Entwicklung eines Tools zur Visualisierung von Simulationsabläufen mittels Petrinetzen

Projektarbeit

im Studiengang
Maschinenbau (M.SC.) - IT in Produktion und Logistik

vorgelegt von

Dominik Srednicki Matr.-Nr.: 175228

Damian Srednicki Matr.-Nr.: 175229

an der Technischen Universität Dortmund

I. Abbildungsverzeichnis

1. Einleitung und Motivation

Die Simulation ist ein etabliertes Verfahren zur Analyse und Bewertung von Produktionsprozessen. Um diese nicht nur anhand von Ergebnistabellen verifizieren und validieren zu können, bietet es sich an, die Abläufe visuell darzustellen. Dies ermöglicht, ausgewählte Modellabschnitte bzw. Simulationszeitpunkte detaillierter zu untersuchen.

Ziel dieser Arbeit ist die Entwicklung eines Tools zur Visualisierung von Simulationsabläufen mittels Petri-Netzen. Als Petri-Netz-Klasse werden Timed Hierachical Object-related Nets (THORNs) gewählt, die ereignisdiskret simuliert werden. Für eine effiziente ereignisdiskrete Simulation ist auf Basis der vorliegenden Erkenntnisse des ITPL zu ermitteln, wann Ereignisse zu erzeugen und auszuführen sind, ohne bedingte Ereignisse zu verwenden. Die Netzstruktur, welche für die Simulation in das zu entwickelnde Tool einzulesen ist, wird mit Hilfe des yED Graph Editors erstellt. Das Tool ist auf Basis vorhandener Bibliotheken mit C++ und QT zu entwerfen.

Im Einzelnen sind folgende Aufgaben zu bearbeiten:

- Aufbereitung der strukturellen und logischen Elemente von THORNs
- Aufbereitung der ereignisdiskreten Simulation für Petri-Netze
- Erarbeiten der notwendigen Ereignisse und deren Ausführungsreihenfolge für die Simulation
- Laden und Visualisieren eines im yED Graph Editors entworfenen THORNs in das zu erstellende Tool
- Simulation des THORNs im erstellten Tool

Ziele

Das Ziel der Arbeit ist die Entwicklung eines Tools zur Visualisierung von Simulationsabläufen mittels Petri-Netzen. Durch den sehr hohen Arbeitsaufwand der verschiedenen Funktionen, wird die Entwicklung in Unterpunkte aufgeteilt, die als Meilensteine definiert werden (vergl. Abschnitt 3.2 Projektplan).

Folgende Ziele werden definiert:

- Einlesen, Extrahieren und Auswerten von Petri-Netzen (XML)
- Visualisieren von Petri-Netzen (Stellen, Transitionen, Kanten)
- Aufbau einer einfachen Grafischen Simulation
- Implementieren einer Benutzeroberfläche

2. Petri-Netz

Petri-Netze, auch bekannt als Stellen-/Transitions-Netz, sind diskrete Modelle zur Modellierung und Analyse von Systemen mit nebenläufigen und parallelen Prozessen. Ein Petri-Netz ist ein gerichteter bipartiter Graph, welcher zwei Knotentypen besitzt. Diese werden Stellen und Transitionen genannt und sind durch Kanten verbunden.

Stellen enthalten Marken und durch das Schalten von Transitionen werden diese abgezogen oder hinzugefügt. Dabei legt die Kante sowohl den Weg als auch die Anzahl der abzuziehenden bzw. hinzuzufügenden Marken durch das Kantengewicht fest.

Petri-Netze werden nicht nur in der Informatik eingesetzt, sondern in vielen weiteren Fachgebieten, wie dem Maschinenbau, der Logistik oder Biologie.

2.1 Definition Petri-Netz

Ein Petri-Netz N = (S, T, K) besteht aus zwei disjunkten Mengen S und T sowie der Beziehung K ⊆ (S ✖ T) ∪ (T ✖ P). Dabei gilt:

- Stelle S = {s1,s2,s3,..s| S |}, nicht leere Menge von Stellen
- Transition T = {t1,t2,t3,…,t| T |}, nicht leere Menge von Transitionen
- Kanten K ⊆ (S ✖ T) ∪ (T ✖ P), nicht leere Menge von Kanten

Vor- und Nachbereich

Es gelte: Petri-Netz N = (S, T, K) mit einem Element x ∈ S T von N.

Vorbereich

Die Menge ●x = {z ∈ S T | z K x} heißt Vorbereich von **x**. Die Menge ●x ist die Anzahl aller Knoten, von denen eine Kante zum Knoten **x** führt.

Nachbereich

Die Menge x●= {z ∈ S T | x K z} heißt Nachbereich von x. Die Menge x● ist die Anzahl aller Knoten, zu denen vom Knoten **x** aus eine Kante führt.

Beispiel: Vor- und Nachbereich

Damit eine Transition schalten kann, müssen alle Vorbedingungen erfüllt sein. Beim Schalten der Transition t1 wird aus den Stellen s1 und s2 des Vorbereichs (Abbildung 1) genau die Anzahl an Marken entnommen, welche dem Kantengewicht entsprechen. In diesem Beispiel ist explizit kein Kantengewicht angegeben, es beträgt eins. Es wird also jeweils eine Marke aus den Stellen s1 und s2 des Vorbereichs abgezogen.

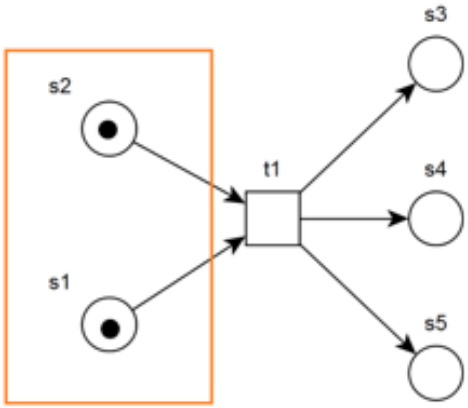


Abbildung 1: Petri-Netz-Vorbereich

Sind alle Bedingungen des Vorbereichs erfüllt und schaltet die Transition, werden alle Marken aus dem Vorbereich der Stellen s1 und s2 entnommen. Im Nachbereich der Transition t1 wird jeweils eine Marke erzeugt (Kantengewicht beträgt eins) und in die Stellen s3, s4 sowie s5 (Abbildung 2) eingefügt. Die Nachbedingungen sind somit Vorbedingungen für nachfolgende Ereignisse.

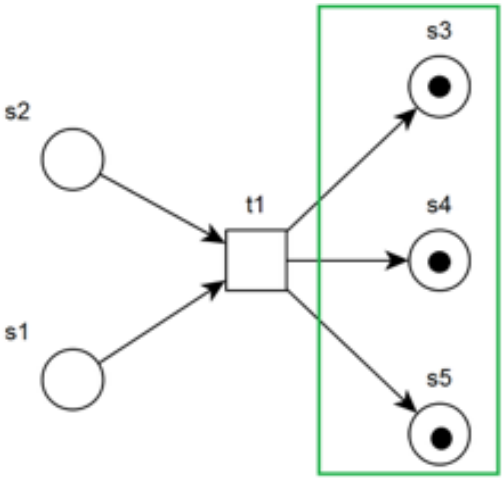


Abbildung 2: Petri-Netz -Nachbereich

2.3 Modellierungssprache Thor-Netze

Zur Modellierung von höheren Petri-Netzen wurde im Rahmen des DNS (Distributed Net Simulation) Projekts eine spezielle Klasse entwickelt, die sogenannte Timed Hierachical Object-Related Nets – kurz Thorn genannt. Diese sind höhere Petri-Netze und erlauben die Modellbildung komplexer Systeme. Zu den Charakteristiken zählen:

- Komplexe Objekte lassen sich wie in einer objektorientieren Programmiersprache (C++, Java) verwenden.
- Stellen können die Struktur Multimerge, Stack, Queue oder Priority Queue annehmen. Die Verwaltung der Stellen erfolgt dabei nach dem bekannten Zugriffsverhalten. Stellen lassen sich zudem durch eine Kapazität beschränken.
- Das Schaltverhalten der Transitionen lässt sich durch Standardkanten, aktivierende, inhibitorische und konsumierende Katen steuern.
- Transitionen erhalten eine Schaltbedingung, Schaltkapazität und Schaltaktion. Die Schaltkapazität gibt an, wie oft eine Transition parallel zu sich selbst schalten kann.
- Um zeitliche Abläufe darzustellen, können Transition zwei Funktionen zur Bestimmung einer Verzögerungszeit und einer Schaltdauer beschreiben. Die Schaltdauer gibt an, wie lange das Schalten einer Transition dauert. Die Verzögerungszeit gibt an, wie lange die Transition aktiv sein muss, um zu schalten.

2.4 Stellen

Stellen sind passive Komponenten und werden mit einem Namen, Typ und ihrer Kapazität beschriftet. Stellen können Dinge speichern, lagern oder Zustände anzeigen. Durch den Typ wird festgelegt, welchen Typ Objekte haben dürfen, die auf der Stelle abgelegt sind. Die Kapazität definiert die maximale Anzahl (auch unbegrenzt möglich) an Objekten, die sich gleichzeitig auf ihr befinden dürfen. Ein Beispiel hierfür wäre die Modellierung einer beschränkten Lagerkapazität. Stellen müssen in Thor-Netzen mit einer Struktur ausgestattet werden, diese definiert die Reihenfolge der Objekte auf einer Stelle. In Thor-Netzen gibt es vier unterschiedliche Stellenstrukturen (Abbildung 3).

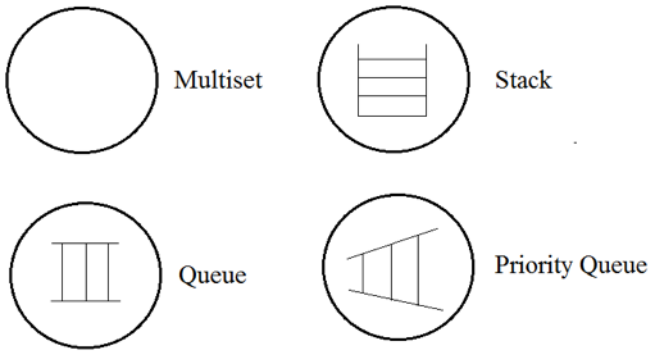


Abbildung 3: Thorn-Netz-Stellen

Multiset-Stellen

Multiset-Stellen haben keine besondere Struktur. Objekte werden als Mengen verwaltet. Das Hinzufügen und Abziehen von Objekten ist beliebig möglich. Durch Referenzen werden diese eindeutig bestimmt.

Queue-Stellen

Objekte werden nach dem FIFO-Prinzip abgezogen. Das Objekt, welches als erstes auf die Stelle hinzugefügt wurde, wird auch wieder zuerst abgezogen.

Stack-Stellen

Objekte werden nach dem LIFO-Prinzip abgezogen. Das Objekt, welches als letztes auf die Stelle hinzugefügt wurde, wird als erstes abgezogen.

Priority-Queue-Stellen

Der Programmierer/Modellierer legt selber fest, welche Priorität die Objekte haben. Das Objekt mit der höchsten Priorität wird als erstes abgezogen.

2.5 Kanten

Kanten haben einen Namen und Kantentyp. Thorn bietet vier Kantentypen (Abbildung 4). Mit diesen lässt sich das Schaltverhalten der Transitionen beeinflussen. Zusätzlich ist es möglich, Kantentypen, Kantengewichte und Variablennamen anzugeben.

Standard-Kanten

Standard-Kanten werden mit einem Kantengewicht angegeben und können zusätzlich mit einem Variablennamen beschriftet sein. Das Kantengewicht einer Standard-Kante definiert, wie viele Marken auf der Stelle liegen müssen, um die Transition zu aktivieren oder die Anzahl an Marken, die erzeugt werden sollen.

Enabling-Kanten

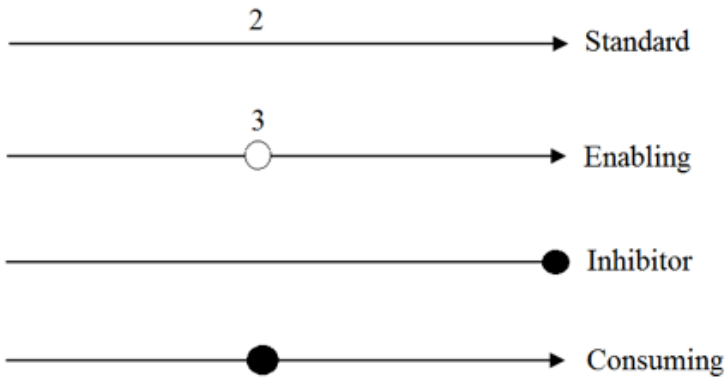


Abbildung 4: Thorn-Netz-Kantentypen

Die Beschriftung der Enabling-Kanten ist mit den Standard-Kanten identisch. Anders als bei diesen werden beim Schalten einer Transition keine Marken über die Kanten konsumiert. Weil beim Schalten der Transition keine Objekte verändert werden dürfen, sind diese auch als Lese- oder Testkanten bekannt.

