoten der zu fusionierenden Knoten verändert werden sollen. Auch hier beruht das Problem darauf, dass sich im zu verändernden Netz mehr Knoten befinden, als der Produktion bekannt ist. Das Einfügen neuer *inscriptions* müsste parallel zur Fusion stattfinden. Durch eine nicht injektive Abbildung der linken Produktionsseite kann es jedoch vorkommen, dass weitere Knoten fusioniert werden müssen, bei denen in der Produktion nicht explizit angegeben ist, wie die Kindknoten zu behandeln sind. Da diese Konstruktion unübersichtlich und aufwendig aufzulösen ist, wird sie bei der Implementierung nicht berechnet. Sobald ein Element des Netzes der rechten Produktionsseite mehr als ein Urbild im Interface Netz besitzt, wird es im Schritt der Veränderung von *inscriptions* nicht beachtet.

Es wurde bereits versucht diese Problem in der Implementierung durch das Java-Interface `NodeMerger` zu lösen. Ein Anwender kann dabei durch eine Implementierung dieses Interfaces sicherstellen, dass die von ihm verwendeten PNML Dokumente immer korrekt fusioniert werden. Durch diese Lösung des Problems ist es jedoch nur möglich, die Fusion von Elementen des Netzes allgemein zu spezifizieren. Eine Fusion zweier konkreter Konten ist jedoch nicht innerhalb einer Produktion definierbar.

3.5 Automatische Generierung von Matchings

Sowohl bei den theoretischen Erläuterungen zu Graphtransformationen aus Kapitel 2, als auch bei der Implementierungen dieser in Kapitel 3.4 wird eine Matching verwendet, welches ein Netz

der Produktion auf das zu transformierende Netz abbildet. Bei der Regelanwendung wird dieses Matching, wie auch das zu transformierende Netz als gegeben vorausgesetzt. Eine sinnvolle Erweiterung der Implementierung wäre die Möglichkeit der automatischen Generierung eines Matchings. Daher soll die Umsetzbarkeit dieser Verbesserung im Folgenden untersucht werden.

Eindeutigkeit

Es kann vorkommen, dass es zu einem Paar von Produktion und zu transformierendem Netz kein Matching gibt. Dies ist beispielsweise der Fall, wenn die Produktion Transitionen enthält, das zu verändernde Netz jedoch nicht. Da das Matching eine totale Abbildung sein muss, müssen die Transitionen des linken Netzes auf Transitionen des zu transformierenden Netzes abgebildet werden. Da das zu transformierende Netz keine Transitionen besitzt, ist dies unmöglich.

Falls ein Matching existiert, so ist dies in der Regel nicht eindeutig. Bestehe die linke Seite der Produktion, welche das Matching abbildet, beispielsweise aus einer Transition, einer Stelle und einer von der Transition ausgehenden Kante. Es gibt sicherlich viele Matchings für diese Produktion, da die Transition der Produktion auf jede Transition mit einer ausgehenden Kante des zu transformierenden Netz abgebildet werden kann, unabhängig davon, wie viele Kanten die Transition insgesamt besitzt.

Diese Mehrdeutigkeit wird durch die Verwendung von nicht-injektiven Abbildungen verstärkt. Sei angenommen, dass die linke Seite einer Produktion aus zwei Stellen, zwei Transitionen sowie vier Kanten besteht, wie in Abbildung 3.11 links dargestellt. Das in Abbildung 3.11 dargestellte Matching bildet beide Transitionen, beide Stellen, beide in die Transitionen eingehenden Kanten und beide aus den Transitionen ausgehenden Kanten auf jeweils das gleiche Objekt ab. Das Bild des Matchings ist damit nicht isomorph zu der linken Produktionsseite. Da Isomorphismus an dieser Stelle nicht gefordert wird, erhöht sich die Anzahl der möglichen Matchings.

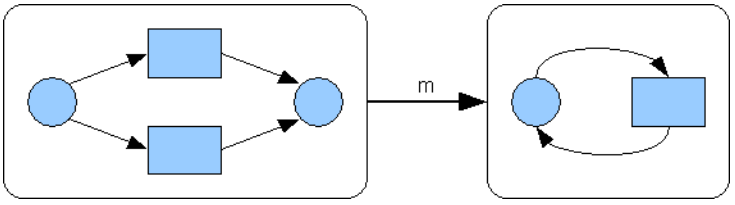


Abbildung 3.11: Ein mögliches nicht-injektives Matching

Automatische Erzeugung

Eine automatische Erzeugung eines Matchings kann im allgemeinen Fall problematisch sein. Dies hängt mit dem Graphisomorphie-Problem zusammen, also der Frage, ob die Knoten eines von zwei gegebenen Graphen so umsortiert werden können, dass der andere Graph entsteht. Dies ist eins der wenigen Probleme, von denen weder bekannt ist, ob es in polynomieller Zeit lösbar noch ob es NP-vollständig ist (siehe [18]). Es liegt allerdings in NP, da die Anzahl der möglichen Umsortierungen von Knoten endlich ist und bei einer gegebenen Umsortierung leicht zu prüfen ist, ob sie korrekt ist.

Ist als Matching nur eine injektive Abbildung erlaubt, so entspricht die Suche nach einem möglichen Matching dem Subgraphisomorphie-Problem. Das Subgraphisomorphie-Problem stellt die Frage, ob bei zwei gegebenen Graphen der erste Graph einen Subgraphen besitzt, der isomorph zum zweiten Graphen ist. Für dieses Problem konnte gezeigt werden, dass es NP-vollständig ist (siehe [10]). Ein allgemeiner, möglichst effizienter Algorithmus ist daher nicht

einfach zu implementieren. Es muss mit Heuristiken gearbeitet werden, um die Laufzeit zu verbessern. Dies kann beispielsweise geschehen durch die Unterscheidung von Knoten des Netzes anhand der Anzahl ihrer eingehenden und ausgehenden Kanten. Stimmen diese für zwei Knoten nicht überein, so macht es keinen Sinn, diese aufeinander abzubilden. Auch sollte die Bipartition von Petri Netzen ausgenutzt werden, indem bei der Erzeugung eines Matchings die Knoten frühestmöglich in Stellen und Transitionen aufgeteilt werden. Auch ein fehlender Zusammenhang eines Netzes kann ausgenutzt werden. Zur automatischen Erzeugung eines Matchings ist es jedoch auch notwendig festzulegen, welches Matching zu wählen ist, falls es mehrere Matchings gibt, die die gestellten Bedingungen erfüllen.

Eine automatische Erzeugung von nicht-injektiven Matchings macht nur bedingt Sinn, da sich diese Matchings auf die Wirkungsweise einer Produktion auswirken. Dies ist abhängig davon, wie das Problem der Mehrdeutigkeit des *pushout* Komplements, wie in Kapitel 2.2.3 erläutert, gelöst ist. Sei hier davon ausgegangen, dass bei Bildung des *pushout* Komplements keine Knoten geteilt werden und sei die Produktion in Abbildung 3.12 betrachtet. Bildet ein Matching nun beide Stellen der linken Produktionsseite auf die gleiche Stelle des zu verändernden Netzes ab, so hat eine Anwendung der Regel keinen Effekt. Dies ist der Fall, da die abgebildete Kante zwar gelöscht wird, aber die neu erzeugte Kante den gleichen Start- und Zielknoten besitzt, da die beiden Stellen der Produktion verschieden sind, nicht jedoch in dem zu verändernden Netz.

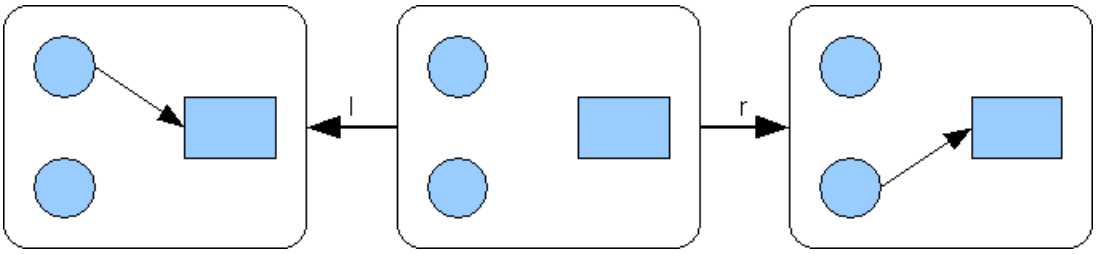


Abbildung 3.12: Eine Produktion, die eine Kante “vertauscht“

3.6 Anweisung der Benutzung

Da in Kapitel 3.4 vor allem der Aufbau und die Funktionsweise der Implementierung im Zentrum der Betrachtung stand, soll in diesem Kapitel erläutert werden, wie das implementierte System aus der Sicht eines Programmierers zu handhaben ist (3.6.1). An dieser Stelle ist es außerdem wichtig auf Schwierigkeiten aufmerksam zu machen, die sich sowohl aus der theoretischen Grundlage, als auch aus der gewählten Implementierung ergeben (3.6.2).

3.6.1 Ablauf einer Transformation

Instantiierung einer Produktion

Zur Durchführung einer Transformation wird eine instantiierte Produktion in Form eines `Rule` Objekts benötigt. Die Erzeugung diese Objekts geschieht durch den folgenden Prozess, bei dem optionale Schritte durch das Schlüsselwort **optional** gekennzeichnet sind:

1. Der Programmierer muss eine Produktion in Form eines XML Dokuments nach der Definition in Kapitel 3.3 erzeugen. Ob die Erzeugung des XML Dokuments manuell oder automatisch geschieht, ist dem Programmierer überlassen. Ein Beispiel einer solchen Produktion ist in Kapitel 3.3.2 in den Listings 3.10 bis 3.13 aufgeführt.

2. Die Erzeugung eines `Rule` Objekts erfolgt nach dem *builder pattern*. Die Klasse `Rule` ist daher nicht direkt instantiierbar, sondern muss durch die Verwendung der Klasse `RuleBuilder` erzeugt werden, welche die vollständige Instantiierung durchführt. Vor dem eigentlichen Schritt der Instantiierung muss einem `RuleBuilder` Objekt alle notwendigen Daten zugewiesen werden. Daher muss eine Instanz dieser Klasse in der folgenden Weise erzeugt werden:

   ```
   RuleBuilder builder = new RuleBuilder();
   ```

   Ein `RuleBuilder` Objekt ist nicht an eine Produktion gebunden, es kann daher zur Generierung beliebig vieler `Rule` Objekte genutzt werden.

3. Dem `RuleBuilder` Objekt muss eine Produktion in Form eines XML Dokuments (3.3) bekannt gemacht werden. Dies geschieht durch die Angabe einer URL, welche auf diese Produktion verweist, durch eine der zwei folgenden Zeilen:

   ```
   RuleBuilder builder = new RuleBuilder(ruleURL);
   builder.setRule(ruleURL);
   ```

   Die Verwendung des parametrisierten Konstruktors der `RuleBuilder` Klasse ist gleich bedeutend mit der Erzeugung eines `RuleBuilder` Objekts durch den parameterlosen Konstruktor und die anschließende Zuweisung der URL durch die `setRule(String)` Methode.

4. **optional** Durch Konfiguration des `RuleBuilder` Objekts kann auf die Erzeugung des `Rule` Objekts Einfluss genommen werden. Dazu können im Wesentlichen die folgenden Methodenaufrufe genutzt werden:

   ```
   builder.setRuleStructureCheck(false);
   builder.setRuleMappingCheck(false);
   builder.setRuleStructureURL(ruleStructureURL);
   builder.setNodeMerger(nodeMerger);
   ```

   Durch die Methode `setRuleStructureCheck(boolean)` ist konfigurierbar, ob bei der Instantiierung der Produktion die Struktur des XML Dokuments geprüft werden soll.

   Unabhängig davon kann durch die Methode `setRuleMappingCheck(boolean)` eingestellt werden, ob die Korrektheit der in der Produktion definierten Abbildungen bei der Instantiierung überprüft werden soll. Wird diese Prüfung deaktiviert, so muss an anderer Stelle sichergestellt werden, dass die in Abschnitt 3.4.1 beschriebenen Bedingungen an Abbildungen gelten.

   Verwendet der Anwender eine andere PNML Definition, so muss die daraus abgeleitete Produktionsdefinition dem `RuleBuilder` Objekt zugewiesen werden. Dies geschieht durch Aufruf der Methode `setRuleStructureURL(String)` unter Angabe einer URL, die auf die Produktionsdefinition verweist.

   Sollen alle von diesem `RuleBuilder` Objekt generierten `Rule` Objekte ein vom Anwender angepasstes `NodeMerger` Objekt (siehe 3.4.4) verwenden, so ist es sinnvoll, dieses dem `RuleBuilder` Objekt durch Aufruf der Methode `setNodeMerger(NodeMerger)` zuzuweisen. Alle generierten `Rule` Objekte verwenden automatisch das zugewiesene `NodeMerger` Objekt bei Fusionen und müssen daher nicht einzeln konfiguriert werden.

5. Ein `Rule` Objekt wird durch den Aufruf der `buildRule()` Methode instantiiert. Werden bei der Verarbeitung dieser Methode Fehler festgestellt, so wird eine `RuleBuildingException`

auftreten. Tritt eine `RuleBuildingException` auf, so kann anhand des Fehlertextes nachvollzogen werden, ob die Struktur der Produktion fehlerhaft ist oder das Problem an einer fehlerhaften Abbildungen liegt. Im letzteren Fall sei auf die zu erfüllenden Bedingungen in Kapitel 3.4.1 verwiesen. Eine detaillierte Liste der möglichen Fehlercodes kann Tabelle 3.1 entnommen werden.

Anwendung einer Regel

Wurde durch die Instantiierung der Produktion ein `Rule` Objekt generiert, so kann dieses angewandt werden, ohne das weitere Einstellungen erforderlich sind. Dies geschieht durch die `apply(String,Mapper)` Methode des `Rule` Objekts. Der erste Parameter muss eine URL in Form eines `String` Objekts sein, welche auf das zu transformierende PNML Dokument verweist. Der zweite Parameter muss ein `Mapper` Objekt sein, welches das linke Netz der Produktion auf das zu transformierende Netz abbildet. Dieses `Mapper` Objekt muss die Bedingungen aus Kapitel 3.4.1 erfüllen, was standardmäßig beim Aufruf der `apply(String,Mapper)` Methode überprüft wird. Soll dies nicht geschehen, so muss dies im Vorhinein durch die Methode `setMapperChecking(boolean)` konfiguriert werden. Im Vorhinein kann auch durch die Methode `setNodeMerger(NodeMerger)` das zu verwendende `NodeMerger` Objekt konfiguriert werden, falls dies nicht bereits bei der Instantiierung der Produktion geschehen ist.

Das Resultat der `apply(String,Mapper)` Methode ist ein `InputStream` Objekt, über welches das transformierte PNML Dokument auslesbar ist. Tritt bei der Regelanwendung ein Fehler auf, so wird eine `RuleApplicationException` geworfen. Die möglichen Fehlercodes dieser `Exception` können der Tabelle 3.1 entnommen werden.

Mögliche Fehlercodes

Tabelle 3.1 zeigt die möglichen Fehlercodes, welche bei der Produktionsinstantiierung oder der Regelanwendung auftreten können. Der Großteil dieser Fehlercodes bezieht sich auf die Korrektheitsprüfung der verwendeten Abbildungen. Im Falle der Produktionsinstantiierung sind dies die Abbildungen der linken und der rechten Seite, welche in der Produktion definiert sind. Bei der Regelanwendung können die Fehlercodes bei der Prüfung des beim Aufruf der `apply(String,Mapper)` Methode übergebenen `Mapper` Objekts auftreten.

3.6.2 Weitere Hinweise zu Produktionen

Inscriptions in Produktionen

Wie die Anwendung einer Regel durch das Angeben von *inscriptions* innerhalb der Produktion beeinflusst wird, ist in Kapitel 3.4.4 beschrieben. Es ist wichtig zu verstehen, dass die in der Produktion aufgeführten Kindknoten eines Elements des linken Netzes im zu transformierenden Netz vorhanden sind, dieses jedoch noch weitere Kindknoten enthalten kann. Die sich daraus ergebende inhärente Unsicherheit ist zu beachten, wie in Kapitel 3.4.6 dargelegt wurde.