Inhibitor-Kanten

Eine Transition wird deaktiviert, wenn die Kante von der Stelle weg- oder hinführt.

Consuming-Kanten

Beim Schalten der Transition werden alle Marken benachbarter Stellen gelöscht. Consuming-Kanten haben keine Auswirkungen auf das Schalten einer Transition.

2.6 Transitionen

Transitionen sind eine aktive Komponente, die mit einem Namen, einer Schaltkapazität, Schaltbedingung sowie Schaltaktion beschriftet werden. Sie können Objekte erzeugen, transportieren, verändern und löschen. Transitionen dienen durch das Schalten dazu, Objekte von Stellen abzuziehen oder hinzuzufügen. Schaltbedingungen ermöglichen zusätzliche Bedingungen der abzuziehenden Marken im Vorbereich. Durch die Schaltaktion legt man das Verhalten der Transition beim Schalten fest. Die Schaltkapazität bestimmt, wie oft eine Transition parallel zu sich selbst schalten kann.

3. Vorbereitung

3.1 Entwicklungsumgebung

Für die Entwicklung des Projektes „Entwicklung eines Tools zur Visualisierung von Simulationsabläufen mittels Petri-Netzen“ wurden als Entwicklungsumgebung QT-Creator 3.3.2 (opensource) und die objektorientierte Programmiersprache C++ verwendet.

Um Abhängigkeiten zu verhindern und zur einfacheren Weiterentwicklung wurde auf zusätzliche Frameworks verzichtet. Zum Erstellen der Petri-Netze kam als Modellierungs-Werkzeug die Software yED Graph Editor 3.14 zum Einsatz.

3.2 Projektplan

Zur besseren Planung des Projekts wurde ein grober Projektplan erstellt. Wie bei jeder Software-Entwicklung verzögern und beschleunigen sich Entwicklungsprozesse. Die Anlage 1: QT-Projekt Roadmap (Vorschau: Abbildung 5) zeigt den optimalen Verlauf vor dem eigentlichen Entwicklungsstart. Während der gesamten Programmierungs- / Entwicklungsphase wurden Funktionen parallel erarbeitet und anschließend ins Hauptprogramm zusammengefasst, um eine fehlerfreie Funktionsweise der Software sicherzustellen. Die Tests erfolgten dabei stets analog zur Entwicklung.

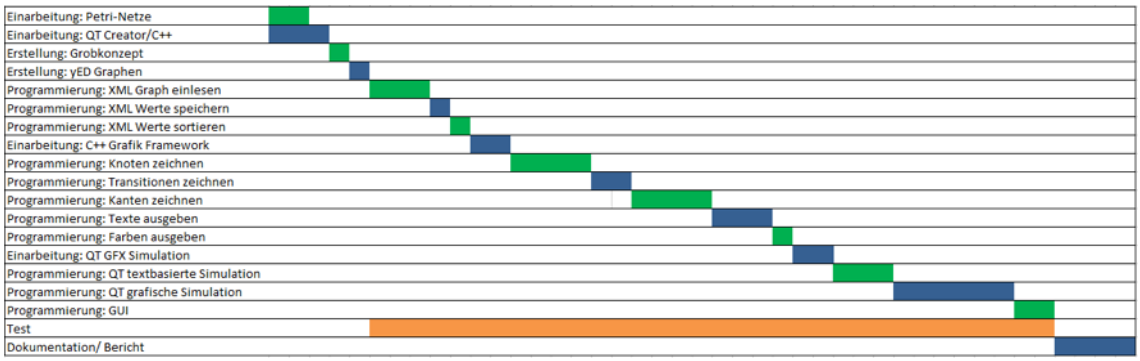


Abbildung 5: QT-Projekt Roadmap

4. XML

XML steht für Extensible Markup Language und ist eine plattformunabhängige erweiterbare Auszeichnungssprache, um hierarchisch strukturierte Daten, wie Tabellen, darzustellen. Um XML für den Datenaustausch zu verwenden, sollte es gültig sein. Damit ein XML-Dokument als gültig gilt, muss es wohlgeformt sein und eine XML-Deklaration (z. B. `<?xml version="1.0"?>`) sowie eine Dokumenttyp-Deklaration enthalten. Sind alle XML-Regeln eingehalten, gilt das XML-Dokument als wohlgeformt.

4.1 XML Regeln

Damit ein XML als wohlgeformt gilt, müssen nachfolgende XML-Regeln eingehalten werden:

- Es existiert mindestens ein Datenelement (z. B. `<Element1>…</Element1>`)
- Es gibt genau ein äußerstes Element, welches alle anderen Datenelemente enthält
- Alle Elemente besitzen einen Beginn-Tag (`<tag>`) und End-Tag (`</tag>`)
- Elemente mit mehreren Attributen mit identischen Namen sind unzulässig
- Attributeigenschaften müssen in Anführungszeichen („“) stehen
- Groß- und Kleinschreibung bei Beginn- und Endtags sind nicht erlaubt

4.2 Parameter speichern

Um erforderliche Parameter für die Simulation und Visualisierung zu ermitteln, wurden ein Test-Petri-Netz-Graph erstellt und die XML-Elemente analysiert. Die Tabelle 1 zeigt einen Ausschnitt der extrahierten Werte.

Tabelle 1: XML-Test-Elemente

Element	XML Attribut
Element - Typ	name = graphics -> key=type
Element - X Position	name = graphics -> key=x
Element - Y Position	name = graphics -> key=y
Element - Breite	name = graphics -> key=w
Element - Höhe	name = graphics -> key=h
Element - Farbe	name = graphics -> key=fill
Element - Text	name = LabelGraphics -> key=text
Kante - Quelle	name=edge -> key=source
Kante - Ziel	name=edge -> key=target
Kante - Text	name=edge -> key=text

4.3 XML-Schnittstelle

Für den Zugriff auf XML-Dokumente gibt es unterschiedlich spezifizierte Schnittstellen. Zu den etabliertesten zählen DOM, SAX und XPATH.

4.3.1 DOM

DOM ist die Abkürzung für Document Object Model. Hierbei handelt es sich um eine Schnittstelle für den Zugriff auf Auszeichnungssprachen (XML oder HTML), welcher vom W3C (World Wide Web Consortium) definiert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<auto>
  <mercedes>
  <limousine>
    <name>E</name>
    <name>S</name>
  </limousine>
  <sport>
    <name>AMG GT</name>
  </sport>
  </mercedes>
. . .
. . .
</auto>
```

Abbildung 6: Beispiel XML-Datei: auto.xml

Abbildung 6 zeigt den Inhalt eines XML-Dokumentes. DOM repräsentiert ein XML-Dokument als Baumstruktur, welches den Aufbau und die Beziehungen der einzelnen XML-Elemente zeigt (Abbildung 7).

Mit HTML DOM ist es außerdem möglich, auf HTML-Dokumente zuzugreifen. Hier erfolgt die Wiedergabe ebenfalls als Baumstruktur.

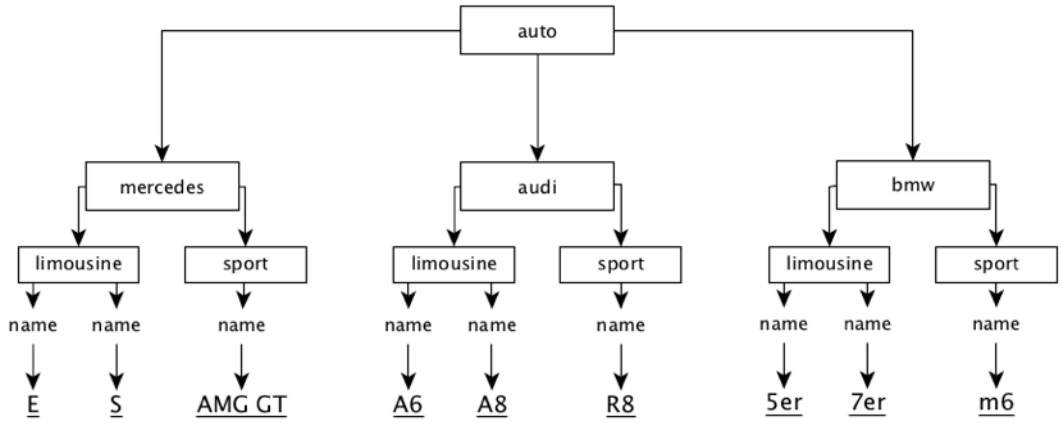


Abbildung 7: DOM-Baumstruktur: auto.xml

4.3.2 XPath

XML Path language (XPath) ist eine Abfragesprache und dient dazu, Knoten in XML-Dokumenten zu adressieren. Zum Adressieren von XML-Elementen und Attributen wird ein Pfad angelegt. Dieser Pfad enthält alle übergeordneten- und die zu adressierenden Elemente. Die Trennung der Elemente erfolgt dabei mit einem „/“. Zusätzlich lässt sich ein Pfad weiter einschränken, indem man Prädikate, welche in eckigen Klammern ([ ]) eingeschlossen werden, verwendet. Attribute werden mit einen „@“ angesprochen. Das Resultat einer XPath-Abfrage gibt alle ermittelten Werte als Element zurück.

Ein Beispiel für eine XPath-Abfrage zeigt der folgende Programmcode:

```
"section/section/section[@name='node']/section/attribute[@key='x']"
```

Das Ergebnis der beispielhaften XPath-Abfrage zeigt Abbildung 8.

```
Element='<attribute key="x" type="double">510.0</attribute>'
Element='<attribute key="x" type="double">510.0</attribute>'
Element='<attribute key="x" type="double">510.0</attribute>'
Element='<attribute key="x" type="double">510.0</attribute>'
Element='<attribute key="x" type="double">360.0</attribute>'
Element='<attribute key="x" type="double">630.0</attribute>'
Element='<attribute key="x" type="double">630.0</attribute>'
Element='<attribute key="x" type="double">630.0</attribute>'
Element='<attribute key="x" type="double">510.0</attribute>'
Element='<attribute key="x" type="double">510.0</attribute>'
```

Abbildung 8: XPath Ausgabe

4.3.3 SAX (Simple API for XML)

Simple API for XML (SAX) ist ein Quasi-Standard. Wie DOM bietet SAX ebenfalls die Möglichkeit, auf XML-Dokumente zuzugreifen. Im Gegensatz zu DOM erfolgt die Repräsentation nicht als Baumstruktur, sondern das Dokument wird sequenziell verarbeitet.

SAX liest ein XML-Dokument seriell, bei einem erkannten Element wird jeweils ein neues Ereignis ausgelöst. Dabei können Ereignisse durch Anweisungen, Zeichendaten, Element-Tags oder die Grenzen des Dokumentes ausgelöst werden. Tabelle 2 zeigt SAX-Ereignisse beim Analysieren des Datenstroms („parsen“) eines XML-Dokumentes.

Tabelle 2: SAX Ereignisse

XML Tag		SAX-Ereignis
	←	startDocument
<section>	←	startElement: section
<attribut>	←	startElement: attribut
123	←	characters: 123
</attribut>	←	endElement: attribut
</section>	←	endElement: section
	←	endDocument

4.4 XML Öffnen und Lesen

Die Abbildung 9 zeigt, für einen besseren Überblick, den kompletten Algorithmus zum Öffnen und Lesen eines XML-Dokumentes. Nachfolgend wird der Algorithmus abschnittsweise beschrieben.

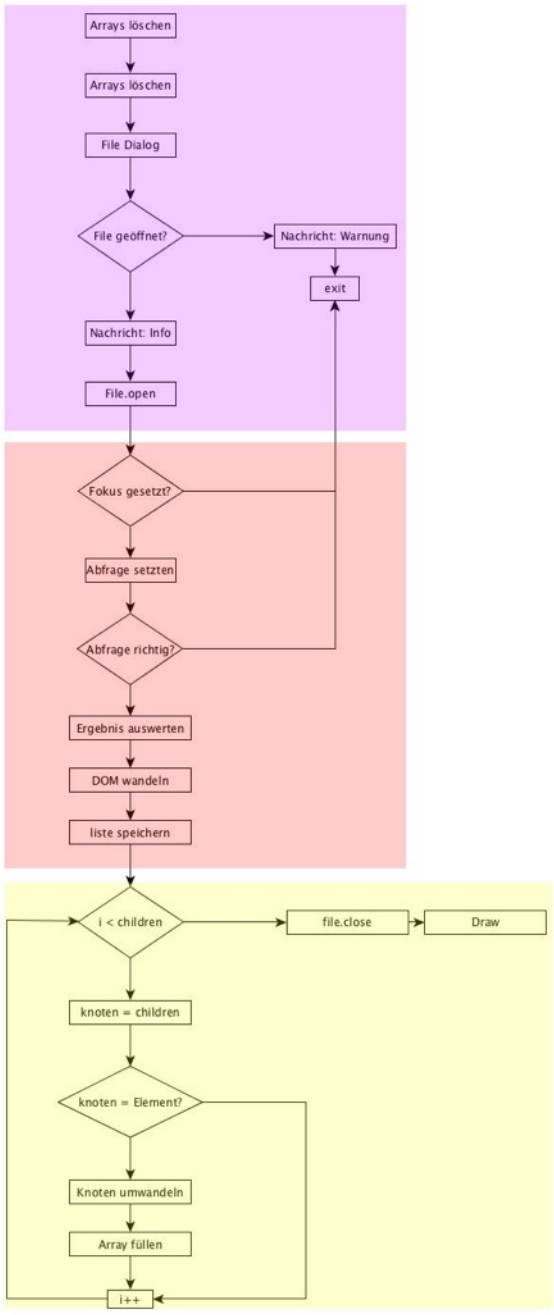


Abbildung 9: Öffnen und Lesen des Algorithmus

4.4.1 XML-Dokument öffnen

```
#include <QMessageBox>
#include <QFile>
#include <QFileDialog>
```

Um XML-Dokumente öffnen zu können, wurde eine Dateiverwaltung implementiert, die es ermöglicht, Dateien in einer „Dialogbox“ auszuwählen und zu öffnen. Für die Dateiverwaltung werden die Klassen QFile, QFileDialog und QMessageBox benötigt.

Bei Funktionsaufruf wird anfänglich der Inhalt der Arrays, welche beschrieben werden sollen, gelöscht.