Nicht-injektive linke Produktionsseite

Die Verwendung einer nicht-injektiven Abbildung in der linken Produktionsseite verursacht die in Kapitel 2.2.3 beschriebenen Probleme, da hierdurch die Eindeutigkeit des *pushout* Komplements nicht gegeben ist. Dies wird in der Implementierung gelöst, indem keine Knoten geteilt werden (3.4.4). An dieser Stelle ist jedoch die *identification condition* für *inscriptions* (2.4.3) zu beachten.

Fehlercode	Beschreibung
Fehlercodes der Korrektheitsprüfung von Mapper Objekten (Produktionsinstantiierung und Regelanwendung)	
`ID_NOT_IN_SOURCE_SPACE`	Das `Mapper` Objekt bildet nicht jede ID des Ursprungsnetzes ab.
`ID_TYPE_MISMATCH`	Der XML Knoten einer ID ist nicht von dem gleichen Typ wie der XML Knoten des Bildes der ID.
`ID_NOT_IN_TARGET_SPACE`	Eine durch das `Mapper` Objekt abgebildete ID besitzt keinen XML Knoten, auf den sie verweist.
`NODE_NOT_FOUND_IN_NET`	Für eine ID, für welche das `Mapper` Objekt ein Bild angibt, existiert kein Knoten, auf den sie verweist.
`TAR_OR_SRC_ID_NOT_IN_SOURCE_SPACE`	Eine der als Endpunkte einer Kante angegebenen IDs wird nicht durch das `Mapper` Objekt abgebildet.
`MAPPING_DESTROYS_STRUCTURE`	Die Bilder der Endpunkte einer Kante stimmen nicht mit den Endpunkten des Bildes dieser Kante überein.
`SOURCE_NOT_PART_OF_TARGET_NODE`	Ein Element des Bildnetzes besitzt nicht alle Kindknoten seiner Urbilder (siehe Definition 10 in Kapitel 3.4.1).
`DELETE_MAPPER_NOT_SAFE`	Die Kindknoten aller Elemente des Interface Netzes I, welche auf das gleiche Element des zu transformierenden Netzes abgebildet werden, sind identisch (*identification condition* für *inscriptions*, siehe 2.4.3).
Spezielle Fehlercodes der Produktionsinstantiierung	
`NO_RULE_GIVEN`	Die `build()` Methode wurde aufgerufen, ohne vorher eine Produktion in Form eines XML Dokuments anzugeben.
`RULE_STRUCTURE_NOT_VALID`	Das XML Dokument der Produktion ist syntaktisch nicht korrekt.
Spezielle Fehlercodes der Regelanwendung	
`GLUING_ILL_DANGLING`	Die *dangling condition* der *gluing condition* ist nicht erfüllt (2.2.3). Es sollen Knoten des zu transformierenden Netzes gelöscht werden, nicht jedoch all ihre Kanten.
`GLUING_ILL_IDENTIFICATION`	Die *identification condition* der *gluing condition* ist nicht erfüllt (2.2.3). Die Regel gibt an, dass Elemente des zu transformierenden Netzes gleichzeitig gelöscht und erhalten bleiben sollen.

Tabelle 3.1: Fehlercodes der Produktionsinstantiierung und Regelanwendung

Für jedes Element des linken Netzes müssen daher alle seine Urbilder im Interface Netz bis auf die IDs identisch sein.

Das Problem der Eindeutigkeit des *pushout* Komplements tritt auch bei einer injektiven linken Produktionsseite auf, falls das bei der Regelanwendung verwendete Matching nicht injektiv ist.

Nicht-injektive linke Produktionsseiten werden im Regelfall nicht benötigt, können jedoch in Kombination mit nicht-injektiven rechten Produktionsseiten hilfreich sein, wie im Folgenden Abschnitt erläutert.

Nicht-injektive rechte Produktionsseite

Eine nicht-injektive Abbildung des Interface Netzes auf das rechte Netz bewirkt eine Fusion von Elementen des Netzes. Dabei werden diejenigen Elemente des zu transformierenden Netzes fusioniert, dessen Urbilder im Interface Netz gleiche Bilder im rechten Netz besitzen. Dazu ist zu beachten, dass das Ergebnis einer Fusion alle Kindknoten beider zu fusionierenden Elemente enthält und daher unter Umständen kein korrektes Element des Netzes nach der verwendeten PNML Definition ist. Um dies sicherzustellen, muss eine eigene Implementation des `NodeMerger` Java-Interfaces stattfinden, welche die Struktur der verwendeten PNML Definition bei der Fusion einbezieht.

Bei der Regelanwendung werden gegebenenfalls durch die rechte Abbildung der Produktion Kindknoten in Elemente des zu transformierenden Netzes eingefügt. Dazu wird für jedes Element des Interface Netzes I der Unterschied zwischen den Kindknoten dieses Elements und den Kindknoten seines Bildes im rechten Netz R bestimmt. Existiert ein Unterschied, so wird dieser in das Bild des Elements des Interface Netzes I im zu transformierenden Netz G eingefügt. Aus Gründen der Implementierung (siehe 3.4.6) wird ein Unterschied zwischen einem Element des Interface Netzes und seinem Bild im rechten Netz nicht beachtet, wenn es noch mindestens ein weiteres Element des Interface Netzes gibt, welches ebenfalls dieses Bild besitzt. Um in einer einzigen Produktion die *inscription* eines Elements zu verändern und dieses Element mit einem weiteren zu fusionieren, muss die Abbildung der linken Produktionsseite nicht-injektiven sein. Die in der Produktion verwendeten Abbildungen müssen dazu grob wie in Abbildung 3.13 definiert sein. Bei der Regelanwendung der dort angegebenen Produktion findet die Veränderung der Kindknoten für den obersten Knoten des angegebenen Interface Netzes normal statt, da kein weiterer Knoten auf sein Bild im rechten Netz abgebildet wird. Danach findet die Fusion von Elementen des Netzes statt, bei der der mittlere Knoten des Interface Netzes mit dem untersten Knoten fusioniert wird. Durch die Abbildung der linken Produktionsseite haben der obere und der mittlere Konten des Interface Netze dasselbe Bild im linken Netz L und damit auch dasselbe Bild im zu verändernden Netz G. Daher kann so erreicht werden, dass bei der Regelanwendung die *inscription* des Knotens des zu verändernden Netzes verändert und dieser Knoten mit einem anderen fusioniert wird. Hierbei ist zu beachten, dass die Veränderung von *inscriptions* bei der Regelanwendung vor der Fusion von Knoten und Kanten stattfindet, wie im Algorithmus in Kapitel 3.4.4 erläutert.

3.7 Zusammenfassung

Im vorhergehenden Kapitel wurde eine Implementierung eines Graphtransformationssystems für Petri Netze auf Basis des in Kapitel 2.2.3 vorgestellten DPO Ansatzes unter Verwendung von XML vorgestellt. Dabei wurde Wert gelegt auf Möglichkeiten der Anpassung der Implementierung an unterschiedliche Formalismen für Petri Netze. Realisiert wurde dies zum Einen durch eine generische Produktionsdefinition (3.3) und zum Zweiten durch die Definition der **Teil von**

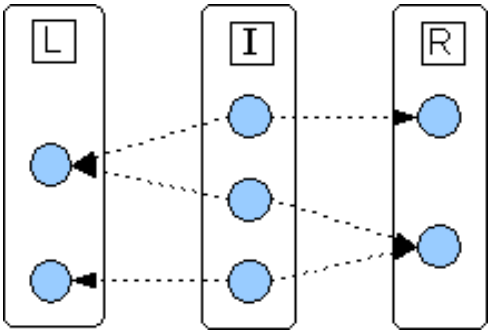


Abbildung 3.13: Beispiel einer Produktion mit zwei nicht-injektiven Seiten

Relation für XML Knoten (Definition 10). Die mögliche Mehrdeutigkeit bei der Bestimmung des *pushout* Komplements (2.2.3) wurde durch Einführung der Konvention umgangen, dass Knoten nicht geteilt werden (3.4.4). Daraus ergibt sich, dass DPO Diagramme existieren, die die *gluing condition* erfüllen und daher gültig sind, jedoch nicht bei der Nutzung der Implementierung entstehen können. Jedes DPO Diagramm, welches nur injektive Abbildungen verwendet, kann allerdings mit der Implementierung erzeugt werden. Da die Implementierung nicht-injektive rechte Abbildungen und damit Fusionen von Elementen des Netzes unterstützt, können echt mehr DPO Diagramme erzeugt werden, als die Menge der injektiven DPO Diagramme beinhaltet. Die sich daraus ergebende Inklusionshierarchie ist in Abbildung 3.14 dargestellt.

Abbildung 3.14: Einordnung der realisierbaren DPO Diagramme

Ist die Abbildung der linken Produktionsseite und das Matching injektiv, so ist das DPO Diagramm eindeutig bestimmt. Die in der Implementierung eingeführte Konvention hat damit keine Auswirkungen auf die Verarbeitung der Regel. Ist die linke Produktionsseite oder das Matching nicht injektiv, so erzeugt die Implementierung nur ein DPO Diagramm. Um dies in zukünftigen Arbeiten zu erweitern, ist es notwendig ein Modul zu implementieren, das die Abarbeitung der linken Produktionsseite direkt in der Produktion vornimmt. Ähnlich wie bei der Fusion von Elementen muss dabei genau angegeben werden, wie die aufgeteilten Knoten zu bilden sind. Auf diese Weise wird eine Invertierung einer Produktion ermöglicht, auch wenn diese eine nicht-injektive Abbildung auf der rechten Seite verwendet. Bei einer solchen Lösung wird zum Einen die Definition von Produktionen komplexer und zum Anderen sind der Regel meist nicht alle Kindknoten eines Elements des Netzes bekannt, weil die **Teil von** Relation verwendet wird. Die Spaltung eines Elements wird durch diese inhärente Unsicherheit stark erschwert.

4 Interaktionsmodellierung

Die Interaktion zwischen Mensch und Maschine stellt ein Teilgebiet in der forschenden Informatik und der Psychologie dar. Ziel ist es, die Interaktion zu verbessern, um dem Benutzer eine angenehmere und effizientere Arbeit mit einer Maschine zu ermöglichen. Zunächst wird ein Modell der Interaktion auf psychologischer Ebene betrachtet, auf dessen Basis näher auf die zu lösenden Fragestellungen in der Interaktion eingegangen werden kann. Hierzu gilt es, die beteiligten Parteien der Interaktion genauer zu charakterisieren.

Die Motivation des Benutzers, eine Maschine zu verwenden, ist, durch sie effizienter Aufgaben lösen zu können. Die Interaktion zwischen Mensch und Maschine kann dabei als Kreislauf dargestellt werden, bei dem der Benutzer abwechselnd Anweisungen erzeugt, diese an die Maschine sendet und die Rückmeldung der Maschine interpretiert, um wiederum neue Anweisungen zu formulieren. Norman [15] bezeichnet diesen Kreislauf als *execution-evaluation circle* (siehe Abbildung 4.1). Er teilt die Interaktion dabei in die *execution* und die *evaluation* Phase auf. In der *execution* Phase muss der Benutzer sein Aufgabenmodell auf sein mentales Modell der Maschine (Prozessmodell) abbilden. Dabei charakterisiert das Aufgabenmodell eine Aufteilung der Ziele des Benutzers, welche er zu erreichen versucht. Das Aufgabenmodell ist benutzerspezifisch, aber weitestgehend unabhängig von der verwendeten Maschine. Zur Modellierung der mentalen Prozesse des Menschen wurden verschiedene Modelle, wie GOMS, CTT oder *keystroke-level modell* in diesem Zusammenhang entwickelt [6]. Die Funktionsweise der Maschine wird durch das sogenannte Prozessmodell abstrakt beschrieben. Die genauen internen Verarbeitungen und Berechnungen, die die Maschine durchführt, sind für die Interaktion nebensächlich, weshalb die Maschine hier als Blackbox betrachtet wird. Diese Betrachtung spiegelt die Sichtweise des Benutzers wieder, der in der Regel kein Wissen über den internen Aufbau der Maschine besitzt. Aus der Sicht des Benutzers nimmt die Maschine Eingaben entgegen und produziert Ausgaben, die er interpretieren muss. Es ist hier zusätzlich zwischen dem Prozessmodell und dem mentalen Prozessmodell eines Benutzers zu unterscheiden. Das mentale Prozessmodell eines Benutzers beschreibt das Verständnis eines Benutzers von einer Maschine und ist daher nicht zwangläufig mit dem tatsächlichen Prozessmodell der Maschine identisch. Dabei gilt es zu erwähnen, dass das mentale Prozessmodell maßgeblich durch die Benutzerschnittstelle und deren Interpretation durch den Benutzer geprägt wird. Dies kann unter anderem zu Problemen führen.

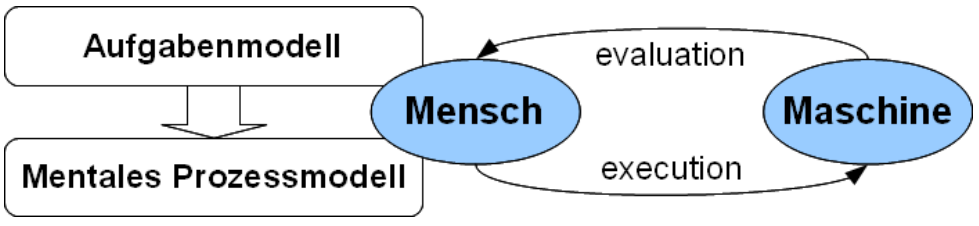


Abbildung 4.1: Der *execution-evaluation circle* nach Norman

Bei der Abbildung seines Aufgabenmodells auf sein mentales Prozessmodell wandelt der Benutzer seine geplanten Teilaufgaben in Anweisungen um, die die Maschine verstehen kann. Möchte ein Benutzer beispielsweise einen Text verfassen, so wird er auf der Ebene seines Auf-

gabenmodells zuerst den Schritt “Textverarbeitungsprogramm starten“ planen. Die Abbildung dieser Aufgabe auf sein mentales Prozessmodell resultiert, abhängig von der verwendeten Maschine, beispielsweise in einem Doppelklick auf das Icon des Textverarbeitungsprogramms. Der Computer nimmt den Startbefehl entgegen, startet das Programm und stellt es auf dem Bildschirm dar. Dies geschieht natürlich nur, wenn kein Fehler aufgetreten ist. Fehler können bei der Interaktion sowohl in der *execution* als auch in der *evaluation* Phase auftreten, die später beschrieben wird.

Startet das Programm nicht, wie es vom Benutzer beabsichtigt war, kann das verschiedene Ursachen haben. Zum Einem kann es sein, dass der Benutzer daneben geklickt hat und deshalb das Programm nicht startet. Zum Anderen kann es jedoch auch sein, dass ein anderes als das gewünschte Programm startet, wenn zwei Programme gleiche Symbole verwenden. Diese Fehler treten meist auf, wenn die Abbildung des Aufgabenmodells des Benutzers auf sein mentales Prozessmodell fehlerhaft ist oder wenn sich sein mentales Prozessmodell vom tatsächlichen Prozessmodell unterscheidet. Norman bezeichnet die Differenz zwischen den verschiedenen Modellen, die diese Fehler verursachen kann, als *gulfs of execution*. Eine fehlerhafte Abbildung des Aufgabenmodells auf das mentale Prozessmodell zu verhindern, ist ein Ziel bei dem Design von Benutzerschnittstellen. Im Falle des fehlgeschlagenen Ausführens des Textverarbeitungsprogramms kann die Schnittstelle beispielsweise durch Veränderung der Symbole verbessert werden, wenn zu ähnliche Symbole Ursache der Verwechslung sind.

Der zweite fehleranfällige Teil der Interaktion zwischen Mensch und Maschine ist die *evaluation* Phase. Der Benutzer erkennt, dass sein Befehl zum Öffnen des Textverarbeitungsprogramms korrekt erfasst wurde, wenn das zu startende Programm auf dem Bildschirm erscheint. In diesem Fall wird er mit der Planung seiner nächsten Aufgabe “Text schreiben“ in seinem Aufgabenmodell fortfahren. Erscheint nichts auf dem Bildschirm, so wird der Benutzer davon ausgehen, dass auf Seiten der Maschine irgendein Fehler aufgetreten ist, und die Ursache suchen oder erneut das Programm starten. Angenommen, der Computer ist zur Zeit des Programmaufrufs ausgelastet, so wird er nicht sofort das Programm starten können. Das der Programmstart im Hintergrund ausgeführt wird, ist jedoch für den Benutzer nicht ersichtlich. Dauert der Start zu lange, wird der Benutzer beispielsweise versuchen, das Programm erneut zu starten, was zu einem mehrfachen Ausführen führt, das der Benutzer nicht beabsichtigt hat. Auch dieses Problem ist auf eine Abweichung des mentalen Prozessmodells des Benutzers und dem tatsächlichen Prozessmodells zurückzuführen, was Norman als *gulfs of evaluation* bezeichnet. Der Benutzer muss daher in der *evaluation* Phase die Rückgaben der Maschine korrekt interpretieren können. Aus diesem Grund sollte diese Abweichung durch ein besseres Design der Benutzerschnittstelle verhindert werden, was durch Anpassung dieser an das mentale Prozessmodell erfolgen kann.

Den *gulf of execution* sowie den *gulf of evaluation* gilt es in diesem Kapitel durch Verwendung von adaptiven Benutzerschnittstellen auf Basis formaler Interaktionsmodellierung (4.1) entgegen zu wirken. Durch adaptive Benutzerschnittstellen kann eine bessere Anpassung der Schnittstelle an den Benutzer erfolgen und somit das Risiko von *gulfs of execution* oder *gulfs of evaluation* verringert werden. Dazu wird eine Software vorgestellt, die zur Realisierung dieser Schnittstellen verwendet werden kann (4.2) und die Verwendung dieser Schnittstellen anhand einer vereinfachten Bedienoberfläche eines Siedewasserreaktors demonstriert (4.3). Für eine umfangreiche Betrachtung der Mensch-Maschine Interaktion unter Einbeziehung weiterer Faktoren wie Wahrnehmung des Menschen, sowie Fehler im Umgang mit komplexen Prozessen, sei auf [6] und [11] verwiesen.

4.1 Interaktionslogik

Im vorigen Abschnitt wurde der *execution-evaluation circle* nach Norman [15] beschrieben und der *gulf of execution* sowie der *gulf of evaluation* vorgestellt, denen es entgegen zu wirken gilt. In der Regel wird diesem Problem durch zusätzlichen Zeit- und Geldaufwand bei dem Design der Benutzerschnittstelle begegnet. Dabei ist es vor allem wichtig, dass im Designvorgang Nutzer eingebunden werden und ausgiebige Tests der Benutzerschnittstellen erfolgen.

Auch bei einem gut gewählten Design einer Benutzerschnittstelle können die *gulfs of execution* und *gulfs of evaluation* nicht vollständig verhindert werden. Dies liegt an den Unterschieden der individuellen Aufgabenmodelle von verschiedenen Nutzern zur Lösung derselben Aufgabe. Eine Schnittstelle, die optimal für einen Benutzer ist, muss es nicht für alle sein. Bei einem hinreichend umfangreichen Programm können auf Grund der Masse die Funktionen nicht alle in gleicher Weise zugänglich sein. Beim Design der Benutzerschnittstelle muss daher entschieden werden, welche Funktionen am meisten genutzt werden, so dass auf diese schnell zugegriffen werden kann. Unterschiedliche Nutzer können mit ein und demselben Programm jedoch unterschiedlich arbeiten und damit Funktionen unterschiedlich stark nutzen.

Klassisch ist auch eine differenzierte Handhabung eines Programms durch erfahrene und unerfahrene Nutzer. Videobearbeitungssoftware bietet in der Regel viele Einstellmöglichkeiten (z.B. Bitraten für Audio und Video), die die Qualität des resultierenden Videos beeinflussen. Erfahrene Benutzer können durch entsprechende Einstellungen genau die gewollte Qualität erreichen, benötigen dazu jedoch Erfahrung oder Wissen über die verwendeten Codierungsverfahren. Ein unerfahrener Benutzer will und kann in der Regel jedoch nicht all diese Einstellmöglichkeiten nutzen. Eine vereinfachte Benutzerschnittstelle ist für den Anfänger geeignet, nicht jedoch für den erfahrenen Benutzer, da dieser bei einer zu einfachen Benutzerschnittstelle nicht die notwendigen Einstellungen vornehmen kann, um sein gewünschtes Ergebnis zu erreichen. Unter Umständen kann eine angepasste Schnittstelle auch unerfahrenen Benutzern komplexe Funktionen zugänglich machen. Die optimale Lösung dieses Konflikts ist die Verwendung von adaptiven, d.h. dem Nutzer anpassbare, Benutzerschnittstellen. Erfahrene wie unerfahrene Nutzer können durch eine Anpassung der Schnittstelle diese optimal für ihre Zwecke einrichten. Die Anpassung kann dabei direkt durch den Nutzer oder automatisch durch das Programm erfolgen.

Um diese Anpassbarkeit zu realisieren, wird der in [19] vorgestellte Ansatz zur Beschreibung formaler Interaktionslogik aufgegriffen. Die Interaktionslogik beschreibt dabei das Verhalten der Benutzerschnittstelle und den Informationsfluss zwischen physischer Repräsentation und der Maschine, auch bezeichnet als Aktionslogik. Hierdurch wird es möglich, die physische Repräsentation der Benutzerschnittstelle unabhängig von ihrer Logik und der eigentlichen Funktionalität der Aktionslogik anzupassen. Darüber hinaus ist es möglich, durch die Interaktionslogik Operationen zu modellieren, die nicht atomar, sondern in klar definierter Form aus mehreren Operationen der Aktionslogik zusammengesetzt sind. Hierdurch werden Teilprozesse der Aktionslogik in der Interaktionslogik modelliert und durch den Benutzer ausführbar. Um dies zu verdeutlichen, wird im Folgenden näher auf die in Abbildung 4.2 dargestellten Bestandteile eingegangen.

Aktionslogik Die Aktionslogik wird in diesem Modell als Blackbox angenommen. Sie besitzt dabei eine feste Anzahl statischer, unveränderbarer Eingangs- und Ausgangsports, über die Teile ihres Zustands entweder beeinflussbar oder beobachtbar sind. Aufgrund dieser Art der Verarbeitung innerhalb der Aktionslogik kann nur durch diese Ports Einfluss genommen bzw. beobachtet werden, wie der Systemzustand verändert wird.

Physische Repräsentation Die physische Repräsentation stellt den Teil der Benutzerschnittstelle dar, mit dem der Benutzer direkt interagiert. Über diese gibt der Nutzer Befehle an das

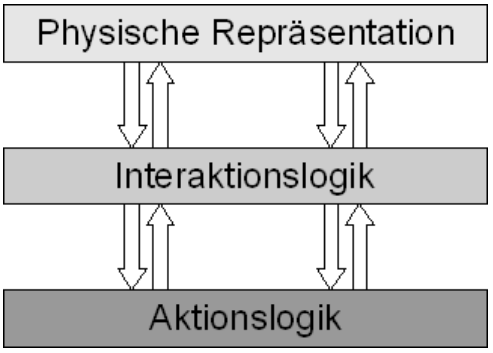


Abbildung 4.2: Position der Interaktionslogik innerhalb einer Benutzerschnittstelle

System und erhält Rückgaben von diesem.

Wie auch die Aktionslogik besitzt die physische Repräsentation der Schnittstelle Eingangs- und Ausgangsports, die die Kommunikation zwischen ihr und der Interaktionslogik ermöglicht. In dem beschriebenen Modell ist eine direkte Verbindung zwischen der physischen Repräsentation und der Aktionslogik nicht vorgesehen, da diese explizit formal durch die Interaktionslogik beschrieben wird. Desweiteren ist die physische Repräsentation nicht statisch. Sowohl ihr Aufbau als auch die Menge der Eingangs- und Ausgangsports ist veränderbar. Eine Veränderung der physischen Repräsentation wird als Redesign der Schnittstelle bezeichnet und ist unabhängig von der Veränderung der Interaktionslogik durchführbar. Es gibt jedoch auch Redesignoperationen, die Veränderungen der Interaktionslogik nach sich ziehen. Dies ist beispielsweise bei der Fusion von Knöpfen der Fall.

Interaktionslogik Die Interaktionslogik verbindet die physische Repräsentation mit der Aktionslogik. Da sie zur Abbildung der veränderbaren, physischen Repräsentation auf die statische Aktionslogik dient, muss die Interaktionslogik ebenfalls veränderbar sein, um die Möglichkeiten des Redesigns der Schnittstelle nicht unnötig einzuschränken. Die Veränderung der Interaktionslogik wird als Rekonfiguration der Schnittstelle bezeichnet. Die Rekonfiguration kann auch unabhängig von Veränderungen der physischen Repräsentation erfolgen, wobei manche Rekonfigurationen Redesignoperationen nach sich ziehen.

Im einfachsten Fall ist die Interaktionslogik nur eine Weiterleitung der Eingaben des Benutzers durch die physische Repräsentation an die Aktionslogik und der Rückgaben der Aktionslogik an die physische Repräsentation. Es ist allerdings durch die Interaktionslogik auch möglich, die Ein- und Ausgabedaten zu modifizieren und dadurch zusätzliche Logik zu realisieren. Dies sei anhand einer Heizung verdeutlicht, die auf drei Temperaturstufen “kalt“, “warm“ und “heiß“ einstellbar ist. Die Aktionslogik dieser Heizung besitzt dabei einen Eingangsport, der diese drei Temperaturstufen entgegen nimmt. Die physische Repräsentation der Schnittstelle ist unabhängig von der Aktionslogik und sei beispielhaft ein Schieberegler. Wird dieser Regler verschoben, so wird der aktuelle Wert des Reglers als eine Prozentangabe von 0 bis 100 % an die Interaktionslogik weitergegeben. Diese muss nun die Prozentangabe in eine der drei Temperaturstufen umwandeln und an die Aktionslogik weiterleiten. Diese Umwandlung findet dabei in der Interaktionslogik und nicht in der Aktionslogik statt und ist damit veränderbar. Durch eine solche Verwendung der Interaktionslogik ist die physikalische Repräsentation nicht mehr durch die Aktionslogik eingeschränkt. Eine Adaption der physischen Repräsentation kann daher einfach vorgenommen werden.

4.1.1 Formalisierung

Bei der Vorstellung der Interaktionslogik wurde in diesem Kapitel bewusst keine Modellierungsform festgelegt, da die Modellierung der Interaktionslogik zunächst nicht an einen Formalismus gebunden sein soll. Eine Modellierungsform der Interaktionslogik muss jedoch bestimmte Voraussetzungen erfüllen, damit die Eigenschaften der Interaktionslogik korrekt abgebildet werden können. Die wichtigste Eigenschaft ist dabei die Möglichkeit der Verarbeitung von Daten und die damit verbundene Möglichkeit zusätzliche Programmlogik zu definieren. Eine Modellierung muss jedoch auch gute Schnittstellen zur Anbindung der physischen Repräsentation und der Aktionslogik bieten.

Ein Formalismus, der diese Eigenschaften erfüllt und durch seine umfangreiche Erforschung bereits eine große theoretische Basis liefert, ist der Formalismus der farbigen Petri Netze. Der Vorteil farbiger Petri Netze ist sowohl die Möglichkeit, typisierte Eingabewerte der physischen Repräsentation und der Aktionslogik abbilden zu können als auch die zusätzliche Funktionalität der Interaktionslogik spezifizieren zu können. Ein weiterer Vorteil der Verwendung von Petri Netzen liegt in den Möglichkeiten der Analyse dieser Netze. Diese können zur formellen Validierung der verwendeten Interaktionslogik genutzt werden. Für eine formale Definition farbiger Petri Netze sei auf [12] verwiesen. Bei Verwendung einer objektorientierten Programmiersprache sind Referenznetze [13] besonders geeignet, da sie Objektorientierung unterstützen. Zusätzlich unterstützen sie sogenannte synchrone Kanäle, die eine Kommunikation der Interaktionslogik in Form eines Referenznetzes mit der physischen Repräsentation und der Aktionslogik ermöglichen.

Realisierung der Ports

Die Schnittstelle der Interaktionslogik in Form eines Referenznetzes zu der physischen Repräsentation und der Aktionslogik ist durch Stellen modelliert, wie in Abbildung 4.3 dargestellt. Eingaben des Benutzers und Rückgaben der Aktionslogik erfolgen durch das Einfügen von Tokens in Stellen des Netzes der Interaktionslogik. Diese so genannten Eingabestellen entsprechen den Ausgabeports der physischen Repräsentation und der Aktionslogik. In Abbildung 4.3 sind die Eingabe- und Ausgabestellen durch eine entsprechende Schraffur markiert. Tokens, welche in die Eingabestellen platziert werden, werden durch die Interaktionslogik verarbeitet und führen zur Generierung neuer Tokens, die in die Ausgabestellen des Netzes platziert werden. Ausgabestellen des Netzes entsprechen den Eingangsports der physischen Repräsentation und der Aktionslogik. Tokens, welche sich in Ausgabestellen befinden, werden an die Aktionslogik bzw. physische Repräsentation direkt weitergeleitet. Alle Stellen und Transitionen des Netzes der Interaktionslogik, welche keine Eingabe- oder Ausgabestellen sind, bilden die Funktionalität der Interaktionslogik.

Da die physische Repräsentation und die Aktionslogik in der Regel keine Petri Netze sind, kann das Netz der Interaktionslogik nicht isoliert betrachtet und simuliert werden, sondern es muss eine Möglichkeit geben, die Belegungen von Eingabe- und Ausgabestellen zu verändern. Die physische Repräsentation und die Aktionslogik müssen eine Möglichkeit besitzen, Tokens direkt in Eingabestellen zu platzieren. Zusätzlich müssen sie benachrichtigt werden, wenn Tokens in Ausgabestellen platziert werden, damit diese entnommen und verarbeitet werden können. Die verwendete Software zur Simulation des Netzes muss dafür entsprechende Möglichkeiten zur Verfügung stellen.

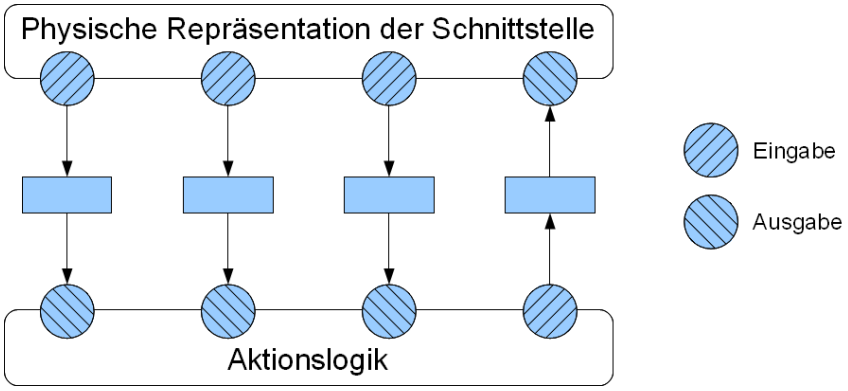


Abbildung 4.3: Anbindung der Interaktionslogik durch Stellen

Simulation von Interaktionslogik

Sobald der Benutzer Eingaben tätigt oder die Aktionslogik Rückgaben liefert, werden diese in die Eingabestellen platziert. Diese Eingaben müssen von dem Netz ohne weitere Einwirkungen von Außen nebenläufig verarbeitet werden. Dazu müssen die Tokens durch Feuern von Transitionen konsumiert und generiert werden, um den Informationsfluss innerhalb der Interaktionslogik zu realisieren. Dies muss durch die Simulation des Petri Netzes erreicht werden. Die Simulation muss automatisiert im Hintergrund stattfinden, so dass der Benutzer oder die Aktionslogik weitere Tokens in die Eingabestellen platzieren können, während vorherige Eingaben noch in der Bearbeitung sind. Angenommen, die Interaktionslogik leitet nur Tokens der Eingabestellen an Ausgabestellen weiter, so müssen für diese Weiterleitungen die entsprechenden Transitionen des Netzes feuern. Ein Simulator muss daher in regelmäßigen Abständen die Transitionen des Netzes bestimmen, die feuern können und diese Transitionen ausführen. Die Abstände müssen so klein gewählt sein, dass der Nutzer nicht bemerkt, dass seine Eingabe durch ein Petri Netz geleitet wird. Der Simulator muss zusätzlich die nötigen Einflussmöglichkeiten auf Eingabe- und Ausgabestellen während der Simulation berücksichtigen, welche im vorigen Abschnitt vorgestellt wurden.