```
NewXarray.clear();
NewYarray.clear();
NewWarray.clear();
. . . .
```

Das Einfügen neuer Elemente in die Arrays erfolgt mit der Funktion "push_back()". Die übergebenen Werte werden an das Ende des Arrays angefügt. Die Länge steigt somit um die Anzahl der angefügten Werte. Um zu verhindern, dass die Arrays unkontrolliert länger werden sowie die richtige Visualisierung der erstellten Petri-Netze mit dem yED Graph Editor zu gewährleisten, müssen diese bei jedem Funktionsaufruf bereinigt werden.

```
_filename = QFileDialog::getOpenFileName(this, tr("Open XML
File"), "/home", tr("XGML Files (*.xml *.xgml);;All Files (*)"));
```

Im Anschluss, wird der Dateiname durch die Hilfsfunktion `QFileDialog::getOpenFileName()` abgefragt.

Die Funktion dient dazu, einen Dialog zu öffnen, welche es dem Anwender ermöglicht, eine Datei auszuwählen. Beim Übergabewert handelt es sich um einen String, der den kompletten Dateipfad enthält oder um einen leeren String im Falle eines Abbruchs. Der String wird in die Variable _filename gespeichert. Tabelle 3 zeigt die Funktion zur individuellen Steuerung und Darstellung der Dialogbox.

Tabelle 3: Argumente

Argument	Funktion
tr("Open XML File")	Überschrift der Dialogbox
"/home"	Startverzeichnis in dem gesucht wird
tr("XGML Files (*.xml *.xgml);;All Files (*)")	Auswahlbeschränkung für die Dateiauswahl

Versucht der Anwender eine Datei zu öffnen, wird zu Beginn geprüft, ob dieser tatsächlich eine Datei ausgewählt hat (Programmzeile 1). Ist dies der Fall, erscheint ein Informationsfenster mit dem Inhalt des gewählten Dateipfades und Dateinamen, das Dokument wird für den weiteren Gebrauch geöffnet sowie mit ReadOnly markiert. ReadOnly schützt das Dokument vor unzulässigen Manipulationen (Programmzeile 1 – 2).

```
1 if(!file.open(QFile::ReadOnly)){
2 QMessageBox::warning(this, tr("Filename"), _filename);}
3 else{
4 QMessageBox::information(this, tr("Error"), tr("Bitte wählen Sie
  eine Datei aus"));
```

Hat der Anwender keine Datei ausgewählt, erscheint ein Fehlerdialog, zuvor möglicherweise geöffnete Dateien werden geschlossen und das Programm wird beendet (Programmzeile 4 – 6).

```
5 file.close();
6 QCoreApplication::exit();}
```

4.4.2 Abfrage

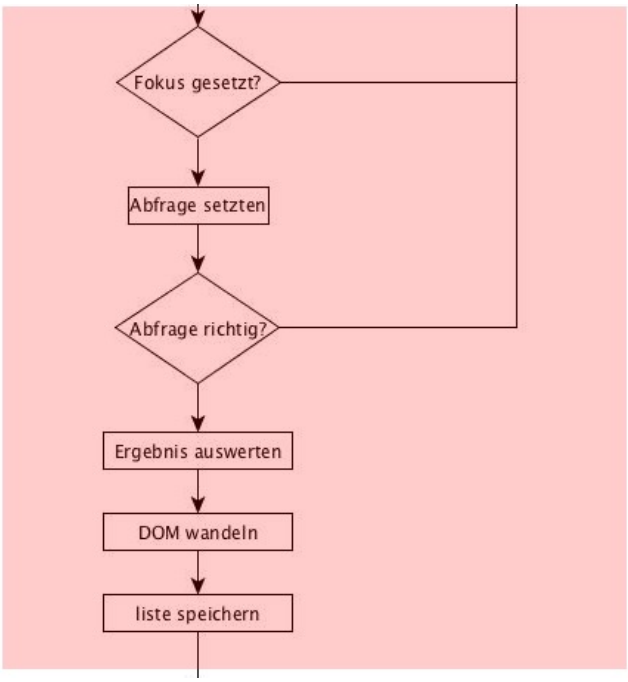


Abbildung 10: Programmablauf nach dem Öffnen

Abbildung 10 visualisiert den weiteren Programmablauf nach dem Öffnen der ausgewählten Datei. Der Fokus muss dabei auf das Dokument gelegt werden, um Abfragen im Dokument zu ermöglichen (Programmzeile 1).

```
1 query.setQuery(QString());
```

Die Abfrage erfolgt dabei durch einen String, welcher die XPath Syntax beinhaltet.

Die XPath-Abfrage (Programmzeile 2) liefert als Ergebnis die Sektionen mit dem Namen „edge“ und den Attribut mit den Key „x“.

```
2 query.setQuery(QString("section/section/section[@name='edge']/
  section/section/section/attribute[@key='x']"));
```

Um die Funktionstüchtigkeit der Software zu gewährleisten, muss die Abfrage vor deren Zwischenspeicherung auf Gültigkeit geprüft werden. Dies übernimmt die Programmzeile 3. Ist das Resultat negativ, erfolgt keine Zwischenspeicherung.

```
3 if (query.isValid())
```

Besteht die Abfrage die Gültigkeitsprüfung, kann das Ergebnis ausgewertet und zwischengespeichert werden (Programmzeile 5).

```
4 QString result;
5 query.evaluateTo(&result);
```

Im Anschluss wird ein „QDomDocument“ erstellt. Programmzeile 7 beschreibt die Umwandlung der gültigen XPath-Abfrage in eine DOM-Struktur. Diese kann anschließend in ein List-Objekt gespeichert werden, welches alle Knoten beinhaltet (Programmzeile 8).

```
6 QDomDocument dom;
7 dom.setContent(QString("<Data>%1</Data>").arg(result));
8 QDomNodeList children = dom.documentElement().childNodes();
```

4.4.3 Werte speichern

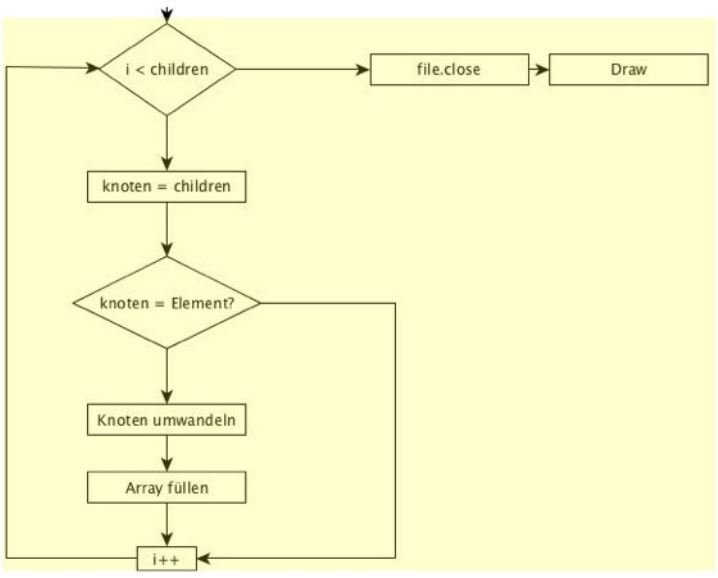


Abbildung 11: Programmablauf Werte speichern

Abbildung 11 zeigt den Programmablauf für das Speichern der Werte.

Nachdem alle Knoten sich in der vordefinierten Liste befinden, können diese nun mit einer Schleife, wie Programmzeile 1 zeigt, durchlaufen werden. Dies geschieht solange, bis alle Werte die sich in der vordefinierten Liste befinden, gespeichert sind.

```
1 for (int i = 0; i < children.count(); i++)
```

Die XPath-Abfrage speichert aber nicht nur relevante Elemente, sondern die kompletten Zeilen des XML-Dokumentes inkl. Tags (Abbildung 7). Für eine richtige Wiedergabe des erstellten Petri-Netz-Modells benötigt man allerdings nur die Werte ohne Tags. Damit dies realisiert werden kann, muss der Datentyp der Liste konvertiert werden (Programmzeile 2). Anschließend können die einzelnen XML Zeilen in ein QDomElement umgewandelt werden (Programmzeile 3 – 4). Die QDomElemente beinhalten den einzelnen Wert und können in ein Array abgespeichert werden (Programmzeile 5).

```
2 QDomNode nod = children.at(i);
3 if(nod.isElement()){
4 QDomElement art = nod.toElement();
5 lineXarray.push_back(art.text());}
```

Abschließend wird das Dokument geschlossen. Dies ist zwingend notwendig, da alle weiteren Werte des XML-Dokumentes mit der identischen Methodik und Variablen (ausgeschlossen die Arrays, welche die Werte speichern) gespeichert werden.

Die Funktion QFile::close() ruft dabei QFile::flush() auf, welches alle zwischengespeicherten Daten löscht. Nach erfolgreichen Löschen der Daten wird die Datei geschlossen (Programmzeile 6).

```
6 file.close();
```

Die Funktion ermöglicht nun die Begutachtung der gespeicherten Parameter in der Entwicklungskonsole (Abbildung 12).

```
2dArray :  ("0", "1", "2", "3", "4", "5", "6", "7", "8", "9")
NewXArray :  ("510.0", "510.0", "510.0", "510.0", "360.0", "630.0", "630.0", "630.0", "510.0", "510.0")
NewYArray :  ("90.0", "150.0", "210.0", "270.0", "360.0", "360.0", "430.5", "501.0", "570.0", "660.0")
NewWArray :  ("30.0", "60.0", "30.0", "60.0", "30.0", "30.0", "60.0", "30.0", "60.0", "30.0")
NewHArray :  ("30.0", "15.0", "30.0", "15.0", "30.0", "30.0", "15.0", "30.0", "15.0", "30.0")
NewTypeArray :  ("ellipse", "rectangle", "ellipse", "rectangle", "ellipse", "ellipse", "rectangle", "ellipse", "rectangle",
"ellipse")
NewFillArray : ("#FFFFFF", "#FFCC99", "#FFFFFF", "#FFCC99", "#FFFFFF", "#FFFFFF", "#FFCC99", "#FFFFFF", "#FFCC99", "#FFFFFF")
NewTextArray : ("3", "1", "1", "0", "1", "0")
NewIDArray : ("0", "1", "2", "3", "4", "5", "6", "7", "8", "9")
LineXArray :  ("510.0", "360.0", "360.0", "510.0", "630.0", "630.0", "630.0", "510.0", "510.0", "360.0", "360.0", "510.0")
LineYArray :  ("270.0", "270.0", "360.0", "270.0", "270.0", "360.0", "501.0", "501.0", "570.0", "360.0", "570.0", "570.0")
SourceArray :  ("0", "1", "2", "3", "3", "5", "6", "7", "4", "8")
targetArray :  ("1", "2", "3", "4", "5", "6", "7", "8", "8", "9")
EdgeWeight  :  ("2", "2", "3", "1", "2", "2", "2", "2", "1", "3")
elip:  QVector("0", "2", "4", "5", "7", "9")
sorCounter:  QVector(1, 1, 1, 2, 1, 1, 1, 1, 1, 0)
tarCounter:  QVector(0, 1, 1, 1, 1, 1, 1, 1, 2, 1)
Text : QVector("3", "0", "1", "0", "1", "0", "0", "1", "0", "0")
Elip ID :  QVector("0", "2", "4", "5", "7", "9")
rec ID :  QVector("1", "3", "6", "8")
recX :  QVector("510.0", "510.0", "630.0", "510.0")
eliupX :  QVector("510.0", "510.0", "360.0", "630.0", "630.0", "510.0")
recW :  QVector("60.0", "60.0", "60.0", "60.0")
MinTrans :  1
MaxTrans :  8
```

Abbildung 12: Konsolen-Output

5. Visualisierung

5.1 Graphics View Framework

Das Graphics View Framework stellt Funktionen für die Erstellung von 2D-Grafik-Elementen (QGraphicsItem) zur Interaktion, wie bspw. Rotieren oder Vergrößern (Zoom) sowie die Verwaltung (Scene) dieser zu Verfügung.

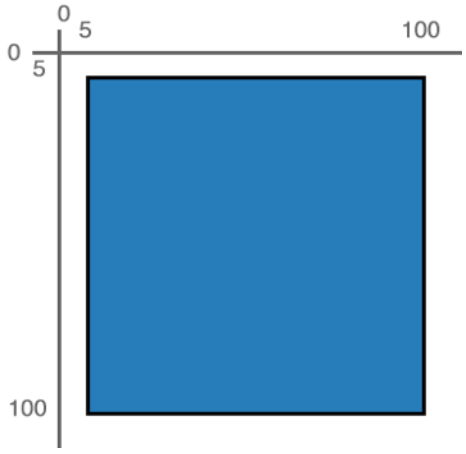
5.1.1 Graphics View Koordinatensystem

Das Koordinatensystem des Graphic View Framework basiert auf dem kartesischen Koordinatensystem. Im zweidimensionalen Raum werden die Geometrie und die Position der Objekte in der Szene mit X- und Y-Koordinaten repräsentiert.

5.1.2 QGraphicsItem

Die QGraphicsItem Klasse stellt mehrere Standard-Objekte für typische Formen wie Rechtecke (QGraphicsRecItem), Ellipsen (QGraphicsEllipseItem) und Texte (QGraphicsTextItem) zur Verfügung. Jedes Objekt benötigt für die exakte Darstellung jeweils zwei X- (x1, x2) und Y- (y1, y2) Eigenschaften. X2 repräsentiert die Breite (w) und y2 die Höhe (h) des zu erstellenden Objektes. Nicht alle Objekte haben aber Breiten- oder Höhenangaben, aus diesem Grund werden diese nur mit x2 und y2 bezeichnet, um alle Objekte darstellen zu können. Weitere Objektdefinitionen sind Linientypen (QPen), mit denen sich die Liniendicke und Farbe spezifizieren lässt, sowie die Füllfarbe (QBrush).

Tabelle 4: Objekterstellung

Programmcode	Ausgabe
`new scene->addRect(5,5,100,100, QPen(Qt::black), QBrush(Qt::blue));`	

Um wie in Tabelle 4 ein blaues Rechteck erstellen zu können, muss eingangs eine neue Szene erstellt werden. Ein Rechteck ist deklariert mit den Koordinaten (X1, X2, Y1, Y2), den Linientypen (QPen) sowie der Füllfarbe (QBrush). Für die Wiedergabe werden den Variablen Werte zugewiesen.

5.1.3 QGraphicsScene

Die Klasse QGraphicsScene ist ein Container für alle QGraphicsItems und wird durch drei Ebenen angesprochen. Das erfolgt durch den Hintergrund (background), eine Zeichenebene (items) und eine übergeordnete Ebene (foreground). Es ermöglicht, mehrere Objekte auf einer Szene zu zeichnen, ohne diese zu überschreiben.

5.1.4 QGraphicsScene – Sortieren

Für ein optimales Erscheinungsbild nach dem Vorbild eines in yED Graph Editors erstellten Petri-Netzes ist es von entscheidender Bedeutung, die Reihenfolge der zu erstellenden grafischen Elemente auf der Zeichenebene zu beachten. Wie in Abbildung 13 (links) zu sehen, überschneiden sich die grafischen Objekte. Die Zeichenabfolge ist suboptimal. Um ein besseres Ergebnis, wie in Abbildung 13 (rechts) zu erzielen, muss die Abfolge nach dem folgenden Schema verändert werden.

Abbildung 13: Zeichenabfolge Falsch (links) und Korrekt (rechts)

```
Kanten (Linien) -> Kanten (Pfeile) -> Knoten/ Transitionen
```

5.1.5 QGraphicsView

Das Widgets QGraphicsView erlaubt es, die komplette oder Teile einer Szene im UI darzustellen. Eingesetzt und bearbeitet wird das Widget im QT Creator mithilfe des Designers. Außerdem ist es möglich, das QGraphicsView über Argumente zusätzlich zu konfigurieren, indem man dieser Funktionen zuweist. Das ermöglicht den Einsatz von Kantenglättung (Programmzeile 2), verschiedener Hintergründe für die Zeichenflächen (Programmzeile 3) oder das Ausrichten der Szene (Programmzeile 4 – 5).

```
1 ui->graphicsView->setScene(scene);

2 ui->graphicsView->setRenderHint(QPainter::Antialiasing);

3 ui->graphicsView->setBackgroundBrush(QBrush(Qt::black, Qt::Den-
  se7Pattern));;

4 ui->graphicsView->viewport()->contentsRect();

5 ui->graphicsView->setAlignment(Qt::AlignTop | Qt::AlignLeft);
```

5.2 Graphen visualisieren

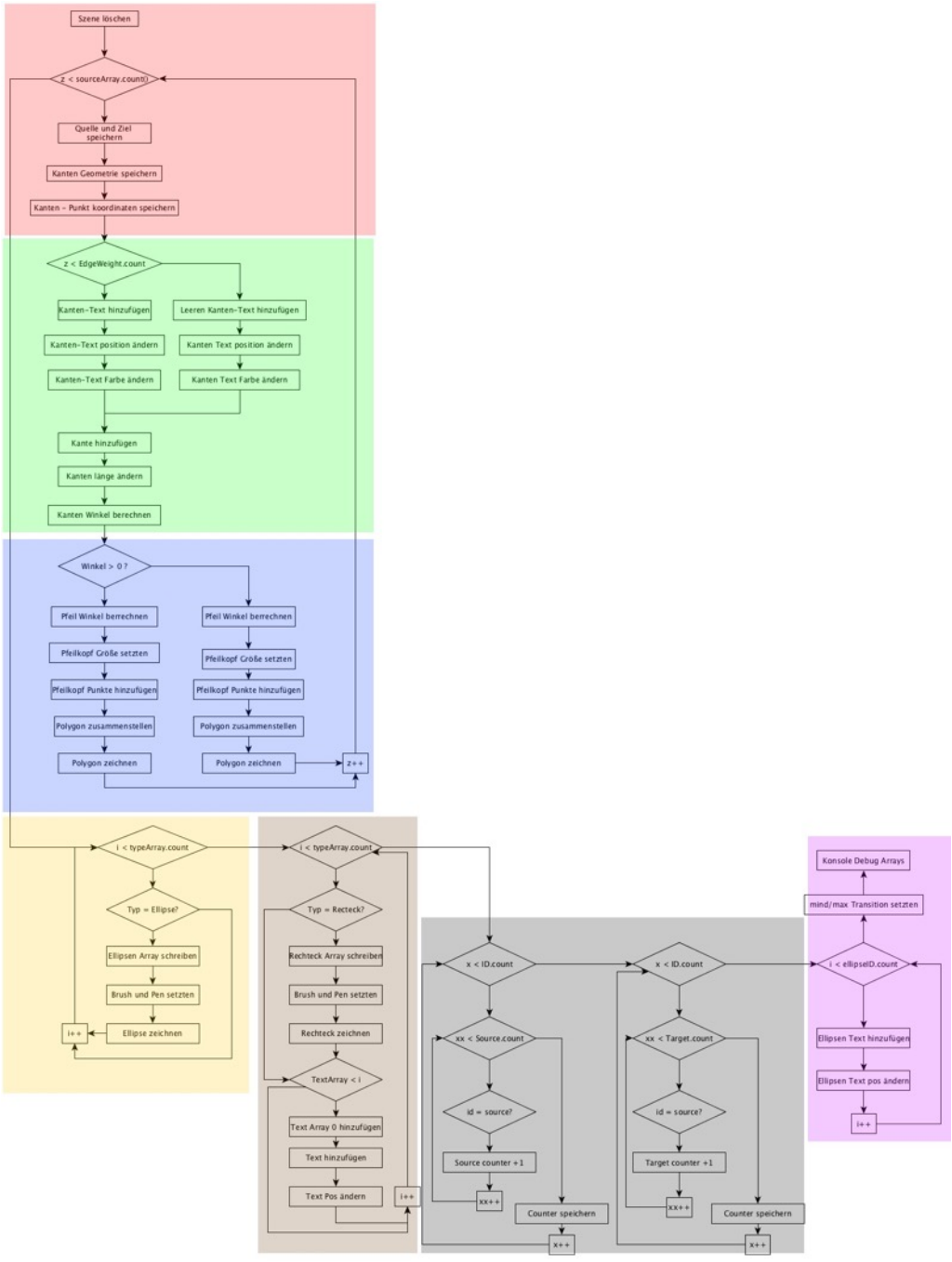


Abbildung 14: Programmablauf Visualisierung

Abbildung 14 zeigt den Programmablauf zur Visualisierung eines in yED Graph Editor erstellten Petri-Netzes. Zur besseren Übersicht wird der Algorithmus abschnittsweise auf den nachfolgenden Seiten beschrieben.

5.2.1 Kanten-Informationen ermitteln

Abbildung 15 schematisiert den Programmablauf nach dem Auswählen eines XML-Dokumentes.

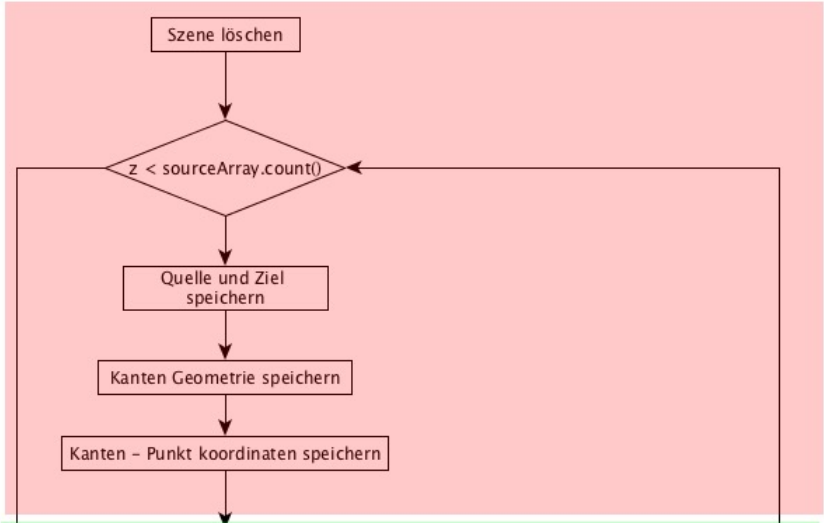


Abbildung 15: Programmablauf Kanten-Information ermitteln

Bei Funktionsaufruf ist es erforderlich, die aktuelle Szene zu bereinigen, damit ein Petri-Netz bei erneutem Hinzufügen auf einer leeren Zeichenfläche visualisiert werden kann. Ohne dieses Vorgehen würden sich die Petri-Netze überlappen und eine Simulation könnte nicht sinnvoll implementiert werden. Die Umsetzung zeigt die Programmzeile 1. Alle Elemente, die sich auf der Szene befinden, werden entfernt.

```
1 qDeleteAll(scene->items());
```

Mit der For-Schleife (Programmzeile 2) wird geprüft, ob das Inkrement kleiner als die Anzahl der sich im Array befindlichen Elemente ist. Im Array befinden sich alle Quell-IDs, diese sind Identifikationsnummern der darzustellenden Objekte. Innerhalb der Schleife werden die IDs der Quelle (Programmzeile 3) und Ziele (Programmzeile 4) in Variablen geschrieben.

```
2 for (int z = 0; z < sourceArray.count() ;z++){
3 int searchSource = sourceArray[z].toInt();
4 int searchTarget = targetArray[z].toInt();}
```

Anschließend kann die Visualisierung der Kanten erfolgen. Durch die Einführung einer Hilfsvariable (Programmzeile 5, MewY) mit der Klasse QLineF (Programmzeile 5) ist es möglich, die Kanten von der Quelle bis zum Ziel inklusive Pfeilkopf sowie mit dem Kantengewicht konsistent abzubilden. Bei der Hilfsvariablen handelt es sich um eine Kante mit den Mittelpunktkoordinaten der Objekte (Ellipsen, Rechtecke), die Quelle und Ziel verbindet, ohne jedoch selbst visualisiert zu werden.