Rekonfiguration

Eine Rekonfiguration einer als Petri Netz modellierten Interaktionslogik entspricht einer Transformation dieses Petri Netzes. Um Rekonfiguration formal zu realisieren, kann das Petri Netz bei jeder Veränderung von Grund auf neu erzeugt oder nur partiell verändert werden. Die erste Möglichkeit ist allerdings kostspielig und daher bei kleinen Änderungen nicht sinnvoll. Durch den Einsatz eines Systems zur Graphtransformation wird dieses Problem umgangen. Die Rekonfiguration kann dadurch auf die Definition und Anwendung einer Produktion zurückgeführt werden. Die theoretischen Grundlagen eines solchen Systems sind in Kapitel 2 erläutert, während eine konkrete Implementierung in Kapitel 3 vorgestellt wurde. Die konkrete Implementierung muss nun mit einer Simulationssoftware kombiniert werden, um eine flexible, ausführbare Interaktionslogik zu bilden.

4.2 Formalismus und Einbettung

Um die als Petri Netz dargestellte Interaktionslogik erfolgreich in eine Benutzerschnittstelle zu integrieren, muss eine geeignete Möglichkeit gefunden werden, Petri Netze zu modellieren

und zu simulieren, sowie in eine weitergehende Struktur einzubinden. Dazu wird im Folgenden Renew[1] verwendet, eine Software der Universität Hamburg. Diese bietet einen Editor zur Modellierung von Stellen/Transition Netzen und Referenznetzen (4.2.1) an, einer besonderen Form von farbigen Petri Netzen. Zusätzlich können diese Netze mittels Renew simuliert werden. Durch Einbindung von Renew als Bibliothek kann dies auch unabhängig vom visuellen Editor geschehen. Renew ist vollständig in Java implementiert und kann daher leicht in Kombination mit der Implementierung des Graphtransformationssystems aus Kapitel 3 genutzt werden. Zur Erläuterung des grundsätzlichen Aufbaus von Renew existiert ein User-Guide [14], welcher immer in der aktuellsten Version auf der Internetseite von Renew zu finden ist. Im Folgenden werden die Eigenschaften und die Handhabung von Referenznetzen durch Renew vorgestellt, um eine ausreichende Grundlage für die Implementierung von Interaktionslogik zu schaffen.

4.2.1 Referenznetze

Referenznetze sind farbige Petri Netze, die in *inscriptions* objektorientierte Konzepte nutzen. In einem Referenznetz können nicht nur einfache Datentypen, sondern auch Referenzen auf Objekte verwendet werden. Es ist dabei möglich, Referenzen auf Objekte und sogar auf Netze als Tokens im Netz zu verwenden oder im Netz neue Objekte zu erzeugen. Desweiteren ist es möglich, Programmcode der verwendeten Programmiersprache direkt in eine *inscription* einer Transition einzufügen, sodass dieser Programmcode bei der Ausführung der Transition ausgeführt wird. Auf diese Weise kann die Vorverarbeitung von Eingabedaten durch die Interaktionslogik realisiert werden.

In seiner Dissertation [13] definiert Kummer Referenznetze allgemein ohne Festlegung auf eine bestimmte Programmiersprache. Bei der Implementierung von Renew wurde als Programmiersprache aus unterschiedlichen Gründen Java gewählt, wie der Möglichkeit dynamisch externe Klassen zu laden, der Freispeicherverwaltung und das Vorhandensein von geeigneten Bibliotheken. Die Syntax von *inscriptions* entspricht daher der Java Syntax, welche um zusätzliche Syntax erweitert wurde, um die Eigenschaften von Referenznetzen besser abbilden zu können. Diese erweiterte Syntax wird bei der Simulation von Referenznetzen durch Renew überprüft. Die Erweiterungen der Syntax werden im Folgenden erläutert.

Deklaration von Variablen

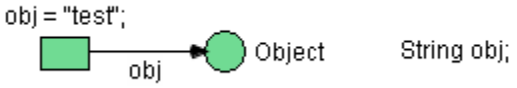
In farbigen Petri Netzen und vor allem in Referenznetzen werden Variablen anders verwendet als in Java. Im Gegensatz zu Java muss eine Variable eines Referenznetzes zwar einen Namen, aber nicht zwangsläufig einen Typ besitzen. Es wird jedoch empfohlen, Typen zu verwenden, um frühzeitig Fehler zu erkennen. Variablen werden global in einem *declaration node* deklariert, haben jedoch keinen global sichtbaren Wert. Eine Variable erhält erst zum Zeitpunkt des Feuerns der mit ihr verbundenen Transition einen Wert und verliert diesen, wenn das Feuern der entsprechenden Transition beendet ist. Der gleiche Variablenname kann daher durch unterschiedliche Transitionen verwendet werden, ohne dass diese ihre Unabhängigkeit voneinander verlieren. Ein Eintrag wie `String obj;` in Netz 4.1 bedeutet, dass jede Variable des Namens `obj`, welche an einer beliebigen Stelle des Netzes verwendet wird, vom Typ `String` ist.

Der *declaration node* kann zusätzlich auch `import` Anweisungen enthalten, wie sie auch in Java verwendet werden. Dem Renew Rahmenwerk müssen dabei unbekannte Klassen durch diese `import` Anweisungen bekannt gemacht werden.

[1] http://www.renew.de/

Inscriptions von Stellen

Wie es bei farbigen Petri Netzen üblich ist, beschreiben *inscriptions* von Stellen die Typen von Tokens, die die jeweilige Stelle enthalten kann. Bei Referenznetzen können diese Typen nicht nur primitive Datentypen wie ganzzahlige Integer Werte, sondern auch Referenztypen wie Arrays oder allgemeine Objekte sein. Bei der Nutzung von Referenztypen kann dabei die Klassenhierarchie ausgenutzt werden. Ist eine Stelle durch eine Klasse A typisiert, so kann diese Stelle Objekte vom Typ A oder von A abgeleitete Typen enthalten. Ist der Typ einer Stelle `Object`, so kann jedes Objekt ein Token dieser Stelle sein, wie in Netz 4.1 demonstriert. Dort wird ein `String` Objekt erzeugt und in eine Stelle vom Typ `Object` platziert.



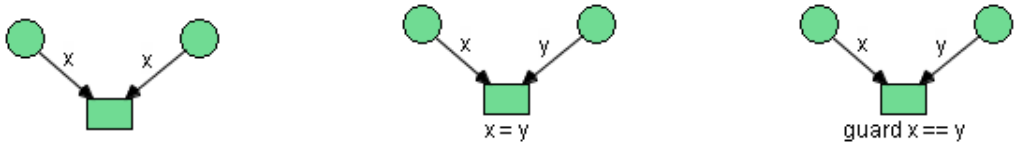
Netz 4.1: Beispiel der Typisierung einer Stelle

Bei Renew ist die Typisierung einer Stelle nicht erforderlich, senkt jedoch die Fehleranfälligkeit eines Netzes, da inkorrekte Tokens nicht erst zur Laufzeit als inkorrekt erkannt werden. Wird eine Stelle wie in Netz 4.1 typisiert, so muss jede Variable ihrer Kanten ebenfalls typisiert sein. Dies geschieht wie bereits beschrieben durch den *declaration node*.

Inscriptions von Kanten

Inscriptions von Kanten sind Bezeichnungen für Variablen. Zum Zeitpunkt des Feuerns einer Transition werden die eingelesenen Tokens mit dem Variablennamen verknüpft, welcher in der *inscription* der jeweiligen Kante angegeben ist. Diese Verknüpfung findet durch einen Unifikationsalgorithmus statt. Das Token kann bei der Verarbeitung der Transition durch diesen Variablennamen angesprochen werden.

Durch die Variablennamen werden indirekt Bedingungen an das Feuern der zugehörigen Transition gestellt. Ein Token kann nur dann von einer Transition eingelesen werden, wenn es von dem gleichen Typ ist wie die Variable der zugehörigen Kante. Zusätzlich findet eine Unifikation auf Basis der Variablennamen statt. Sei dies durch die drei Teilnetze in Netz 4.2 verdeutlicht.



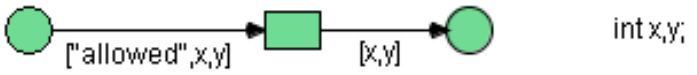
Netz 4.2: Drei Möglichkeiten der Gleichheitsprüfung mit und ohne Unifikation

Im linken Teilnetz von Netz 4.2 verwenden beide Kanten denselben Variablennamen. Es werden daher beide eingelesene Tokens miteinander unifiziert. Die Transition kann daher nur feuern, wenn es ein Paar aus zwei Tokens gibt, bei der die Unifikation ein positives Resultat liefert. Die gleiche Situation ergibt sich bei dem mittleren Teilnetz. Die Gleichsetzung von `x` und `y` bewirkt eine Unifikation beider Tokens, wie im linken Teilnetz. Hier ist der Operator `=` nicht mit dem Zuweisungsoperator von Java zu verwechseln, auch wenn er in einer ähnlichen Weise verwendet wird. Dies zeigt sich deutlich bei komplexeren Bedingungen wie `x*2=y+5`, die als *inscription* erlaubt sind, jedoch keine korrekte Zuweisung im Sinne der Java Syntax darstellen. Die

Implementierung des hier verwendeten Unifikationsalgorithmus basiert bei der Unifikation von Objekten auf der Verwendung der `equals(Object)` Methode und nicht des `==` Operators. Das rechte Teilnetz von Netz 4.2 verwendet keine Unifikation, prüft aber dennoch die Gleichheit von `x` und `y`. Wie dies geschieht wird im nachfolgenden Unterkapitel erläutert.

Besitzt eine Transition ein Paar aus eingehender und ausgehender Kante, die beide denselben Variablennamen verwenden, so wird auch hier eine Unifikation durchgeführt. Das in der Stelle der ausgehenden Kante erzeugte Token besitzt dabei den gleichen Wert oder die gleiche Referenz wie das eingelesene Token. Auf diese Weise kann eine Referenz durch ein Referenznetz transportiert werden, sodass mehrere Transitionen dasselbe Objekt zur Verarbeitung nutzen können.

Die Implementierung des Unifikationsalgorithmus ist in [13] detailliert erläutert. Hier soll daher nur auf die Verarbeitung von Tupeln und Listen durch den Unifikationsalgorithmus näher eingegangen werden. Um die Verarbeitung einer Menge von Daten zu erleichtern, wurde in Renew sowohl eine Tupelimplementierung als auch eine Listenimplementierung definiert, welche direkt durch eine entsprechende Syntax im Netz verwendet werden können. Ein Beispiel eines Tupels ist `["string",2,1.1]`. Das Tupel besteht hier aus drei Elementen, die durch Kommas getrennt sind, wobei Anfang und Ende des Tupel durch eckige Klammern gekennzeichnet sind. Das Tupel kann aus unterschiedliche Typen bestehen, die identisch oder kompatibel zueinander sein müssen, wie in diesem Fall `String`, `int` und `double`. Für Listen gelten die gleichen Eigenschaften wie für Tupel. Listen können dabei wie Tupel genutzt werden, verwenden jedoch geschweifte Klammern zur Begrenzung der Liste, z.B. {`"string",2,1.1`}. Im Gegensatz zu Tupeln ist die Länge von Listen variabel und kann daher zusätzlich dynamisch angesprochen werden. Eine Liste kann durch {`x:y`} angegeben werden, wobei links die ersten n Elemente der Liste (hier $n = 1$) stehen und `y` wiederum eine Liste ist.



Netz 4.3: Beispiel von Unifikation bei Verwendung eines Tupels

Da die Unifikation von Tupeln und Listen identisch vorgenommen wird, werden in Netz 4.3 beispielhaft die Unifikation von Tupeln dargestellt und beschrieben. Die einzelnen Komponenten des Tupels werden dabei unabhängig voneinander unifiziert. Die Transition in Netz 4.3 kann daher genau dann feuern, wenn die Eingangsstelle ein Tupel der Länge 3 enthält, dessen erstes Element ein `String` Objekt mit dem Wert `"allowed"` ist und dessen zweites sowie drittes Element vom Typ `int` sind. Beim Feuern der Transition wird dabei ein Tupel in der Ausgangsstelle platziert, dessen Elemente `x` und `y` mit den Werten des Tupels der Eingangsstelle unifiziert werden. Beim Feuern der Transition wird dabei implizit ein Tupelobjekt erzeugt.

Inscriptions von Transitionen

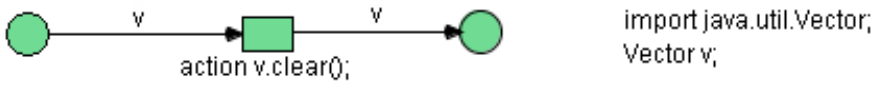
Inscriptions von Transitionen können Bedingungen an das Feuern der Transition stellen und Programmcode enthalten.

Wie die Kombination von *inscriptions* von Transitionen und Kanten durch Verwendung von Unifikation zur Definition von Bedingung genutzt werden kann, wurde bereits im vorigen Abschnitt erläutert. Eine zusätzliche Möglichkeit der Definition von Bedingungen ist die Verwendung des `guard` Schlüsselwortes, wie im rechten Teilnetz von Netz 4.2 dargestellt. Eine `guard` Bedingung nutzt dabei keine Unifikation, sondern erfordert die Angabe eines booleschen Ausdrucks. Eine Transition kann nur dann feuern, wenn dieser Ausdruck bei der Auswertung den

Wahrheitswert `true` ergibt. Im Falle des rechten Teilnetzes von Netz 4.2 kann die Transition nur Feuern, wenn `x` und `y` den selben Wert besitzen. Hier sei angemerkt, dass als boolescher Ausdruck jeder boolesche Ausdruck in Java Syntax angegeben werden kann. Der hier verwendete `==` Operator ist daher der in Java definierte `==` Operator.

Im vorigen Abschnitt wurde klargestellt, dass der `=` Operator Unifikation bewirkt und kein Zuweisungsoperator im Sinne von Java ist. Daraus ergibt sich jedoch das Problem, dass es nun keinen Zuweisungsoperator gibt, welcher in *inscriptions* verwendet werden kann. Zusätzlich kann es sein, dass der Programmcode von *inscriptions* ausgeführt wird, obwohl die Transition nicht feuert. Dieser Fall kann beispielsweise bei Benutzung der *inscription* `x = y.method();` auftreten. Bei dem Versuch der Unifikation muss die Methode `method()` des Objekts `y` aufgerufen werden, damit das Ergebnis mit `x` unifiziert werden kann. Daher kann erst nach dem Aufruf der Methode `method()` entschieden werden, ob die Transition feuern kann oder nicht. Dies ist nicht weiter problematisch, wenn die aufgerufene Methode rein lesender Natur ist, also den internen Zustand von `y` nicht verändert. Wird `y` jedoch durch den Methodenaufruf verändert, so darf die Methode nicht in Kombination mit Unifikation verwendet werden.