```
5 QLineF MewY(NewXarray[searchSource].toDouble(), NewYarray[search-
  Source].toDouble() + (NewHarray[searchSource].toDouble()/2), New-
  Xarray[searchTarget].toDouble(), NewYarray[searchTarget].toDou-
  ble() + (NewHarray[searchTarget].toDouble()/2));
```

Zur Darstellung der Kanten inklusive Pfeilkopf und Kantengewicht erfolgt eine Segmentierung der Hilfslinien in Punkte. Die Klasse QPointF definiert diese. Zum Identifizieren der Punkte auf der Hilfslinie dient die Funktion pointAt(). Für die Simulation und Visualisierung werden nur der Startpunkt (Programmzeile 7), Mittelpunkt (Programmzeile 8) und Endpunkt (Programmzeile 6) benötigt, grundsätzlich lassen sich aber alle Punkte ermitteln und speichern.

```
6 QPointF parent_max_point = MewY.pointAt(1);
7 QPointF parent_start_point = MewY.pointAt(0);
8 QPointF parent_mid_point = MewY.pointAt(0.5);
```

QPointF

Die Klasse QPointF definiert einen Punkt auf der Zeichenebene. Die Ermittlung der Positionen erfolgt durch kartesische Koordinaten (x, y) und kann mittels Fließkommazahlen präzisiert werden.

5.2.3 Kanten visualisieren

Nach Bestimmung der geometrischen Informationen der Kanten werden, wie in Abbildung 16 zu sehen, die Kanten und Kantengewichte der Szene hinzugefügt sowie die Kantenwinkel berechnet.

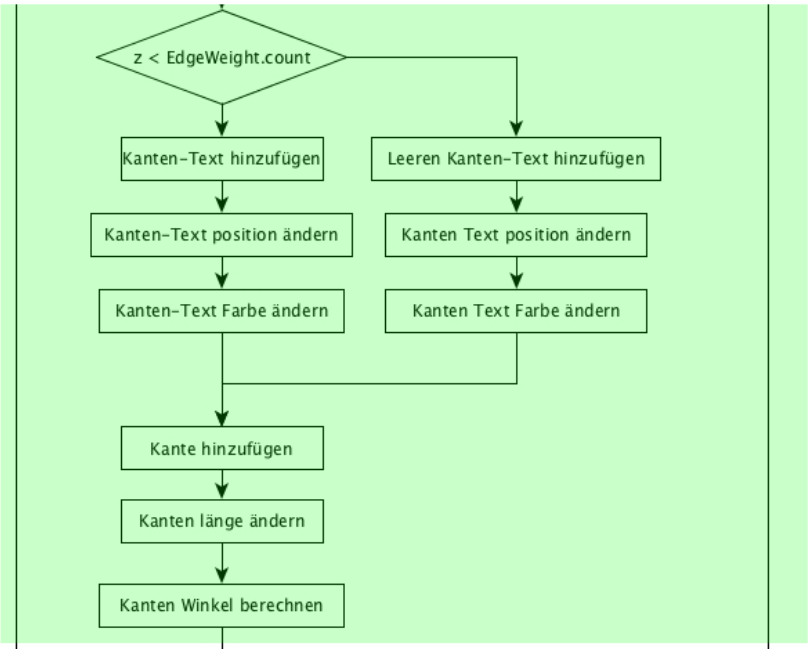


Abbildung 16: Programmablauf Kanten visualisieren

Die Umsetzung erfolgt durch das Ermitteln der Kantengewichte. Allerdings ist nicht explizit für jede Kante ein Kantengewicht angegeben.

Für die Kanten-Texte wird das Array ausgelesen, welches den Kanten-Text beinhaltet (EdgeWeight). Weil nicht jede Kante einen Text besitzen muss, wird mit einer IF-else-Abfrage das Array erst dahingehend überprüft, ob sich an der aktuellen Kante ein Text befindet (Programmzeile 1). Solang ein Wert im Array zur Verfügung steht, wird der Text hinzugefügt, welcher im Array abgespeichert ist. Befindet sich kein Wert im Array, wird eine „0“ hinzugefügt (Programmzeile 2).

```
1 if (z < EdgeWeight.count()){…}else{
2 EdgeWeight.push_back(„0");}
```

Die eigentliche Visualisierung und Formatierung erfolgen mit der Programmzeile 3 bis 5. Der Szene werden der Array-Inhalt mit der Schriftart Arial, der Schriftgröße sieben und der Schriftstill Fett hinzugefügt. Die Orientierung des Textes ist der Mittelpunkt der Kante, mit einem Versatz von sieben Pixeln (Programmzeile 4), um eine Überschneidung der Textausgabe und Kante zu vermeiden. Diese Werte wurden mithilfe von Tests für eine optimale Darstellung ermittelt.

```
3 textWeight = scene->addText(EdgeWeight[z], QFont("Arial", 7,
  QFont::Bold) );
4 textWeight->setPos(parent_mid_point.x()+7, parent_mid_point.y());
5 textWeight->setDefaultTextColor(Qt::red);
```

Mittels der Klasse QLineF und der Koordinaten des Start- und Endpunktes (Programmzeile 6) wird die Kante in die Szene implementiert (Programmzeile 7).

```
6 QLineF arrowLine(parent_start_point.x(), parent_start_point.y(),
  (parent_max_point.x()), (parent_max_point.y()));
7 scene->addLine(arrowLine);
```

5.2.4 QLineF

Die QT Klasse QLineF stellt einen zwei dimensionalen Vektor zur Verfügung. Die Position der Linien ist definiert als X1, Y1, X2 und Y2 oder Startpunkt (P1) sowie Endpunkt (P2). QLineF ermöglicht es, jeden Punkt auf der Linie anzusprechen. Das QT GraphicsView Framework bietet keine Pfeile als Objekte an. Bei einem Petri-Netz wird zur Verbindung von Stellen und Transitionen aber eine Kante (Pfeil) zur Anzeige verwendet. Um diese in der zu entwickelnden Software zu realisieren, werden Polygone, mittels der Klasse QPolygonF, zum Darstellen des Pfeilkopfes (Dreieck) verwendet. Zu diesem Zweck muss jeder Punkt der Linie, an den der Pfeilkopf angefügt werden soll, abgestimmt werden. Der Startpunkt des Polygons liegt dabei direkt auf der Linie. Die weiteren Punkte haben einen Versatz, der sich durch Berechnung der Winkel bestimmen lässt.

Der Pfeilkopf kann aber nicht direkt an das Ende der Kante gesetzt werden, weil sich das Kantenende im Mittelpunkt der Quelle bzw. des Ziels befindet (Abbildung 17). Aus diesem Grund ist eine neue Definition der Länge erforderlich.

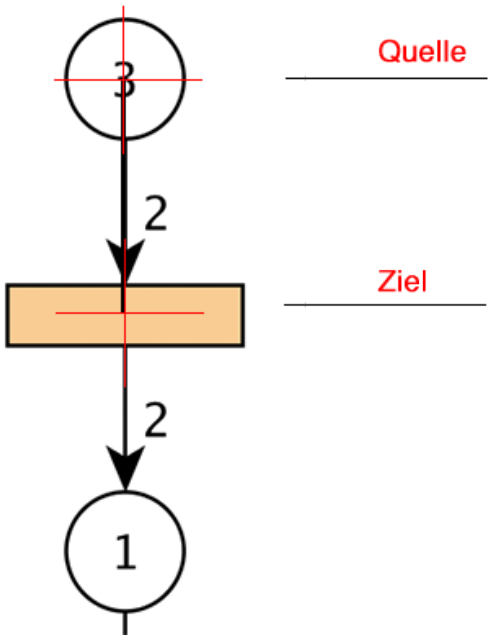


Abbildung 17: Kanten visualisieren

5.2.5 Berechnung Kantenlänge

Mit der Funktion setLength() erhält die Kante eine neue länge (Programmzeile 2). Für die neue Länge wird die aktuelle Kantenlänge mit der Breite der Quell-Stelle/Transition subtrahiert (Programmzeile 1). Anschließend kann der Winkel der Kante berechnet und in eine Variable (angle) abgespeichert werden (Programmzeile 3).

```
1 double par2 =MewY.length()-(NewWarray[searchSource].toDouble()/
  2)-5;
2 arrowLine.setLength(par2);
3 double angle = acos(arrowLine.dx() / arrowLine.length());
```

5.2.6 Berechnung Winkel-Pfeilkopf

Die Abbildung 18 schematisiert den Algorithmus zum Berechnen, der Konfiguration und Ausgabe des Pfeilkopfes.

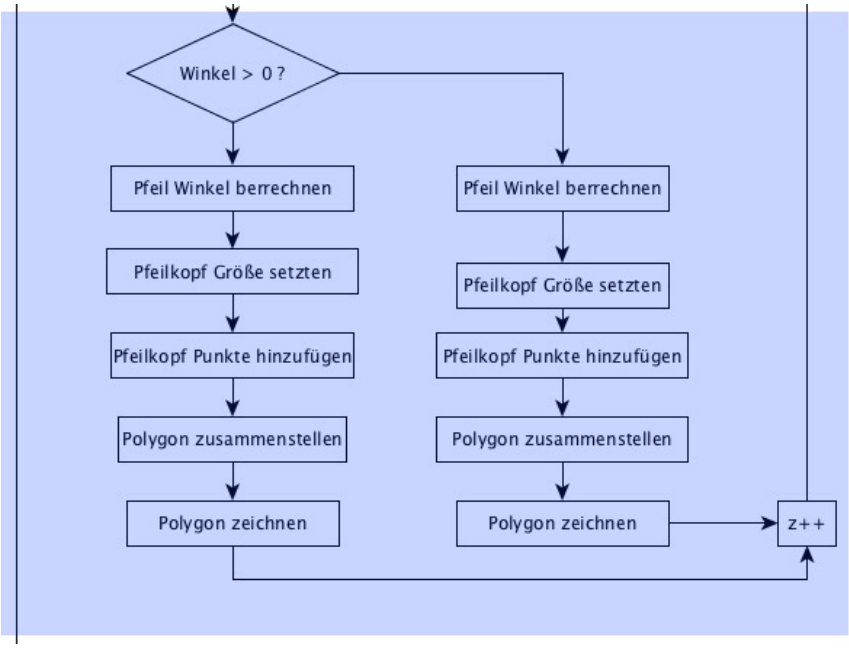


Abbildung 18: Programmablauf: Pfeilkopf visualisieren

Zur Berechnung der Pfeilkopf-Winkel muss die vertikale Position der betroffenen Kante bestimmt werden. Dabei gilt: Ist die vertikale Position der Kante größer oder gleich Null (Programmzeile 1), wird der berechnete Kantenwinkel subtrahiert (Programmzeile 2), bei einer vertikalen Position kleiner Null addiert (Programmzeile 3).

```
1 if (arrowLine.dy()>=0){. . . }else{. . .}


2 angle = (PI*2)-angle;    <- Für v-Pos: >= 0
3 angle = (PI*2)+angle;    <- Für v-Pos: < 0
```

Die berechneten Winkel aus Programmzeile 2 und 3 dienen dazu, die weiteren Punkte des Polygons zu bestimmen (Programmzeile 5 – arP1 und Programmzeile 6 – arP2).

Die Größe des Pfeilkopfes bestimmt die Hilfsvariable „arSize“ (Programmzeile 4)

```
4 qreal arSize = 10;
5 QPointF arP1 = arrowLine.p2() + QPointF(sin(angle - PI + PI / 3)
  * arSize, cos(angle - PI + PI / 3) * arSize);
6 QPointF arP2 = arrowLine.p2() + QPointF(sin(angle - PI / 3) * ar-
  Size, cos(angle - PI / 3) *arSize);
```

Mit allen Punkten kann das Polygon zusammengesetzt (Programmzeile 7) und in die Szene eingefügt werden (Programmzeile 8).

```
7 m_arrowhead << arrowLine.p2() << arP2 << arP1 ;
8 scene->addPolygon(m_arrowhead,outlinePen,Brush);
```

5.2.8 Stellen visualisieren

Im Anschluss an die Visualisierung der Kanten werden die Stellen (Ellipsenform) berechnet und auf die Zeichenfläche ausgegeben. Abbildung 19 zeigt den Algorithmus zur Realisierung der Funktionalität.

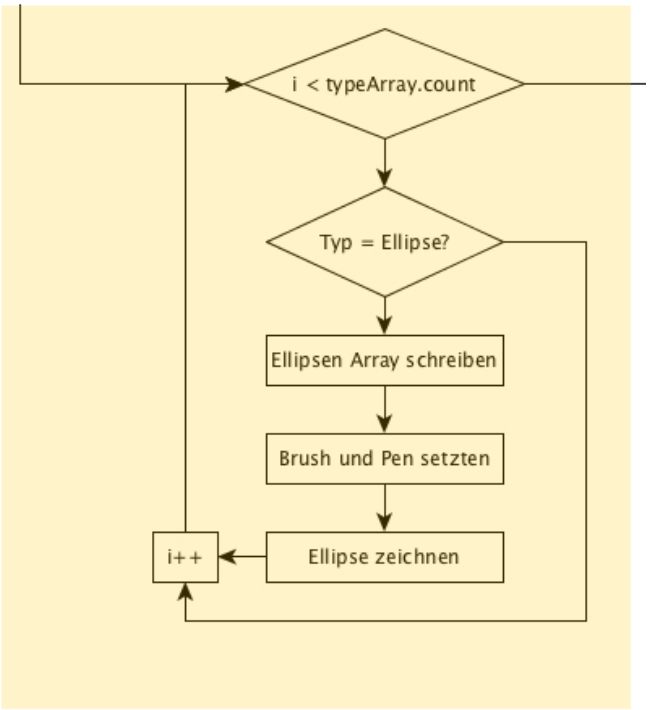


Abbildung 19: Programmablauf: Stellen visualisieren

Die Umsetzung der Stellen (hier Ellipsen) und Transitionen (hier Rechtecke) in der entwickelten Software bedarf des Einsatzes einer For-Schleife (Programmzeile 1), welche das gesamte Typen-Array initialisiert. In diesem befinden sich alle Stellen und Transitionen. Die Schleife läuft solange, bis alle Typen aufgenommen wurden. Zur Unterscheidung von Stellen und Transition wird innerhalb der Schleife geprüft, um welche Form es sich handelt (Programmzeile 2).

```
1 for(int i = 0; i < NewTypeArray.count(); i++){
2 if(NewTypeArray[i] ==„ellipse“){…}}
```

Die identifizierten Stellen und Transitionen sowie deren dazugehörige Ausprägungen wie die Identifikation (Programmzeile 3), die X- und Y-Koordinaten (Programmzeile 4 – 5), die Breite und Höhe (Programmzeile 6 – 7), werden jeweils in ein neues Array überführt.

```
3 elip_ID.push_back(NewIdArray[i]);
4 elip_X.push_back(NewXarray[i]);
5 elip_Y.push_back(NewYarray[i]);
6 elip_W.push_back(NewWarray[i]);
7 elip_H.push_back(NewHarray[i]);
```

Weitere grafische Feineinstellungen wie die Füllfarbe (Brush, Programmzeile 9) oder die Rahmenfarbe (Pen, Programmzeile 10), realisieren die Programmzeilen 8 – 11.

```
 8 QBrush Brush(Qt::black);
 9 Brush.setColor(NewFillArray[i]);
10 QPen outlinePen(Qt::black);
11 outlinePen.setWidth(1);
```

Die Ausgabe im GUI erfolgt durch die Programmzeilen 12, welche die Stellen (Ellipsen) in die Szene integrieren.

```
12 ellipse = scene->addEllipse(NewXarray[i].toDouble() - (NewWar-
   ray[i].toDouble()/2), NewYarray[i].toDouble(), NewWarray[i].to-
   Double(), NewHarray[i].toDouble(), outlinePen, Brush);
```

Die Generierung der Stelle (Ellipse) erfolgt durch die Angabe der X- und Y-Koordinaten sowie der Breiten- und Höhenangaben, der Radius wird von QT automatisch durch die Funktion „addEllipse“ erzeugt.

5.2.9 Transition-Texte

Die Abbildung 20 schematisiert den Algorithmus zur Visualisierung der Transitionen. Die Funktionsweise ist identisch mit der Stellen-Erzeugung, jedoch werden die Texte direkt in dieser Funktion verarbeitet.

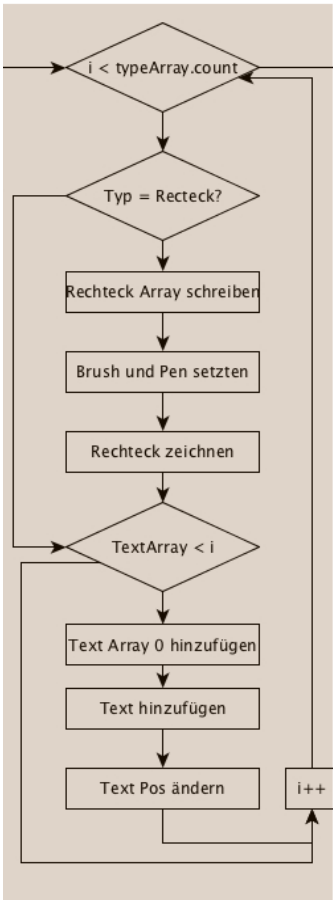


Abbildung 20: Programmablauf: Transition und Transition-Texte

Für die Simulation und spätere Weiterentwicklung, ist es notwendig, dass die Transitionen einen Text beinhalten. Transitionen sind grundsätzlich inhaltslos, weil diese nur schalten. In Folge dessen wird in diese nur der Wert „0“ eingefügt (Programmzeile 2).

Bei Textobjekten erfolgt die Formatierung der Position immer nach dem Initialisieren des Textes, weil dieser stets mit den Koordinaten „X=0, Y=0“ in die Szene implementiert wird. Programmzeile 4 setzt die Textposition in die Mittelpunkte der Transitionen.

```
1 if (i < NewTextArray.count())
  . . .
2 NewTextArray.insert(i, "0");
3 textA = scene->addText(NewTextArray[i], QFont("Arial", 10) );
4 textA->setPos(NewXarray[i].toDouble(), NewYarray[i].toDouble());
```

Um Programmfehler und Abstürze zu verhindern, prüft Programmzeile 1, ob sich das Inkrement noch innerhalb des Arrays befindet.

5.2.10 Quellen und Ziele zählen

Im Anschluss an die Visualisierung und zur Vorbereitung der Simulation wird, wie in Abbildung 21 schematisiert, geprüft, wie viele Quellen und Ziele die jeweiligen grafischen Elemente wie Stellen und Transitionen besitzen.

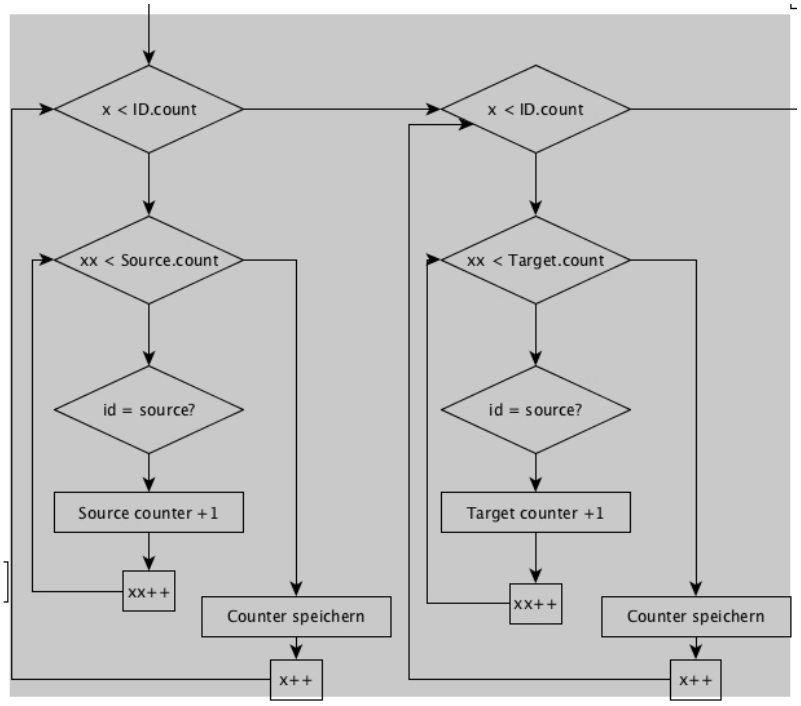


Abbildung 21: Programmablauf: Quelle und Ziele zählen

Die Realisierung erfolgt dabei mit einer gestaffelten For-Schleife (Programmzeile 2 – 3). Jede ID wird innerhalb des Quellen bzw. Ziel-Arrays überprüft. Immer, wenn eine ID an einer Stelle im Quell- bzw. Ziel-Array gefunden wurde, wird die Zählvariable hochgezählt (Programmzeile 4 – 5). Abschließend kann die Zählvariable in ein neues Array gespeichert werden (Programmzeile 6).

5.2.11 Stellenbeschriftung und Outputstream

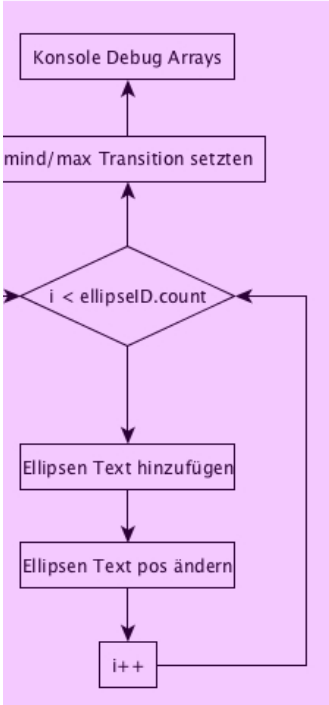


Abbildung 22: Programmablauf: Stellenbeschriftung

Zur Vervollständigung des Visualisierungs-Algorithmus veranschaulicht Abbildung 22 die Implementierung der Stellenbeschriftung. Die Vorgehensweise ist kongruent zur Kantenvisualisierung.

```
1 int cot = 0;
2 for (int x = 0; x < NewIdArray.count(); x++){
3 for (int xx = 0; xx < sourceArray.count(); xx++){
4 if (NewIdArray[x] == sourceArray[xx]){
5 cot ++;}}
6 sorCount.push_back(cot);
```

Für das Verfolgen und Kontrollieren aller Arrays wird zusätzlich ein Outputstream gestartet (Programmzeile 1 – 2). Der Outputstream gibt in der Entwicklerkonsole alle Arrays aus.

```
1 qDebug() << "\r\nText :" <<NewTextArray;
2 qDebug() << "\r\nElip ID : " <<elip_ID;
3 . . .
```

6. Simulation

6.1 Einleitung

Definition: Simulation ist das Nachbilden eines dynamischen Prozesses in einem System mit Hilfe eines experimentierfähigen Modells, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind (VDI, 2011).

6.2 Ereignisorientierte Simulation

Für das Projekt wurde der Ansatz der ereignisorientieren Simulation verwendet. Bei einer solchen handelt es sich um eine diskrete Simulation. Dabei können sich die Werte der Modellvariablen nur zu bestimmten, diskreten Zeitpunkten verändern, außerdem gibt es klar abgegrenzte Modellzustände. Der Simulationsfortschritt erfolgt dabei durch Ereignisse. Verglichen mit der kontinuierlichen Simulation hat die ereignisorientierte Simulation den Vorteil, dass diese mit standardisierten Elementen, wie Wahrscheinlichkeitsverteilungen oder Zufallszahlen, darstellbar ist. Bezogen auf das Projekt ist dies die logische Wahl des Simulationsmodells. Bei einem Petri-Netz übernehmen die Transitionen die eigentliche Steuerung, weil sie die einzige aktive Komponente darstellen. Bei der Programmierung und Simulation muss also an der Stelle der Transition geprüft werden, welche Kantengewichte und Schaltregeln vorhanden sind; dies geschieht zu diskreten Zeitpunkten, die Variablen ändern sich nur sprunghaft.

6.3 Grafische Simulation Version 1 - Intervall markieren

Die grafische Simulation dient dazu, den Anwender darüber zu informieren, welches Objekt gegenwärtig aktiv ist. Die Basis der grafischen Simulation (Abbildung 23) ist eine For-Schleife, die durch eine Zeitfunktion (Timer-Funktion) gesteuert wird. Das Intervall der Zeitfunktion beträgt 1000 Millisekunden (einer Sekunde).

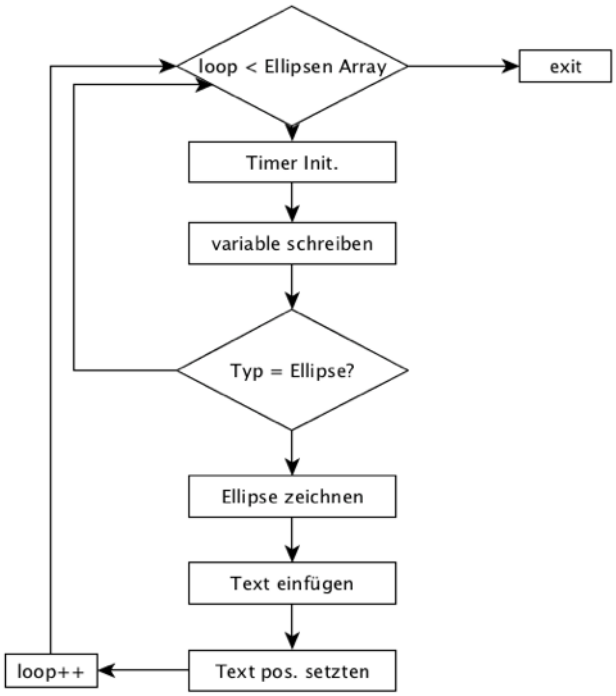


Abbildung 23: Programmablauf: Grafische Simulation

Die grafische Simulation markiert die Stellen (Ellipsen) und ändert den Stelleninhalt auf die berechnete Markenanzahl. Die Schleife läuft solange, bis alle Stellen abgearbeitet sind (Programmzeile 1).