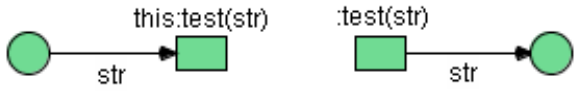
Um dennoch einen richtigen Zuweisungsoperator zur Verfügung zu stellen, bietet Renew das Schlüsselwort `action` an, wie in Netz 4.4 dargestellt. Zeilen, denen das `action` Schlüsselwort vorangeht, werden bei der Unifikation nicht berücksichtigt und werden nur ausgeführt, wenn bereits feststeht, dass die Transition feuert. Ausdrücke nach einem `action` Schlüsselwort können normale Java Ausdrücke sein. Daher wird ein verwendeter `=` Operator als Zuweisungsoperator verarbeitet und nicht zur Unifikation verwendet. In Netz 4.4 wird beim Feuern der Transition ein Vektor der Eingangsstelle gewählt, geleert und in der Ausgangsstelle platziert. In solchen oder ähnlichen Situationen, bei denen Methodenaufrufe Tokens verändern, sollte das `action` Schlüsselwort immer verwendet werden, um sicherzustellen, dass die Veränderung nur dann stattfindet, wenn die Transition tatsächlich feuert.



Netz 4.4: Beispiel der Verwendung einer `action` Anweisung

Renew bietet die Möglichkeit, Transitionen durch synchrone Kanäle direkt auszulösen. Dazu dient die *inscription* `:test(str)` in Netz 4.5, wobei `test` für einen beliebigen, selbst wählbaren Namen steht und `str` eine normale Variable ist, der nicht notwendigerweise ein Typ zugewiesen sein muss. Die zugehörige Transition kann bei Verwendung dieser *inscription* nur dann feuern, wenn diese explizit durch eine andere Transition aufgerufen wird, selbst wenn alle ihre Eingangsstellen benutzbare Tokens beinhalten. Der Aufruf dieser Transition erfolgt durch die *inscription* `this:test(str)` unter Verwendung des zuvor gewählten Namens und entsprechender Variablen. Der Aufruf der linken Transition in Netz 4.5 bewirkt, dass der Wert oder die Referenz eines Tokens der Eingangsstelle an die rechte Transition weitergeleitet wird und von dieser in der Ausgangsstelle platziert wird.

Diese Form des Aufrufs durch synchrone Kanäle kann auch in einem beliebigen Codefragment außerhalb des Netzes erfolgen, um Daten in das Netz einzuspeisen. Wie dies umgesetzt werden kann, wird im folgenden Kapitel erläutert.



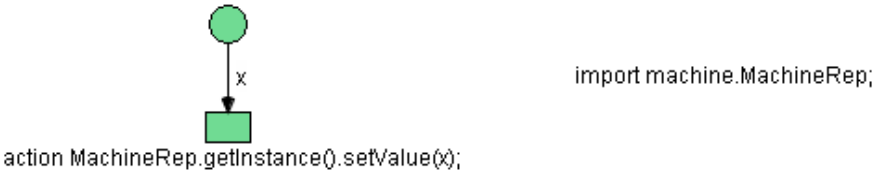
Netz 4.5: Beispiel der Nutzung eines synchronen Kanals

4.2.2 Einbettung des Petri Netzes

Die Schnittstelle der Interaktionslogik ist, wie in Abschnitt 4.1.1 vorgestellt, beschrieben durch Eingabe- und Ausgabestellen. Es ist jedoch bei Renew nicht vorgesehen, während einer Simulation Tokens in beliebige Stellen zu platzieren. Um dies dennoch zu realisieren, müssen zusätzliche Transitionen zu Hilfe genommen werden, wie im Folgenden verdeutlicht wird.

Aufrufe aus dem Netz

Das Petri Netz der Interaktionslogik muss durch zwei Schnittstellen sowohl mit der Aktionslogik, also auch mit der physische Repräsentation der Benutzerschnittstelle verbunden sein. Dazu werden zwei Klassen `Machine` und `UserInterface` definiert. `Machine` repräsentiert dabei die Aktionslogik und ermöglicht den Aufruf ihrer Funktionalität. `UserInterface` steht für die physische Repräsentation und ermöglicht so die Realisierung aller Rückmeldungen der Interaktionslogik an die Benutzerschnittstelle. Zur Verarbeitung von Ausgabestellen der Interaktionslogik wird für jede Ausgabestelle eine Transition definiert, welche die zugehörige Methode der Klasse `Machine` bzw. `UserInterface` aufruft, wie in Netz 4.6 dargestellt. Die Aufgabe der Transition ist dabei die Entnahme des Tokens der Ausgabestelle und der Aufruf der zugehörigen Methode der Aktionslogik bzw. physischen Repräsentation. Wie in Netz 4.6 dargestellt, wird dazu das `action` Schlüsselwort verwendet um ungewollte Nebeneffekt zu verhindern.

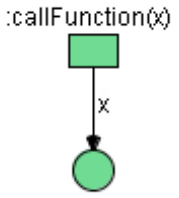


Netz 4.6: Realisierung eines Aufrufs nach außen

Das Problem bei diesem Ansatz ist, dass dem Netz das zur Laufzeit verwendete `Machine` bzw. `UserInterface` Objekt bekannt sein muss. Dies ist am besten durch eine Singleton Implementierung der Klassen zu realisieren. Dazu wird beispielsweise ein Klasse `MachineRep` definiert, die eine statische Methode `getInstance()` besitzt, die bei jedem Aufruf das verwendete `Machine` Objekt zurückgibt. Auf diese Weise ist sichergestellt, dass immer das gleiche `Machine` Objekt verwendet wird. Methoden der `Machine` Klasse können direkt auf dem von `getInstance()` zurückgegebenen Objekt aufgerufen werden. Analog dazu wird eine zweite Klasse `UserInterfaceRep` definiert, welche die gleiche Funktion für das zur Laufzeit verwendete `UserInterface` Objekt bereitstellt.

Aufrufe an das Netz

Aufrufe, welche von Außen an das Netz gerichtet sind, können durch synchrone Kanäle realisiert werden. Dazu wird, wie schon bei den Ausgabestellen, für jede Eingabestelle eine zusätzliche Transition benötigt. Diese wird wie in Netz 4.7 mit einem Namen versehen und platziert beim Aufruf die Eingabedaten als Tokens in der Eingabestelle. Der Aufruf einer solchen Transition kann innerhalb des Netzes und bei Verwendung einer Stub-Klasse auch außerhalb des Netzes an einer beliebigen Stelle im Java Code stattfinden.



Netz 4.7: Realisierung eines Aufrufs nach innen

Eine Stub-Klasse ist eine Java Klasse, welche von Renew erzeugt werden kann, um Zugriffe auf Netze zu ermöglichen. Dazu muss eine Stub-Datei generiert werden, aus der diese Klasse erzeugt werden kann. Ein Beispiel einer solchen Datei ist in Listing 4.1 angegeben. Die Syntax der Stub-Datei ist dabei Java sehr ähnlich. Das `package` der erzeugten Klasse entspricht dem angegebenen `package`. `NetClass` steht für den Namen der erzeugten Klasse und `someNet` für den Namen des Netzes, auf welches über diese Klasse zugegriffen wird. Innerhalb der geschweiften Klammern erfolgt eine Angabe der Methoden in Java Syntax, die die erzeugte Klasse besitzen soll. Die Sichtbarkeit aller erzeugten Methoden ist automatisch `public`. Innerhalb der Definition einer Methode wird die aufzurufende Transition in der Syntax der Referenznetze `this:callFunction(x)` angegeben. Der Name einer Methode der Klasse ist dabei unabhängig von dem Namen des verwendeten synchronen Kanals.

```
package net;
class NetClass for net someNet {
  void callSomeFunction(String x) {
    this:callFunction(x);
  }
}
```

Listing 4.1: Beispiel einer Stub-Datei

Eine Stub-Datei, wie die in Listing 4.1 angegebene, kann durch die Renew Klasse `StubCompiler` in eine Java Klasse umgewandelt werden.

Flexibilität des Ansatzes

Das für die Interaktionslogik verwendete Petri Netz muss veränderbar sein. Dies gilt insbesondere für die Stellen, welche der Kommunikation mit der physischen Repräsentation der Benutzerschnittstelle dienen. Dazu müssen gegebenenfalls die Möglichkeiten der Aufrufe erweitert werden, welche in das Netz oder aus dem Netz stattfinden. Die Möglichkeiten der Aufrufe aus dem Netz sind ohne Probleme erweiterbar, da nur entsprechende Stellen und Transitionen mit den nötigen *inscriptions* hinzugefügt werden müssen. Durch Verwendung des Singleton Pattern kann jede Transition das verwendete `Machine` bzw. `UserInterface` Objekt abfragen und direkt darauf Methoden aufrufen.

Schwieriger ist die Erweiterung der Möglichkeiten von Aufrufen in das Netz. Auch hier ist es nötig, neue Stellen und Transitionen mit entsprechenden *inscriptions* zu erzeugen. Die so neu definierten synchronen Kanäle sind dadurch jedoch nicht von außen zugreifbar. Um sie auch von außen zugreifbar zu machen, ist es nötig eine neue Stub-Datei zu generieren und diese durch die Klasse `StubCompiler` zu kompilieren. Diese veränderte Klasse muss anschließend dynamisch durch einen `ClassLoader` in das System geladen werden. Da sich die Stub-Klasse dadurch zur Laufzeit verändert, muss zur Nutzung dieser Klasse auf die Reflection-API von Java zurückgegriffen werden. Die Nutzung der Möglichkeit dynamisch Klassen zu laden und die Nutzung der Reflection-API erhöhen die Komplexität der resultierenden Software, ermöglichen jedoch eine flexible Veränderung des Netzes und erfüllen damit die Anforderungen an ein Rahmenwerk zur Verwendung formal modellierter Interaktionslogik.

4.2.3 Simulation mit Renew

Die Simulation eines Petri Netzes mit Renew ist durch die Klasse `SimulatorPlugin` realisiert. Um ein `SimulatorPlugin` Objekt verwenden zu können, muss zuerst der `PluginManager` erzeugt werden. Dies kann wie in Listing 4.2 geschehen, wobei `url` die `URL` der `Loader` Klasse ist. `Loader` ist eine Klasse von Renew, welche zum Laden des visuellen Editors benötigt wird. Wichtig ist hier anzumerken, dass sich der `PluginManager` automatisch deaktiviert, wenn kein Plugin angegeben ist, welches als "Blocker" fungiert. Daher wird ein solches Plugin vor dem Aufruf der `main`-Methode dem `PluginManager` zugewiesen.

```
IPlugin blocker = new PluginAdapter(PluginProperties.getUserProperties());
PluginManager.getInstance().blockExit(blocker);
PluginManager.main(new String[]{},url);
```

Listing 4.2: Erzeugung eines `PluginManager` Objekts

Ist der `PluginManager` erzeugt, so kann durch die in Listing 4.3 angegebenen Schritte eine Simulation gestartet werden. Die Implementierung des `SimulatorPlugin` basiert auf dem Singleton Pattern, sodass die aktuelle Instanz dieser Klasse durch die Methode `getCurrent()` abgerufen werden muss. Im zweiten Schritt wird die Konfiguration durch Angabe eines `Properties` Objekts konfiguriert, welches jedoch auch `null` sein kann. Durch dieses Objekt können Einstellungen der Simulation konfiguriert werden, wie die Anzahl der verwendeten Simulationsthreads oder deren Priorität. Im dritten und vierten Schritt wird dem `SimulatorPlugin` durch ein `ShadowNetSystem` die verwendeten Netze bekannt gemacht und diese durch Aufruf der `createNetInstance(String)` Methode unter Angabe ihres Namens für die Simulation instantiiert. Das `ShadowNetSystem` Objekt muss für diesen Schritt vorher generiert werden, indem alle zu verwendenden Netze importiert werden. Im letzen Schritt wird die Simulation gestartet und läuft von da an nebenläufig ab.

```
SimulatorPlugin simulatorPlugin = SimulatorPlugin.getCurrent();
simulatorPlugin.setupSimulation(null);
simulatorPlugin.insertNets(shadowNetSystem);
simulatorPlugin.createNetInstance("nameOfNet");
simulatorPlugin.getCurrentEnvironment().getSimulator().startRun();
```

Listing 4.3: Notwendige Schritte zum Start einer Simulation

4.3 Inkrementelle Erweiterung der Benutzerschnittstelle eines Siedewasserreaktors

Im Folgenden wird anhand einer Simulation eines Siedewasserreaktors ein praktischer Einsatz der Interaktionslogik demonstriert. Der dabei zugrunde liegende Simulator des Kraftwerks stammt von Henrik Eriksson und ist im Internet zugänglich[2]. Die Animation des Kraftwerks und Teile des Simulators wurden von Benjamin Weyers im Rahmen seines Dissertationsvorhabens verbessert und freundlicherweise für diese Arbeit zur Verfügung gestellt. Die Simulation des Siedewasserreaktors wurde im Rahmen der Arbeit [4] untersucht, modelliert und im Bezug auf mögliche Fehler durch die Interaktion analysiert. Die Implementierung der hier vorgestellten Demonstration befindet sich auf der beigelegten CD (siehe Anhang B) als ausführbare Datei.

Das in Abbildung 4.4 dargestellte Kraftwerk besteht aus einem Reaktor, einem Kühltank und einer Turbine mit angeschlossenem Generator. Diese Teile sind durch Röhren, Ventile und Pumpen verbunden. Die Pumpe, die sich rechts des Kühltanks befindet, sorgt dabei für die Kühlung dieses Kondensators, der den Restdampf in Wasser kondensiert. Die anderen beiden Pumpen transportieren Wasser vom Kühltank zum Reaktor. An jeder dieser beiden Pumpen befindet sich ein Ventil, mit dem der Wasserfluss blockiert werden kann. Im Reaktor erzeugter Dampf wird durch eine Turbine in den Kühltank zurückgeleitet und so Strom produziert. Das verdampfte Wasser kann jedoch auch über ein separates Rohr direkt in den Kühltank geleitet werden ohne die Turbine anzutreiben, um die Produktion des Stroms zu senken oder einen Turbinenschaden zu umgehen. An dieser Stelle kann die Richtung des Dampfes über zwei Ventile gesteuert werden.

Die Oberfläche des Siedewasserreaktors ist in Abbildung 4.4 dargestellt. Der Simulator des Kraftwerks ist dabei mit der Oberfläche allein über die Interaktionslogik verbunden. Diese Verbindung erfolgt wie in Kapitel 4.1 beschrieben. Für die Interaktionslogik werden Referenznetze verwendet sowie Renew zur Simulation dieser Netze eingesetzt. Die Möglichkeiten der Kommunikation des Netzes der Interaktionslogik mit der physischen Repräsentation der Schnittstelle und der Aktionslogik sind dabei wie in 4.2 beschrieben realisiert.

Am rechten Rand in Abbildung 4.4 sind insgesamt 7 Paare von Knöpfen zu sehen. Jedes dieser Paare wird dazu verwendet, um ein Ventil des Kraftwerks zu öffnen und zu schließen oder eine Pumpe zu aktivieren und zu deaktivieren. Beim Drücken einer der Knöpfe ruft die Oberfläche eine vorher definierte Methode des Netzes der Interaktionslogik auf. Der Aufruf wird durch das Netz geleitet und schlussendlich an den Simulator des Kraftwerks weiter gegeben. Dieser nimmt die Daten entgegen und verarbeitet sie. Das gesamte Netz ist in Netz 4.8 dargestellt.

Die oberen Stellen dieses Netzes dienen der Kommunikation mit der physischen Repräsentation, während die unteren Stellen die Kommunikation mit der Aktionslogik sicherstellen. Beide Mengen von Stellen sind für eine bessere Übersichtlichkeit gelb markiert. Bis auf die Kette an Stellen und Transitionen, die die `state` Variable verwendet, sind alle oberen Stellen Eingabestellen und alle unteren Stellen Ausgabestellen.

Zur Verarbeitung der Aufrufe aus dem Netz und deren Weiterleitung an den Simulator oder an die Visualisierung sind die Klassen `NuclearNetRep` und `NuclearGUIRep` definiert. Diese besitzen jeweils eine `getInstance()` Methode, die das aktuell verwendete Simulator bzw. GUI Objekt zurück gibt. Den durch diese Methode erhaltenen Objekten werden Daten direkt durch den Aufruf entsprechender Setter-Methoden übergeben. Eine der beiden wichtigsten Teilnetze ist die Kette aus Stellen und Transitionen, die vom Simulator durch den synchronen Kanal `:setNewState(state)` aufgerufen wird und eine `NPPState` Objekt an die physische Repräsen-

[2] http://www.ida.liu.se/ her/npp/demo.html

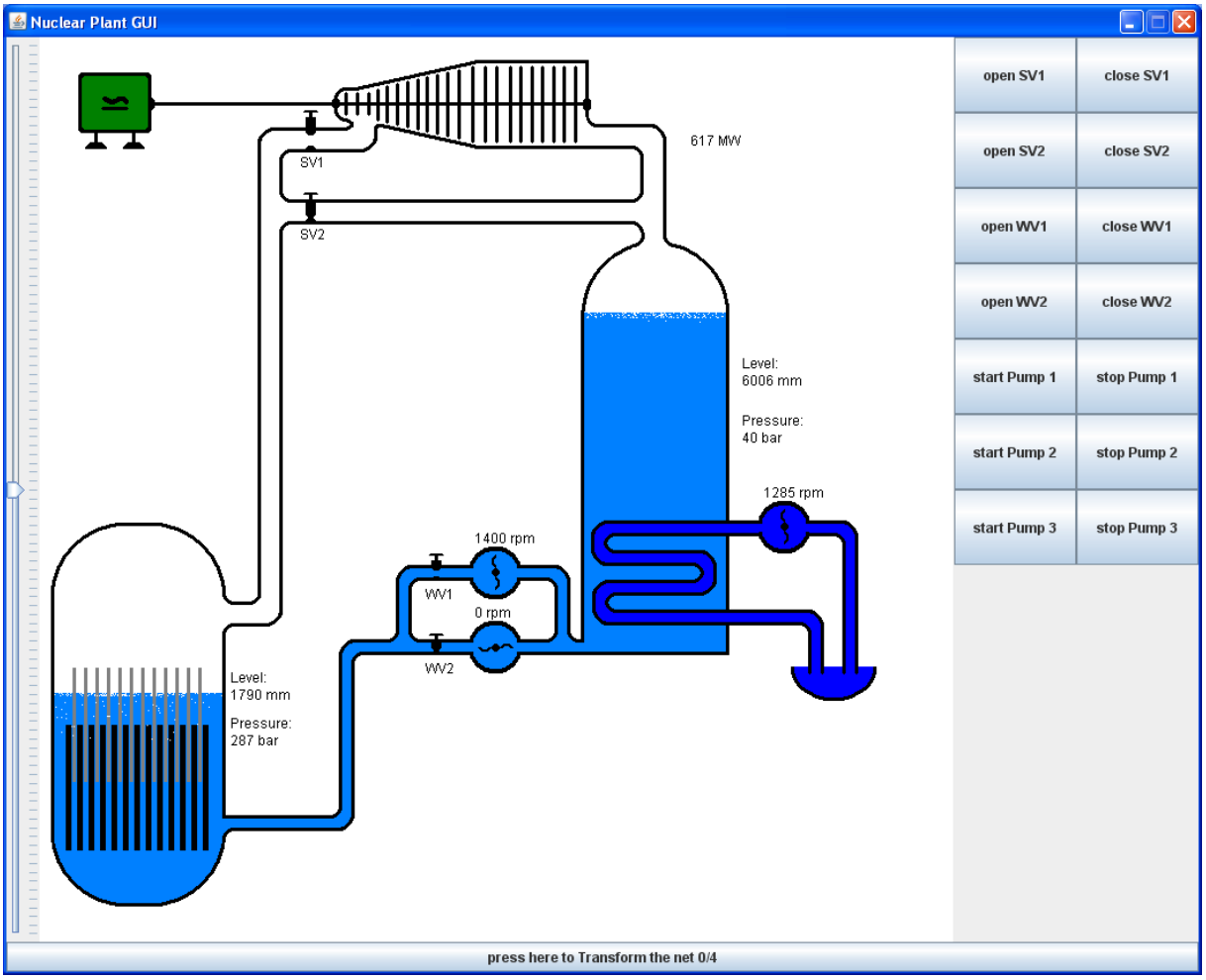
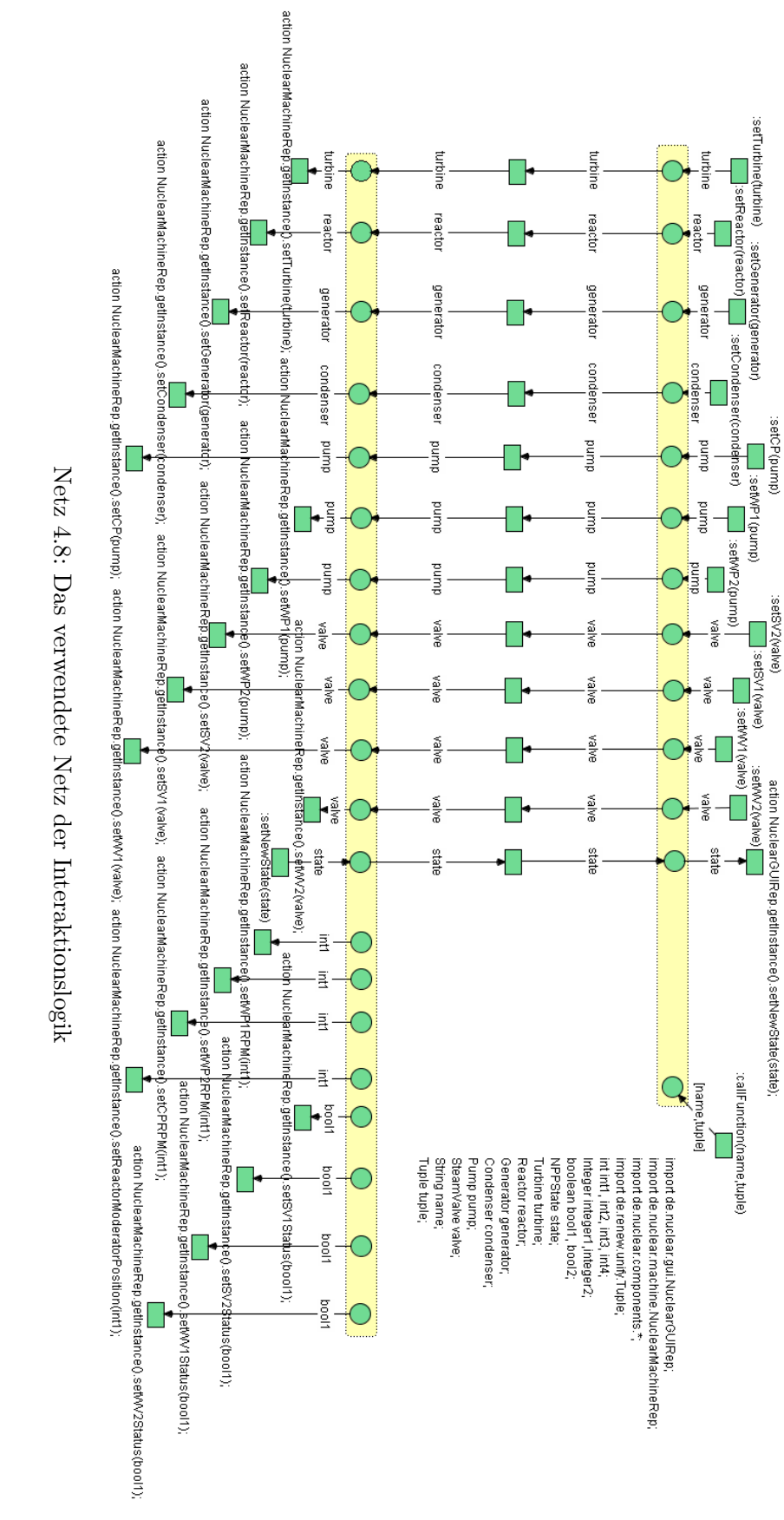


Abbildung 4.4: Oberfläche des Siedewasserreaktors vor den Transformationen

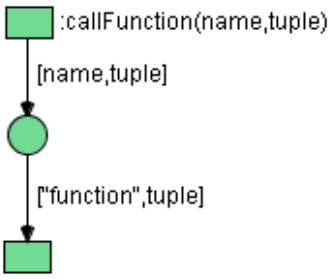


Netz 4.8: Das verwendete Netz der Interaktionslogik

tation weiterleitet. Diese stellt den Zustand des Simulators unter Verwendung der Daten des `NPPState` Objekts in Form der animierten Kraftwerksdarstellung dar. Um die Benutzerschnittstelle immer aktuell zu halten, ruft der Simulator die `setNetState(NPPState)` Methode des Netzes regelmäßig innerhalb der umgebenden Implementierung auf.

Die oben rechts befindliche Stelle und die ihr angeschlossene Transition sind für die Erweiterung der Interaktionslogik gedacht. Dazu wird diese Methode von der physischen Repräsentation unter Angabe eines Namens und eines `Tuple` Objekts aufgerufen. Wird die Interaktionslogik später verändert, so werden zusätzliche Transitionen eingeführt, die Daten aus dieser Stelle lesen. Wegen dieser Realisierung ist die Veränderung der Stub-Klasse zur Laufzeit nicht erforderlich. Diese Lösung bietet keine sehr hohe Flexibilität, ist jedoch für dieses Beispiel ausreichend. Um nach einem Aufruf der `callFunction(String,Tuple)` Methode bestimmen zu können, welche Transition die in der angeschlossenen Stelle abgelegten Daten verarbeiten soll, wird ein Name verwendet. Dieser muss daher beim Aufruf der Methode übergeben werden. Eine an diese Stelle angeschlossene Transition muss den Namen prüfen, so dass diese nur feuern kann, wenn die Daten für sie bestimmt sind. Wie in Netz 4.9 dargestellt, kann dies durch Unifikation des Namens (erstes Element des Tupels) geschehen. Die untere Transition kann dabei nur Tupel einlesen, deren erstes Element des Tupels ein String Objekt mit dem Inhalt `function` ist.

Das hier für die Interaktionslogik verwendete Petri Netz leitet nur Nachrichten weiter ohne eigene Logik zu implementieren. Die Eingabe- bzw. Ausgabestellen, die nur mit einer Transition verbunden sind, werden für später bei der Transformation entstehende Transitionen benötigt.



Netz 4.9: Beispiel einer im Nachhinein eingefügten Transition

Ablauf der Erweiterung

Für die Oberfläche des Atomkraftwerks sind vier Erweiterungen implementiert. Diese sind als vier weitere Knöpfe am rechten Rand des Fensters in Abbildung 4.5 zu erkennen. Jede dieser Erweiterungen stellt eine Funktion dar, die vollständig mit Hilfe der Interaktionslogik realisiert ist. Folgende Erweiterungen sind implementiert:

1. Die Oberfläche wird um einen Notfall-Knopf erweitert, der bei Aktivierung alle Ventile schließt, Pumpen stoppt und die Brennstäbe in den Ruhezustand versetzt. Diese Funktion bewirkt eine Reaktorschnellabschaltung und ermöglicht schnelle Notfallmaßnahmen im Falle eines Unfalls.

2. Es wird ein Knopf eingefügt, der die Ventile, Pumpen und Brennstäbe auf Anfangseinstellungen zurückstellt. Wurde der Reaktor beispielsweise durch den Notfall-Knopf heruntergefahren, so kann durch den hier eingefügten Knopf der Normalbetrieb wiederaufgenommen werden.

3. Es wird ein Knopf eingefügt, der beide Ventile zwischen Reaktor und Kühltank öffnen und die dazugehörigen Pumpen startet. Diese Funktion ermöglicht einen schnellen Transport der Kühlflüssigkeit zum Reaktor, falls dieser sich zu schnell aufheizt.

4. Die Oberfläche wird um einen Knopf erweitert, der das Ventil zwischen Turbine und Reaktor schließt und das alternative Ventil öffnet. Im Falle eines Turbinenschadens oder notwendiger Wartungsarbeiten kann so die Turbine stillgelegt werden.

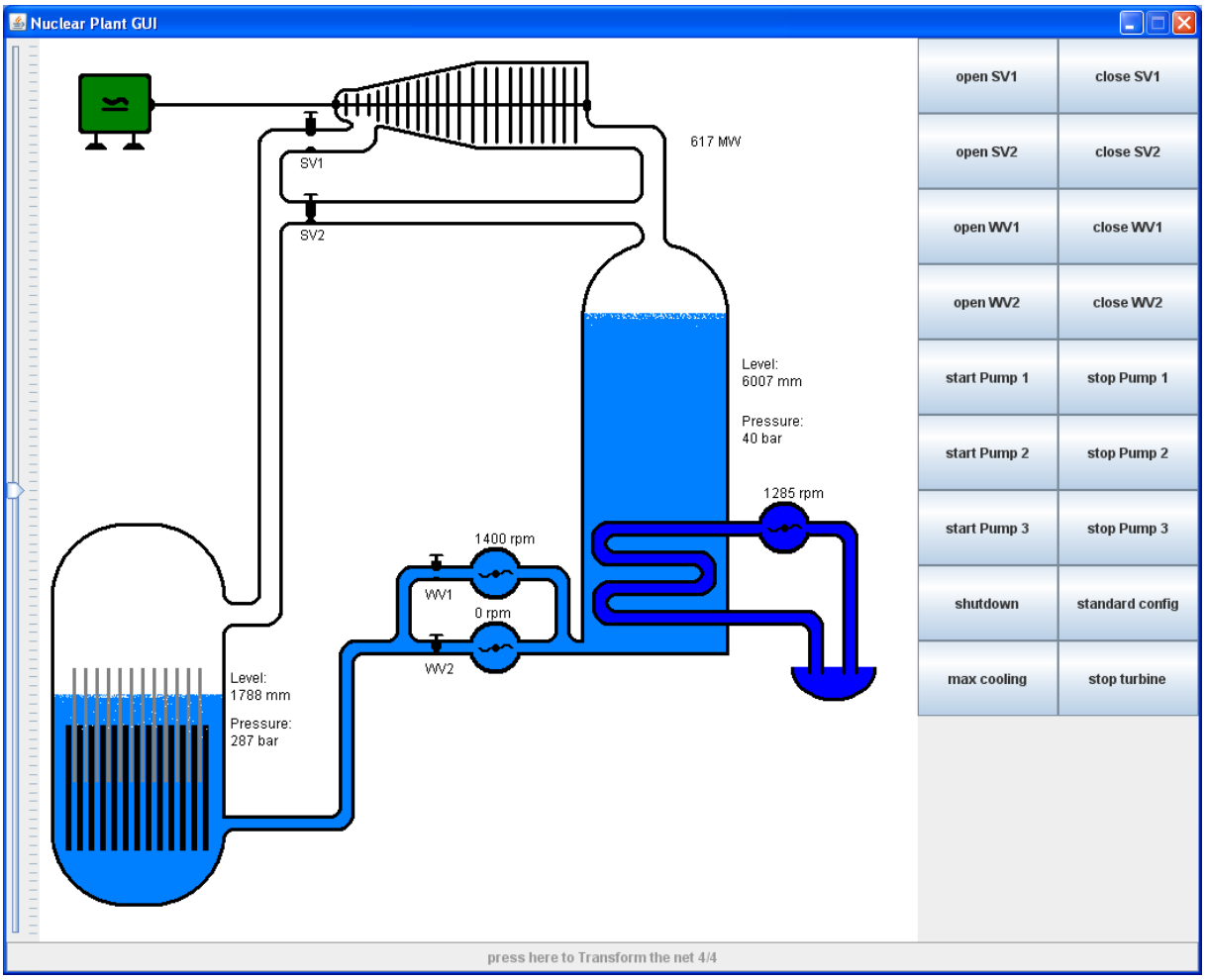
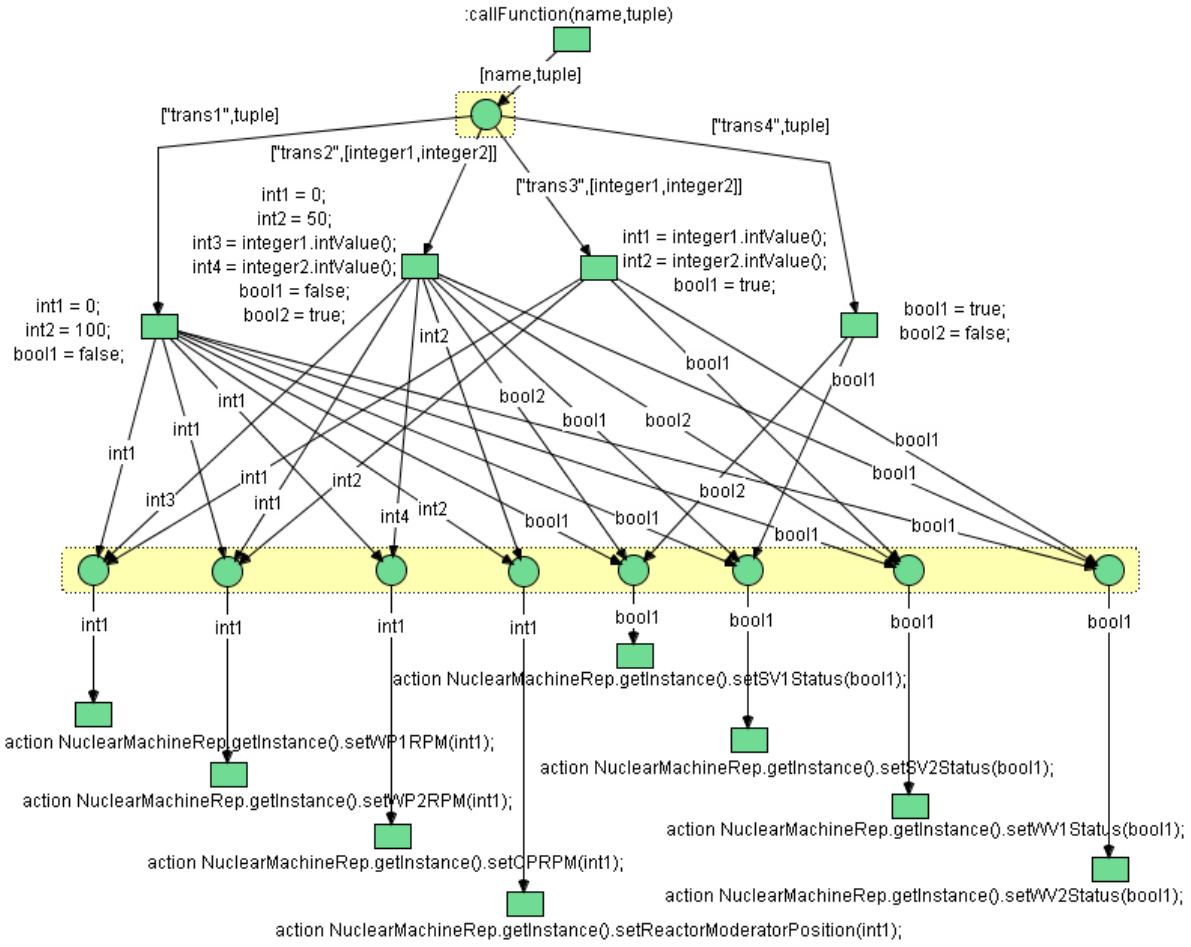