```
1 if (loop < elip_ID.count())
```

Die Initialisierung des Zeitmessers (Timer) erfolgt innerhalb der Schleife mit einem deklarierten Zeitintervall von 1000 Millisekunden (Programmzeile 1), der den „Slot“ aktiviert (Programmzeile 3).

```
2 int simTime = 1000;
  . . .
3 QTimer::singleShot(simTime, this, SLOT(singleShotGreen()));
```

Die in Programmzeile 4, deklarierte Variable „sor“ enthält alle zu markierenden Stellen (Ellipsen).

```
4 int sor = elip_ID[loop++].toInt();
```

Zur Differenzierung der Objekttypen prüft Programmzeile 5 mithilfe der bereits ermittelten Identifikatoren (IDs), ob es sich beim identifizierten Objekt um eine Stelle (Ellipse) handelt.

```
5 if(NewTypeArray[sor] =="ellipse"){
```

Die Ausgabe im GUI erfolgt nach dem bereits bekannten Schema, wie in Programmzeile 6 zu sehen.

```
6 ellipse = scene->addEllipse(NewXarray[sor].toDouble() - (NewWar-
  ray[sor].toDouble()/2), NewYarray[sor].toDouble(),
  NewWarray[sor].toDouble(), NewHarray[sor].toDouble(), outline-
  Pen, Brush);
```

6.3.1 Mengen berechnen

Die Abbildung 24 zeigt den Algorithmus zum Berechnen der Token / Mengen beim Schalten einer Transition.

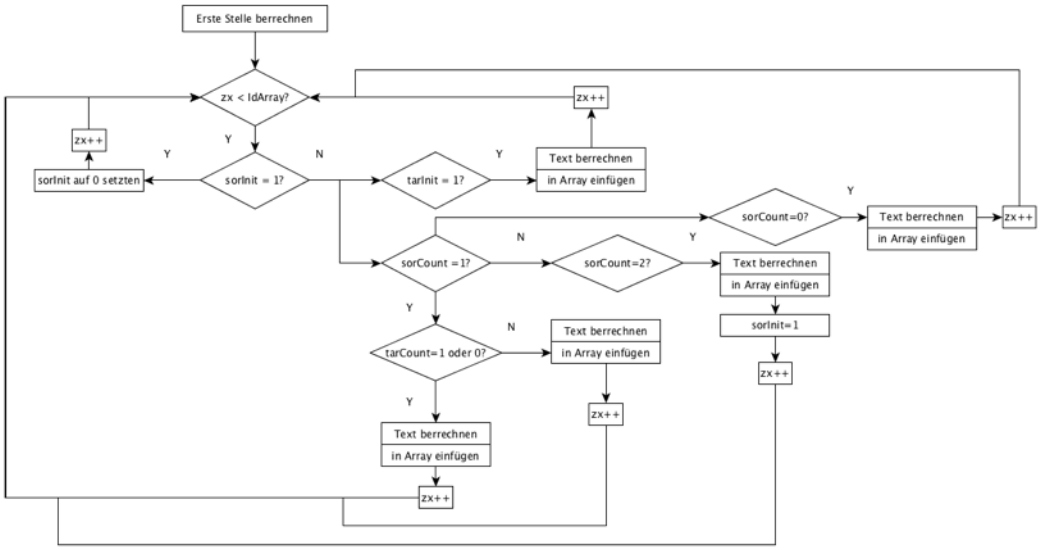


Abbildung 24: Programmablauf: Mengen berechnen

Die Berechnung der neuen Token / Mengen muss die Start-Menge mit dem Kantengewicht vor der Stelle addiert und mit dem Kantengewicht nach der Stelle subtrahiert werden. Die nachfolgende Formel bezieht sich auf das Beispiel in Abbildung 25.

Neue Menge = Start Menge + Gewicht (Quelle) - Gewicht (Ziel)

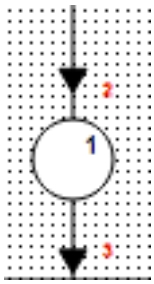


Abbildung 25: Mengen berechnen Beispiel

Eine Ausnahme bildet die erste Stelle im Petri-Netz, diese besitzt keine Quelle und kann somit direkt berechnet- sowie ins Array eingefügt werden (Programmzeile 1).

```
1  simText.push_back(NewTextArray.first().toInt() - EdgeWeight.-
   first().toInt());
```

Es ist notwendig, zu unterscheiden, wie viele Quellen und Ziele eine ID besitzt. Je nach Anzahl der Quellen und Ziele müssen unterschiedliche Berechnungsweisen angewendet werden.

Die Berechnung der Anzahl (Quellen sowie Ziele der jeweiligen ID) ist bereits erfolgt (vergl. Abschnitt 5.2.10). Diese Arrays werden für die weiteren Berechnungen genutzt. Für jede Stelle findet eine Prüfung der Anzahl an Quellen und Ziele statt.

Je nach Anzahl der Quellen und Ziele einer Stelle erfolgt eine Fallunterscheidung, wie nachfolgend dargestellt:

Quellen = 1 Ziele = 1 oder 0	simText.push_back(NewTextArray[zx].toInt() + Edge- Weight[zx - 1].toInt() - EdgeWeight[zx].toInt());

Quellen = 1
Ziele =/= 1
oder 0
	simText.push_back(NewTextArray[zx].toInt() + Edge-
	Weight[zx].toInt() + EdgeWeight[zx - 1].toInt() - Ed-
	geWeight[zx + 1].toInt());
	init = sourceArray[zx].toInt();
	tarInit = 1;

Quellen = 2	simText.push_back(NewTextArray[zx].toInt() + Edge-
	Weight[zx - 1].toInt() - EdgeWeight[zx].toInt() - Ed-
	geWeight[zx + 1].toInt());
	sorInit = 1;

Quellen = 0	simText.push_back(NewTextArray[zx].toInt() + Edge- Weight[zx].toInt());
sorInit = 1	Überspringen
tarInit = 1	simText.insert(init, NewTextArray[init].toInt() + Ed- geWeight[init - 1].toInt() - EdgeWeight[zx - 1].to- Int());

6.4 Grafische Simulation Version 2 - Animation

Zur Simulation des Petri-Netzes wurde in der ersten Programmversion eine simplere Animation entwickelt und beibehalten, weil eine Weiterentwicklung dieser Animationsmethode möglicherweise interessant sein könnte. Anders als bei der grafischen Simulation Version 1 (vergl. Abschnitt 6.3) werden die aktuell aktiven Objekte nicht markiert, sondern eine Ellipse durchläuft das eingefügte Petri-Netz. Abbildung 26 schematisiert den Algorithmus dieser Simulationsmethode, welche nachfolgend beschrieben wird.

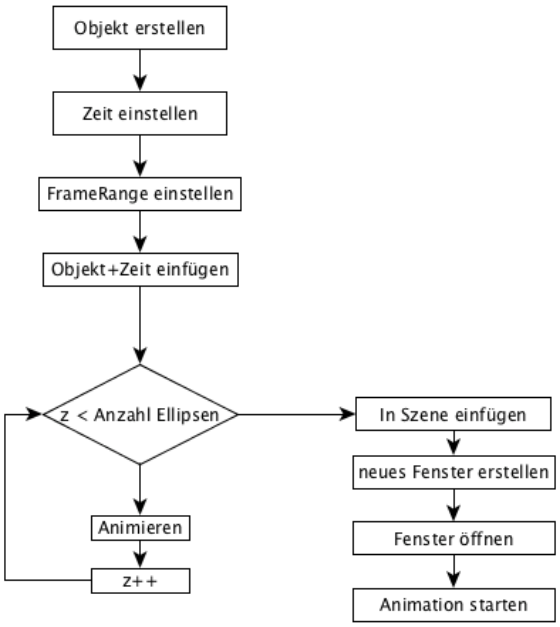


Abbildung 26: Programmablauf: Grafische Simulation 2

Die Animation benötigt ein Objekt. Das in Programmzeile 1 erstellte Objekt ist eine Ellipse mit der Größe von 20 x 20 Pixel.

```
1 QGraphicsItem *token = new QGraphicsEllipseItem(0,0,20,20);
```

Die Steuerung der Animation übernimmt die Zeitleiste. Diese legt fest, wie lange die Animation abläuft. Weil diese je nach Größe des zu simulierenden Petri-Netzes unterschiedlich lang ausfällt, wird die Anzahl der Stellen innerhalb der Szene mitberücksichtigt (Programmzeile 2 – 3).

```
2 timeline = elip_X.count()*timeline;
3 QTimeLine *timeA = new QTimeLine(timeline);
```

Die Konfiguration der „Frame Range“ regelt die Anzahl der zu animierenden Einzelbilder und den Start sowie das Ende der Simulation (Programmzeile 4).

```
4 timeA->setFrameRange(0,10000);
```

Die Zuordnung der zu animierenden Objekte und die Zeitleiste erfolgen durch die Programmzeile 5 – 7.

```
5 QGraphicsItemAnimation *animation = new QGraphicsItemAnimation;
6 animation->setItem(token);
7 animation->setTimeLine(timeA);
```

Das Simulationsobjekt (Ellipse) soll nacheinander alle Stellen ansteuern, realisiert durch die For-Schleife (Programmzeile 8).

Das Animationsobjekt bewegt sich zu jeder Position der Stellen. Die Regelung der Geschwindigkeit übernimmt die Variable „StepValue“ (`z/50`), welche innerhalb der Funktion setPosAt() definiert wird (Programmzeile 9).

```
8 for (int z = 0; z < elip_X.count() ;z++){
9 animation->setPosAt(z / 50.0, QPointF(elip_X[z].toDouble() +
  (elip_W[z].toDouble()/2) , elip_Y[z].toDouble() + (elip_W[z].to-
  Double()/2)));}
```

Die Ausgabe der Simulation im GUI erfolgt durch das Einsetzen des zu animierenden Objektes in die Szene (Programmzeile 10). Außerdem erscheint ein neues Animationsfenster (Programmzeile 12), in welchem die Simulation abläuft.

```
10 scene->addItem(token);
11 QGraphicsView *view = new QGraphicsView(scene);
12 view->show();
13 timeA->start();
```

6.5 Text-Simulation

Die Entwicklung der Text-Simulation ist ein Hilfsmittel zur Programmierung der Simulationssoftware und zur Weiterentwicklung. Bei der textbasierten Simulation verändert sich nur der Inhalt eines „Labels“, dieses zeigt die Identifikatoren (ID) der Stellen, die zum Zeitpunkt der Simulation aktiv sind an. Zur Realisierung dieser Simulationsmethode dient das QWidget: Text-Browser.

Um Programm- und Darstellungsfehler zu verhindern, wird der Inhalt im Text-Browser bei Funktionsaufruf gelöscht (Programmzeile 1).

```
1 ui->textBrowser->clear();
```

Die Umsetzung der Text-Simulation ist nahezu kongruent zur grafischen Simulation X. Ein Zeitmesser startet in einem angegebenen Zeitintervall den Slot, solange die Anweisung der Schleife erfüllt ist (Programmzeile 2 – 3).

```
2 if (loop < elip_ID.count()){

3 QTimer::singleShot(1000, this,
  SLOT(on_pushButton_8_clicked()));}
```

Innerhalb der Schleife verändert sich nur der im Text-Browser anzuzeigende String (Programmzeile 4).

```
4 QString rowloop = "Token position -->"+elip_ID[loop++];
```

Damit der Anwender erkennen kann, dass die Text-Simulation abgeschlossen ist, wird an das Schleifenende der Text „Ende“ (Programmzeile 5 – 7) gesetzt.