Abbildung 4.5: Oberfläche des Siedewasserreaktors nach den Transformationen

Netz 4.10 zeigt den rechten Ausschnitt von Netz 4.8 nach der Anwendung aller vier Transformationen. Jede der Transformationen erzeugt dabei eine Transition und die mit ihr verbundenen Kanten. Welche Transition durch welche Transformation erzeugt wird, kann anhand des ersten Elements des Tupels der zur Transition führenden Kante nachvollzogen werden. Die bei der jeweiligen Transformationen verwendeten Produktionen sind in Anhang C angegeben.

Beim Aufruf der `callFunction(String,Tuple)` Methode wird ein Tupelobjekt in die obere Stelle eingefügt. Das erste Element des Tupels (`name`) gibt an, durch welchen Knopf die Erzeugung des Tupels ausgelöst wurde. Durch die Unifikation auf Basis der *inscriptions* der Kanten, kann nur genau eine an der oberen Stelle angeschlossene Transition feuern. Beim Feuern einer Transition wird das Tupel gelöscht und entsprechend der *inscriptions* der Transition und der

ausgehenden Kanten neue Tokens generiert, die in die entsprechenden Ausgabestellen geleitet werden. Dadurch werden alle notwendigen Daten an den Simulator des Kraftwerks weitergeleitet.



Netz 4.10: Das resultierende Teilnetz nach allen vier Transformationen

Jede der oben aufgeführten Erweiterungen kann durch einen Klick auf den Knopf am unteren Rand des Fensters eingebunden werden. Um dies zu tun führt das Programm folgende Schritte aus:

1. Die für den jeweiligen Schritt definierte Transformationsregel wird ausgelesen und auf das aktuell verwendete Netz angewendet. Dazu wird das implementierte System zur Graphtransformation aus Kapitel 3 verwendet. Das Ergebnis der Transformation wird gespeichert und anschließend in die `ShadowNetSystem` Datenstruktur umgewandelt, die Renew zur Darstellung von Petri Netzen verwendet. Dieser Schritt entspricht der Rekonfiguration der Interaktionslogik.

2. Der einzufügende Knopf wird erzeugt, konfiguriert und anschließend in die Benutzeroberfläche eingefügt. Zusätzlich wird für den Knopf gespeichert, mit welchen Werten die `callFunction(String,Tuple)` Methode aufgerufen wird, wenn dieser gedrückt wird. Dieser Schritt entspricht dem Redesign der physischen Repräsentation.

3. Zuletzt muss die Simulation neu gestartet werden, damit die Änderungen übernommen werden können. Mit Simulation ist hier sowohl die Simulation des Kraftwerks, als auch die Simulation des Petri Netzes gemeint. Beide Simulationen werden daher gestoppt, mit den neuen Werten initialisiert und erneut gestartet.

Fazit

Die in diesem Kapitel vorgestellte Implementierung ist für dieses vereinfachte Beispiel ausreichend. Für mehr Flexibilität muss jedoch die Stub-Klasse anpassbar sein, was hier nicht realisiert wurde, da die angesprochenen Erweiterungen allein durch die `callFunction(String,Tuple)` Methode umsetzbar sind. Es wird zudem deutlich, dass bei diesem noch sehr einfachen Beispiel bereits große Netze und Produktionen erforderlich sind. Das verwendete Netz hat beispielsweise bereits vor den Transformationen über 1400 Zeilen in XML Darstellung.

Bei der Simulation des Petri Netzes steht zu befürchten, dass diese nicht schnell genug erfolgt und der Benutzer mit übermäßigen Zeitverzögerungen zu rechnen hat. Dieses Problem hat sich in der Implementierung nicht gezeigt, obwohl in regelmäßigen, kurzen Abständen `NPPState` Objekte durch die Maschine in das Netze eingebracht wurden. Ob sich das angesprochene Problem bei großen Netzen zeigt, müsste durch umfangreichere Tests der Performanz ermittelt werden.

5 Fazit & Ausblick

5.1 Fazit

Ziel dieser Arbeit war die Implementierung eines Systems zur Graphtransformation von farbigen Petri Netzen. Dazu wurden zunächst der mengentheoretische, kategorientheoretische sowie der logikorientierte Ansatz vorgestellt, wobei nur auf den kategorientheoretischen Ansatz näher eingegangen wurde. Dieser teilt sich in den *single pushout* (SPO) und den *double pushout* (DPO) Ansatz auf. Vor allem weil sein Verhalten in Konfliktfällen vorhersehbarer ist, als es beim SPO Ansatz der Fall ist, wurde der DPO Ansatz als Basis der Implementierung gewählt. Der DPO Ansatz wurde sowohl für allgemeine Graphen, als auch in einer auf Petri Netze spezialisierter Form dargestellt. Als Basis der späteren Implementierung wurde jedoch der allgemeine Ansatz verwendet, da der spezialisierte Ansatz durch die implizite Behandlung von Kanten zu viele Einschränkungen darstellt und damit eine Realisierung erschwert. Um eine spätere Verwendung in der Modellierung von Interaktionslogik zu ermöglichen, musste der gewählte Ansatz auf farbige Petri Netze erweitert werden. Dazu wurden verschiedene Ansätze der Transformation von *inscriptions* vorgestellt. Realisiert wurde in der Implementierung der Ansatz, der die wenigsten Löschungen erfordert. Dies ermöglicht das leichte Durchführen von kleinen Veränderungen des zu transformierenden Netzes unter minimalem Aufwand.

Zur Implementierung wurde der allgemeine Aufbau eines Systems zur Graphtransformation anhand eines Prozesses herausgestellt und durch geeignete Klassen in Java umgesetzt. Als Datenstruktur für Petri Netze und für Produktionen wurde aus Gründen der Austauschbarkeit auf XML gesetzt. Für Abbildungen zwischen Netzen wurde zur Erhöhung der Allgemeinheit der Implementierung die **Teil von** Relation auf XML Konten eingeführt. Durch diese Ordnungsrelation ist es möglich, einzelne Kindknoten von Elementen des Netzes zu verändern, ohne Beachtung anderer Kindknoten. Da in der Produktion nicht aufgeführte Knoten auch nicht beachtet werden, erfahren diese bei der Transformation keine Veränderung. Dies ist analog zum Verhalten des DPO Ansatzes, Knoten und Kanten des zu transformierenden Netzes nicht zu ändern, wenn diese nicht in der Produktion vorkommen.

Zum Schluss wurden adaptive, auf formal modellierter Interaktionslogik basierende Benutzerschnittstellen vorgestellt. Die Interaktionslogik wurde dabei auf Basis von Referenznetzen unter Verwendung von Renew modelliert und realisiert. Dieser spezielle Formalismus farbiger Petri Netze eignet sich vor allem wegen der Mächtigkeit der *inscriptions* und wegen seiner Nähe zu Java. Die Rekonfiguration von Interaktionslogik wurde mit Hilfe der erstellten Implementierung realisiert und anhand eines Beispiels vorgeführt.

Schlussendlich ermöglicht die erstellte Software die Transformation XML-basierter Petri Netze durch formal definierte Produktionen. Die Forderung des hohen Maßes an Austauschbarkeit, sowie die Anpassbarkeit des verwendeten Petri Netz-Formalismus konnten dabei erfüllt werden. Desweiteren wurde die Konformität zur theoretischen Grundlage sichergestellt.

5.2 Ausblick

Die Implementierung des Systems zu Graphtransformation orientiert sich stark an der theoretischen Basis. Diese bietet jedoch nicht alle Möglichkeiten der Transformation, die in der Praxis hilfreich wären. Eine dieser Möglichkeiten ist beim Matching für Elemente des linken Netzes Kindknoten anzugeben, die ihr Bild im zu transformierenden Netz nicht besitzen dürfen. Dies ist im theoretischen Ansatz nicht vorgesehen, in der Praxis jedoch vor allem in Verbindung mit der **Teil von** Relation hilfreich, da eine Einführung derartiger "verbotener" Kindknoten bei der Lösung des Problems der inhärenten Unsicherheit (3.4.6) helfen kann. Wird durch eine Produktion, einem Element des zu transformierenden Netzes ein neuer Kindknoten zugewiesen, so kann es bei Verwendung der **Teil von** Relation sein, dass das Element diesen Kindknoten bereits besitzt. Dies ist problematisch, wenn Elemente des Netzes diesen Kindknoten nur einmal besitzen dürfen. Die Korrektheit des Netzes kann nur durch eine zusätzliche Prüfung nach der Regelanwendung stattfinden. Wird dieser Kindknoten jedoch explizit in der Produktion für ein Element verboten, so würde das Matching bei der Regelanwendung als fehlerhaft erkannt, wenn das Bild des Elements diesen Kindknoten besitzt. Damit kann die Korrektheit des resultierenden Netzes bereits vor der Regelanwendung sichergestellt werden.

Ein weiteres Problem, welches sich aus der verwendeten Theorie ergibt, ist, dass nur die Veränderung von Stellen, Transitionen und Kanten eines Netzes in einer Produktion definierbar ist. Im Falle von Referenznetzen besitzen diese jedoch zusätzlich einen `name` Knoten und einen *declaration node*, welche direkte Kindknoten des Netzes und damit durch die Implementierung nicht veränderbar sind. Um Veränderungen dieser Teile des Netzes mit Hilfe von Produktionen zu ermöglichen, muss der verwendete Regelformalismus erweitert werden. Auch die Veränderungen solcher Kindknoten kann auf Basis der **Teil von** Relation geschehen, wie es bereits bei Stellen, Transitionen und Kanten der Fall ist.

Ein anderes Problem des verwendeten theoretischen Ansatzes ist die Mehrdeutigkeit des *pushout* Komplements bei Verwendung einer nicht-injektiven linken Produktionsseite. Diese wurde in der Implementierung durch Einführung einer Konvention umgangen. Allerdings tritt dabei das Problem auf, dass Produktionen bei Verwendung dieser Konvention möglicherweise nicht mehr invertierbar sind. Invertierbarkeit kann sichergestellt werden, wenn in der Produktion direkt festgelegt ist, in welcher Weise ein Element des Netzes gespalten bzw. fusioniert werden soll. Eine solche Beschreibung muss die Verarbeitung sowohl der Kanten, als auch der Kindknoten von Elementen des Netzes festlegen.

Zusammengefasst konnten folgende Arbeitspunkte für zukünftige Arbeiten identifiziert werden:

- Entwicklung eines Formalismus, der es ermöglicht, Kindknoten zu definieren, die das Bild einer Stelle, Transition oder Kante nicht besitzen darf.
- Erstellung einer Möglichkeit Kindknoten von Netzen zu transformieren, die keine Stellen, Transitionen oder Kanten sind.
- Erweiterung der Implementierung um eine flexiblere Verarbeitung der Mehrdeutigkeit verursacht durch nicht-injektive linke Produktionsseiten.

A RelaxNG Definitionen

Dieses Kapitel listet alle auf RelaxNG basierenden Definitionen auf, die in Kapitel 3 verwendet werden.

PNML Definition eines Referenznetzes

```
<?xml version="1.0" encoding="UTF-8"?>

  <grammar xmlns="http://relaxng.org/ns/structure/1.0"
       xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">

  <a:documentation>
    PNML definition of a reference net.
    version: 1.0
    author: Jan Stueckrath
  </a:documentation>

  <a:documentation>
    We include PNML with the correct URI for our Petri Net Type Definition.
  </a:documentation>
  <include href="basicPNML.rng">
    <define name="nettype.uri" combine="choice">
      <a:documentation>
        We define the net type URI declaring the namespace of this
        Petri net type definition.
      </a:documentation>
      <value>refNet</value>
    </define>
  </include>

  <define name="net.labels" combine="interleave">
    <interleave>
      <a:documentation>
        The net may have a name.
      </a:documentation>
      <optional>
        <element name="name">
          <ref name="text.node"/>
        </element>
      </optional>
      <a:documentation>
        The reference net may have a declaration node, where the used
        variables are declared. If there is no declaration node given,
        an empty one is assumed.
      </a:documentation>
      <optional>
        <element name="declaration">
          <ref name="inscription.content"/>
        </element>
      </optional>
    </interleave>
```

```
</define>

<a:documentation>
  An arc can have an inscription and a type (e.g. ordinary, double arc,
  etc.). If no type is given, the arc is assumed to be ordinary.
</a:documentation>
<define name="arc.labels" combine="interleave">
  <interleave>
    <optional><ref name="inscription.label"/></optional>
    <optional><ref name="arc.type"/></optional>
  </interleave>
</define>

<a:documentation>
  The type of an arc defines its behavior in the net.
</a:documentation>
<define name="arc.type">
  <element name="type">
    <ref name="text.node"/>
  </element>
</define>

<a:documentation>
  A place may have an inscription.
</a:documentation>
<define name="place.labels" combine="interleave">
  <optional><ref name="inscription.label"/></optional>
</define>

<a:documentation>
  A transition may have an inscription.
</a:documentation>
<define name="transition.labels" combine="interleave">
  <optional><ref name="inscription.label"/></optional>
</define>

<a:documentation>
  An inscription can define the behavior of a transition. It then
  contains code specifying how the input tokens are processed and how the
  output tokens are generated. Inscriptions of places and arc restrict
  the kind of tokens the place can contain and the arc can transport.
</a:documentation>
<define name="inscription.label">
  <element name="inscription">
    <ref name="inscription.content"/>
  </element>
</define>

<a:documentation>
  The inscription has a text node as data part and may be described with
  a standard graphics node.
</a:documentation>
<define name="inscription.content">
  <interleave>
    <optional>
      <element name="graphics">
        <ref name="annotationgraphics.content"/>
      </element>
    </optional>
```

```
      <ref name="text.node"/>
    </interleave>
  </define>

  <a:documentation>
    Defines a node containing text data.
  </a:documentation>
  <define name="text.node">
    <element name="text">
      <text/>
    </element>
  </define>

</grammar>
```

Listing A.1: PNML Definition eines Referenznetzes

Generische Definition der Produktion

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
     datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <a:documentation>
    Generic rule structure definition. Here the rule structure is defined
    without a petri net definition. To generate a full definition
    'net.definition' must be a non-empty.
    version: 1.0
    author: Jan Stueckrath
  </a:documentation>

  <start>
    <ref name="trans.rule"/>
  </start>

  <define name="trans.rule">
    <a:documentation>
      A rule consists of a petri net for the left side, the right side
      and the interface. Additionally it has two mappings, mapping the
      interface to the left side and the right side, here presented by
      'trans.mapping'.
    </a:documentation>
    <element name="rule">
      <interleave>
        <ref name="trans.left"/>
        <ref name="trans.interface"/>
        <ref name="trans.right"/>
        <ref name="trans.mapping"/>
      </interleave>
    </element>
  </define>

  <a:documentation>
    The left side is sufficiently defined through a single net.
  </a:documentation>
  <define name="trans.left">
    <element name="deleteNet">
      <ref name="net.definition"/>
```

```
    </element>
  </define>

  <a:documentation>
    The interface is sufficiently defined through a single net.
  </a:documentation>
  <define name="trans.interface">
    <element name="interface">
      <ref name="net.definition"/>
    </element>
  </define>

  <a:documentation>
    The right side is sufficiently defined through a single net.
  </a:documentation>
  <define name="trans.right">
    <element name="insertNet">
      <ref name="net.definition"/>
    </element>
  </define>

  <a:documentation>
    The net is not defined here. A full rule definition can be achieved
    by only replacing this definition by an actual net definition.
  </a:documentation>
  <define name="net.definition">
    <empty/>
  </define>

  <a:documentation>
    This mapping represents the morphisms used to map the interface to
    the left and the right side of the rule. Every ID in the interface
    net must have exactly one mapping in the left and right side
    (represented as 'mapElement' tags).
  </a:documentation>
  <define name="trans.mapping">
    <element name="mapping">
      <zeroOrMore>
        <ref name="trans.mapping.element"/>
      </zeroOrMore>
    </element>
  </define>

  <a:documentation>
    One 'mapElement' contains an interface ID and its mappings in the
    left and right side. This means:
    left(interfaceID) = deleteID and right(interfaceID) = insertID
  </a:documentation>
  <define name="trans.mapping.element">
    <element name="mapElement">
      <attribute name="interfaceID">
        <data type="ID"/>
      </attribute>
      <attribute name="deleteID">
        <data type="ID"/>
      </attribute>
      <attribute name="insertID">
        <data type="ID"/>
      </attribute>
```

```
    </element>
  </define>
</grammar>
```

Listing A.2: Generische Definition der Produktion

Definition der Produktion für Referenznetze

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
     datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <a:documentation>
    Rule structure definition using the Reference net definition "refPNML.rng".
    version: 1.0
    author: Jan Stueckrath
  </a:documentation>

  <include href="refPNML.rng">
    <start combine="interleave">
      <empty/>
    </start>
  </include>

  <include href="generic_rule.rng"/>

  <define name="net.definition" combine="interleave">
    <ref name="net.element"/>
  </define>

</grammar>
```

Listing A.3: Definition der Produktion für Referenznetze

B Inhalt der CD

In diesem Anhang soll der Inhalt der beigelegten CD kurz vorgestellt werden. Die Struktur der CD ist dabei wie folgt aufgebaut:

example Dieser Ordner enthält das ausführbare Beispiel aus Kapitel 4. Dieses kann durch Aufruf der Datei `nuclearPower.jar` gestartet werden, wenn Java auf dem startenden System installiert ist. Dieser Aufruf kann nicht direkt von der CD geschehen, da dieses Programm Schreibrechte für den Ordner benötigt, indem es sich befindet. Daher kann das Beispiel erst nach dem Kopieren dieses Ordners auf einen beschreibbaren Datenträger ausgeführt werden.

source code Dieser Ordner enthält den Quellcode der Implementierung aus Kapitel 3 (Unterordner `cpn`) und des Beispiels aus Kapitel 4 (Unterordner `nuclearPower`). Beide Projekte sind mit der Entwicklungsumgebung Eclipse[1] erstellt worden und enthalten dadurch die automatisch erstellten Projektdateien, sodass sie direkt mit Ecplise geöffnet werden können.

transformation Dieser Ordner enthält alle Dateien, die zur Einbindung des erstellen Graphtransformationssystems (Kapitel 3) in eine Software benötigt werden.

- **doc** Dieser Ordner enthält die Dokumentation des erstellten Graphtransformationssystems. Dazu wurde aus dem Quellcode eine JavaDoc API erstellt, die alle Klassen und ihre Methoden beschreibt, welche in der beigelegten Bibliothek (`transform.jar`) enthalten sind.
- **schema** Dieser Ordner enthält die RelaxNG Definitionsdateien der Produktionen und der Petri Netze, die als Schema bei der Validierung der XML Dokumente benötigt werden.
- **transform.jar** Diese Bibliothek kann direkt in eine Software eingebunden werden, um das erstellte Graphtransformationssystem zu nutzen. Sie enthält zusätzlich den Quellcode der implementierten Klassen.
- **README.txt** Diese README-Datei erläutert, wie die erstelle Bibliothek in eine eigene Software integriert werden kann.
- **jing.jar** Jing[2] ist eine Bibliothek zur Validierung von XML Dokumenten. Diese muss ebenfalls zur Nutzung des erstellen Graphtransformationssystems eingebunden werden.

diplomarbeit_stueckrath.pdf Dies ist die vorliegende Diplomarbeit in Form einer pdf-Datei.

index.html Diese html-Seite gibt eine Übersicht über den Inhalt der CD.

[1] http://www.eclipse.org/
[2] http://www.thaiopensource.com/relaxng/jing.html

C Produktionen der Erweiterung des Siedewasserreaktors

Dieses Kapitel führt alle vier Produktionen auf, die in Kapitel 4 zur Erweiterung der Oberfläche eines Siedewasserreaktors verwendet wurden.

Produktion zum Hinzufügen eines Notfall-Knopfes

```
<rule>
  <deleteNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
    </net>
  </deleteNet>
  <interface>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
    </net>
  </interface>
  <insertNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
      <arc id="arc1" source="place1" target="trans1">
        <inscription><text>["trans1",tuple]</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
```

```
      <transition id="trans1">
        <inscription><text>int1 = 0;
int2 = 100;
bool1 = false;</text></inscription>
      </transition>
      <arc id="arc2" source="trans1" target="place2">
        <inscription><text>int1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc3" source="trans1" target="place3">
        <inscription><text>int1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc4" source="trans1" target="place4">
        <inscription><text>int1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc5" source="trans1" target="place5">
        <inscription><text>int2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc6" source="trans1" target="place6">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc7" source="trans1" target="place7">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc8" source="trans1" target="place8">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc9" source="trans1" target="place9">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
    </net>
  </insertNet>
  <mapping>
    <mapElement interfaceID="place1" deleteID="place1" insertID="place1"/>
    <mapElement interfaceID="place2" deleteID="place2" insertID="place2"/>
    <mapElement interfaceID="place3" deleteID="place3" insertID="place3"/>
    <mapElement interfaceID="place4" deleteID="place4" insertID="place4"/>
    <mapElement interfaceID="place5" deleteID="place5" insertID="place5"/>
    <mapElement interfaceID="place6" deleteID="place6" insertID="place6"/>
    <mapElement interfaceID="place7" deleteID="place7" insertID="place7"/>
    <mapElement interfaceID="place8" deleteID="place8" insertID="place8"/>
    <mapElement interfaceID="place9" deleteID="place9" insertID="place9"/>
  </mapping>
</rule>
```

Listing C.1: Produktion zum Hinzufügen eines Notfall-Knopfes

Produktion zum Hinzufügen eines Reset-Knopfes

```
<rule>
  <deleteNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
```

```
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
    </net>
  </deleteNet>
  <interface>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
    </net>
  </interface>
  <insertNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <place id="place6"/>
      <place id="place7"/>
      <place id="place8"/>
      <place id="place9"/>
      <arc id="arc1" source="place1" target="trans1">
        <inscription><text>["trans2",[integer1,integer2]]</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <transition id="trans1">
        <inscription><text>int1 = 0;
int2 = 50;
int3 = integer1.intValue();
int4 = integer2.intValue();
bool1 = false;
bool2 = true;</text></inscription>
      </transition>
      <arc id="arc2" source="trans1" target="place2">
        <inscription><text>int3</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc3" source="trans1" target="place3">
        <inscription><text>int1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc4" source="trans1" target="place4">
        <inscription><text>int4</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc5" source="trans1" target="place5">
```

```
        <inscription><text>int2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc6" source="trans1" target="place6">
        <inscription><text>bool2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc7" source="trans1" target="place7">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc8" source="trans1" target="place8">
        <inscription><text>bool2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc9" source="trans1" target="place9">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
    </net>
  </insertNet>
  <mapping>
    <mapElement interfaceID="place1" deleteID="place1" insertID="place1"/>
    <mapElement interfaceID="place2" deleteID="place2" insertID="place2"/>
    <mapElement interfaceID="place3" deleteID="place3" insertID="place3"/>
    <mapElement interfaceID="place4" deleteID="place4" insertID="place4"/>
    <mapElement interfaceID="place5" deleteID="place5" insertID="place5"/>
    <mapElement interfaceID="place6" deleteID="place6" insertID="place6"/>
    <mapElement interfaceID="place7" deleteID="place7" insertID="place7"/>
    <mapElement interfaceID="place8" deleteID="place8" insertID="place8"/>
    <mapElement interfaceID="place9" deleteID="place9" insertID="place9"/>
  </mapping>
</rule>
```

Listing C.2: Produktion zum Hinzufügen eines Reset-Knopfes

Produktion zur Einrichtung der maximalen Kühlung

```
<rule>
  <deleteNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
    </net>
  </deleteNet>
  <interface>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
    </net>
  </interface>
  <insertNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
```

```
      <place id="place2"/>
      <place id="place3"/>
      <place id="place4"/>
      <place id="place5"/>
      <arc id="arc1" source="place1" target="trans1">
        <inscription><text>["trans3",[integer1,integer2]]</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <transition id="trans1">
        <inscription><text>int1 = integer1.intValue();
int2 = integer2.intValue();
bool1 = true;</text></inscription>
      </transition>
      <arc id="arc2" source="trans1" target="place2">
        <inscription><text>int1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc3" source="trans1" target="place3">
        <inscription><text>int2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc4" source="trans1" target="place4">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc5" source="trans1" target="place5">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
    </net>
  </insertNet>
  <mapping>
    <mapElement interfaceID="place1" deleteID="place1" insertID="place1"/>
    <mapElement interfaceID="place2" deleteID="place2" insertID="place2"/>
    <mapElement interfaceID="place3" deleteID="place3" insertID="place3"/>
    <mapElement interfaceID="place4" deleteID="place4" insertID="place4"/>
    <mapElement interfaceID="place5" deleteID="place5" insertID="place5"/>
  </mapping>
</rule>
```

Listing C.3: Produktion zur Einrichtung der maximalen Kühlung

Produktion zur Umgehung der Turbine

```
<rule>
  <deleteNet>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
    </net>
  </deleteNet>
  <interface>
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
    </net>
  </interface>
  <insertNet>
```

```
    <net id="n1" type="refNet">
      <place id="place1"/>
      <place id="place2"/>
      <place id="place3"/>
      <arc id="arc1" source="place1" target="trans1">
        <inscription><text>["trans4",tuple]</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <transition id="trans1">
        <inscription><text>bool1 = true;
bool2 = false;</text></inscription>
      </transition>
      <arc id="arc2" source="trans1" target="place2">
        <inscription><text>bool2</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
      <arc id="arc3" source="trans1" target="place3">
        <inscription><text>bool1</text></inscription>
        <type><text>ordinary</text></type>
      </arc>
    </net>
  </insertNet>
  <mapping>
    <mapElement interfaceID="place1" deleteID="place1" insertID="place1"/>
    <mapElement interfaceID="place2" deleteID="place2" insertID="place2"/>
    <mapElement interfaceID="place3" deleteID="place3" insertID="place3"/>
  </mapping>
</rule>
```

Listing C.4: Produktion zur Umgehung der Turbine

Literaturverzeichnis

[1] Jiri Adamek, Horst Herrlich, and George E. Strecker. *Abstract and concrete categories.* John Wiley and Sons, New York, 1990.

[2] Bernd Baumgarten. *Petri-Netze.* BI-Wissenschaftsverlag, Mannheim, Germany; Wien, Austria; Zürich, Switzerland, 1990.

[3] Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri net markup language: Concepts, technology, and tools. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–506. Springer-Verlag, Berlin, 2003.

[4] H. Boussairi. Petrinet-based implementation of a process model - SOM-based automatic surveillance of human-machine interaction. Master's thesis, University of Duisburg-Essen, 2008.

[5] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In Rozenberg [16], pages 163–246.

[6] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall, 3rd edition, 2003.

[7] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [16], pages 247–312.

[8] Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations of petri nets. *Electr. Notes Theor. Comput. Sci.*, 148(1):151–172, 2006.

[9] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Claudia Ermel, Ulrike Prange, Enrico Biermann, and Tony Modica. Petri net transformations. In Vedran Kordic, editor, *Petri Net, Theory and Applications*, chapter 1, pages 1–16. I-Tech Education and Publishing, 2008.

[10] Michael R. Garey and David S. Johnson. *Computers and intractability; A guide to the theory of NP-completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[11] Erik Hollnagel. *Cognitive reliability and error analysis method (CREAM).* Elsevier Science, 1998.

[12] K. Jensen. *Coloured petri nets. Basic concepts, analysis methods and practical use*, volume 1: Basic Concepts of *EATCS Monographs in Theoretical Computer Science.* Springer, Berlin, 2nd edition, 1996.

[13] Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.

[14] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew-user guide, 2001. http://www.renew.de/.

[15] Donald A. Norman. *The psychology of everyday things*. Basic Books, New York, 1988.

[16] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, Volume 1: Foundations*. World Scientific, 1997.

[17] Andy Schürr and Bernhard Westfechel. Graph grammars and graph rewriting systems. Skript, Mai 1992.

[18] M. Sipser. *Introduction to the theory of computation*. Thomson Course Technology, 2005.

[19] Benjamin Weyers and Wolfram Luther. Formal reconfiguration and redesign of human-computer interfaces. In *Interface and Interaction Design for Learning and Simulation Environments, Readings of the Summer Academy, Duisburg 2009*, pages 67–79. Logos Verlag, 2009.

Versicherung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt oder veröffentlicht.

Universität Duisburg-Essen, 30. März 2010 Jan Stückrath