```
5 tokenID = "Ende";

6 QString row = "Token position -->"+tokenID;

7 ui->textBrowser->append(row);
```

7. GUI

7.1 Benutzeroberfläche

Abbildung 27: Benutzeroberfläche

Abbildung 27 visualisiert die Benutzeroberfläche. Für eine bessere UX (User Experince) wurden keine Buttons innerhalb des Interfaces definiert, weil sich diese bei unterschiedlichen Bildschirmgrößen verschieben. Alle Funktionen sind durch eine Menübar oder durch „Hotkeys“ bedienbar. Die Funktionalitäten werden nachfolgend beschrieben.

Nr.	Objekt	Funktion
1	Menüleiste	XML-Datei laden, Programm schließen, Animation zurücksetzen, Simulation starten, Text-Simulation starten, Fokus setzen, Animationstest

Nr.	Objekt	Funktion
2	Simulationsfenster	Visualisierung des in yED Graph Editors erstellten Petri-Netzes
3	Objekt-Tabelle	zeigt Informationen zu allen im Simulationsfenster dargestellten Objekten, wie ID, Objekttyp, Koordinaten (X,Y), Breite sowie Höhe
4	Objekt-Auflistung	listet alle Stellen und Transitionen, sortiert nach aufsteigender ID sowie die dazugehörigen Marken innerhalb dieser auf
5	Simulationszeit (v1)	betrifft nur die Simulation (v1); Einstellung mithilfe eines Reglers und Anzeige der Simulationszeit. Je höher die gewählte Zeit, desto schneller läuft die Simulation ab.
6	Simulationszeit (v2)	betrifft nur die Simulation (v2); Einstellung mithilfe eines Reglers und Anzeige der Simulationszeit. Je höher die gewählte Zeit, desto schneller läuft die Simulation ab.
7	Text-Browser	Visualisierung der Text-Simulation

7.2 GUI-Funktion

Abbildung 28 zeigt die Funktionen hinter dem Menüleistenpunkt Datei.

Abbildung 28: Menü: Datei

- **Datei öffnen:** öffnet und lädt ein in yED Graph Editor erstelltes Petri-Netz. Es können nur XML-Dateien ausgewählt werden.
- **Beenden:** schließt die Anwendung

Über die Menüleiste „Bearbeiten“ gelangt man zu Funktionen zur Steuerung der Simulation, wie in Abbildung 29 zu erkennen.

Abbildung 29: Menü: Bearbeiten

- **Szene Aktualisieren:** setzt die Simulation zurück; alle Berechnungen, wie z. B. Mengen der Stellen, werden in den
- Ursprungszustand gesetzt.**Simulation (v2) Starten:** startet die grafische Simulation Version 2, ein elliptisches Objekt durchläuft das Petri-Netz (vergl. Abschnitt 6.4).
- **Simulation (v1) Starten:** Hierbei handelt es sich um die finale Version der Simulation. Die aktuellen Mengen der Stellen werden berechnet und kenntlich gemacht (vergl. Abschnitt 6.3).
- **Text-Simulation Starten:** initialisiert die Text-Simulation und gibt diese im Text- Browser aus (vergl. Abschnitt 6.5).

Hinter „Extras“ befinden sich Funktionen wie das Setzen des Fokus und ein Animationstest, wie Abbildung 30 visualisiert.

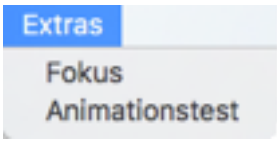


Abbildung 30: Menü: Extras

- **Fokus:** Zum Setzen des Fokus muss zuvor in der Tabelle ein Objekt ausgewählt werden. Das gewählte Objekt wird anschließend im zu simulierenden Petri-Netz markiert.
- **Animationstest:** Dabei handelt es sich um ein Fragment der Entwicklung, um Erkenntnisse über das GraphicsView Framework zu erhalten. Dabei wird ein rotes Objekt im GUI animiert.

8. Weiterentwicklung

Die aktuelle Programmversion bietet eine gute Basis für die zukünftige Weiterentwicklung. Elementare Funktionen, wie das Einlesen eines in yED Graph Editor erstellen Petri-Netzes im XML-Format sowie die grafische Ausgabe im GUI, sind implementiert. Als Hilfsmittel speziell für die Programmierung dienen die Tabelle Objektinformationen, Objektaufzählungen und die textbasierte Simulation. Die Tabelle „Objektinformation“ zeigt alle Stellen, Transitionen, Kantengewichte inklusive der Koordinaten (x, y). Außerdem ist es möglich, einzelne Objekte durch Auswahl zu fokussieren, um eine bessere Übersicht zu erhalten.

Die Funktion zur Simulation des Petri-Netzes inklusive Kantengewicht ist ebenfalls programmiert. An dieser Stelle müssen für eine vollständige Simulation des THOR-Netzes noch nachfolgende Funktionen realisiert werden.

- Implementierung der Kantentypen: Enabling-Kanten, Inhibitor-Kanten, Consuming-Kanten
- Implementierung der Stellentypen: **Queue-Stellen, Stack-Stellen, Priority-Queue-Stellen**

Eine Implementierung der Consuming-Kanten muss geprüft werden, weil dieser Kantentyp keine Auswirkung auf das Schalten der Transitionen hat.

Der Programmcode ist prozedural aufgebaut, auf eine Klassenstruktur musste aus zeitlichen Gründen verzichtet werden. Zur besseren Übersicht und einfacheren Weiterentwicklung sollte diese aber in Klassen strukturiert sein. Daraus könnten nachfolgende Klassen resultieren.

- XML Lesen und Werte speichern
- Visualisierung: Stellen
- Visualisierung: Kanten
- Visualisierung: Transitionen
- Visualisierung: Texte
- Mengenberechnung (Simulation)
- Grafische Simulation (Intervall)

Desweiteren sind Anpassung der implementierten Szene möglich. Besonders in Hinblick auf eine „Responsive-Design“ Oberfläche (Oberfläche passt sich automatisch der Bildschirmgröße an), lässt sich das UI und die Grafische-Szene optimieren.

9. Zusammenfassung

Das Ziel der Entwicklung war es eine Basis, für das Tool zur Visualisierung von Simulationsabläufen mittels Petri-Netzen, zu erstellen. Der Kern der Software bildet das Einlesen und Extrahieren der, in yED-Grapheditor erstellten, Petri-Netze (vergl. Abschnitt 4.4.1 - 4.4.3). Durch die Komplexität der vorhandenen XML-Struktur, war es notwendig, Daten durch unterschiedliche Methoden zu parsen. Das Extrahieren der XML-Werte, wurde mit der Schnittstelle „X-Path“ realisiert (vergl. Abschnitt 4.4.2), welche wiederum mithilfe von „DOM“ in Arrays gespeichert werden konnten (vergl. Abschnitt 4.4.3).

Die Visualisierung (vergl. Abschnitt 5), wurde durch die Verwendeung des „Graphics View Framework“ ausgearbeitet, welches alle Elemente eines Petri-Netzes in eine definierte Szene implementiert. Durch die Verwendung des „Graphics View Framework“ war es möglich, vordefinierte Elemente (beispielsweise Ellipsen oder Rechtecke) direkt anzuwenden, ohne weitere Anpassung vornehmen zu müssen (vergl. Abschnitt 5.2.8). Elemente welche kein Bestandteil des Frameworks sind, mussten durch eine eigene Logik visualisiert werden. Das war unter anderem für die Visualisierung der Pfeile notwendig (vergl. Abschnitt 5.2.3 - 5.2.6). Weitere Elemente wie die Stellen-Texte oder die Kanten-Gewichte, konnten direkt aus den Arrays gelesen und in die Szene eingesetzt werden (vergl. Abschnitt 5.2.11).

Die grafische Simulation bietet drei unterschiedliche Methoden. Die erste Simulationsmethode (vergl. Abschnitt 6.3), markiert (in einem gegebenen Intervall) die jeweils aktive Stelle und ändert den Stelleninhalt auf die berechnete Markenanzahl. Anders als bei der ersten Simulationsmethode, werden bei der zweiten grafischen Simulationsmethode (vergl. Abschnitt 6.4), die aktuell aktiven Objekte nicht markiert, sondern eine Ellipse durchläuft das eingefügte Petri-Netz. Die dritte Simulationsmethode (vergl. Abschnitt 6.5) verändert, dagegen nur den Inhalt eines „Labels“, welches die ID (Identifikations-Nummer) der aktiven Stelle anzeigt.

Für die bessere Interaktion mit der Software, wurde abschließend ein GUI (Benutzeroberfläche) entwickelt und die einzelnen Funktionen aufgelistet (vergl. Abschnitt 7).

Alle Basis-Funktionen, die als Meilensteine definiert waren (vergl. Anhang 1: Roadmap), sind implementiert und funktionsfähig. Die entwickelte Software bildet somit einen Ausgangspunkt für zukünftige Weiterentwicklung.

10. Persönliches Fazit

10.1 Persönliches Fazit: Dominik Srednicki

Ich hatte mir von der Projektarbeit zwei Sachen erhofft: Erstens, Erfahrungen im Bereich Simulationssoftware zu sammeln und zweitens, meine Programmierkenntnisse zu erweitern. Beide persönlichen Ziele wurden teilweise erreicht. Ich hatte im Rahmen dieser Projektarbeit viel Zeit mit der Einarbeitung in die Entwicklungsumgebung QT verbracht, da mir diese bis dahin unbekannt war.

Durch den sehr hohen Arbeitsaufwand der zu implementierenden Funktionen war die Einarbeitung in QT ein störendes Element, welches ich gerne mit einer anderen Entwicklungsumgebung verhindert hätte (Microsoft Visual Studio Pro). Besonders in Bezug auf Erweiterung (bspw. .net Framework - XML Reader Klassen oder .NET Compact für grafische Funktionen) würde sich eine andere Entwicklungsumgebung eher lohnen. Positiv überrascht wurde ich von QT durch die fehlerfreie Zusammenarbeit zwischen Windows und MacOSX-Systemen, welche keine Konvertierung des Codes erforderte.

Außerordentlich fordernd war die Entwicklung der Funktionen: „XML Lesen und speichern“ und die Visualisierung des Graphen, da es immer mehrere Methoden gibt, diese zu implementieren. Die beste Methode zu finden und diese dann zu implementieren, stellte sich ebenfalls als sehr zeitaufwendig dar. Selbst kleine Visualisierungsaufgaben (bspw. Zeichnen eines Pfeilkopfes) stellten sich als anspruchsvoll heraus, da QT keine fertige Funktionen bietet, diese zu zeichnen.

All dies führte dazu, dass die Simulationsfunktion der Arbeit leider sehr knapp ausfiel und eine Weiterentwicklung der Simulationsfunktion erforderlich macht.

10.2 Persönliches Fazit: Damian Srednicki

Die Arbeit einen Petri-Netz-Simulators umzusetzen, stellte sich als anspruchsvolle Entwicklung heraus. Das lag vor allem an der Einarbeitungsphase in die Entwicklungsumgebung QT Creator. Das Erlernen von bereits bekannten Funktionen, wie sie mir beispielsweise durch Visual Studio oder Netbeans bekannt sind, ist eine zeitaufwendige Prozedur und vergleichbar mit dem Studieren einer neuen Programmiersprache, die viel Engagement verlangt. Die Priorität lag dabei stets auf einer Software, die ein Petri-Netz simulieren kann, um die gestellte Aufgabe zu erfüllen. Leider ist somit der Programmcode nur prozedural aufgebaut. Dieser sollte zum besseren Verständnis und einer einfacheren Weiterentwicklung durch mehrere Entwickler in Klassen unterteilt werden (vergl. Abschnitt 8).

Die vorliegende Programmversion bietet für die Weiterentwicklung eine gute Basis. Speziell das Einlesen, Visualisieren und Extrahieren der benötigten XML-Elemente ist abgeschlossen. Dies beanspruchte allerdings mehr Zeit als ursprünglich geplant, weil verschiedene XML-Schnittstellen, wie DOM oder SAX, nicht die gewünschten Ergebnisse erzielten. Aus diesen Grund wurde XPath verwendet, welches letztendlich nicht nur die erforderlichen Resultate erreichte, sondern auch eine verständliche Syntax hervorbrachte. Die eigentliche Simulation in grafischer Form dient als Plattform für die zukünftige Entwicklung. Die Logik ist meiner Meinung nach verständlich und in dieser Arbeit dokumentiert. Jedoch muss die Entwicklung speziell in Richtung Thor-Netz weitergeführt werden. Eine komplette Implementierung war in der gegebenen Zeit leider nicht mehr möglich. Um diese abzuschließen, müssen noch die Objekte wie Queue-Stellen, Stack-Stellen, Priority-Queue-Stellen und Kantentypen, wie Enabling-Kanten, Inhibitor-Kanten oder Consuming-Kanten implementiert werden. Auf Grundlage der programmierten Funktionen und Erkenntnisse sollte dies jedoch keinen übermäßig hohen Entwicklungsaufwand erfordern.

II. Literaturverzeichnis

Uni Erlangen (2013): Lehrstuhl für Software Engineering: Petir Netze. http://www11.informatik.uni-erlangen.de/Lehre/WS1314/PR-SWE/Folien/folienA3.pdf

TU-Dortmund (2014): Prof. Dr.- Ing. Markus Rabe Vorlesungsunterlagen Grundlagen der Simulationstechnik

TU-Dortmund (1997): Schöf, Sonnenschein et al. 1997 - Distributed Net Simulation.pdf

The Qt Company (2015): Qt Documentation. http://doc.qt.io

Timed Hierarchical Object-Oriented Petri Net, Petri Net, Theory and Applications (2008): Hua Xu State Key Laboratory of Intelligent Technology and Systems Vedran Kordic (Ed.), ISBN: 978-3-902613-12-7

Distributed Net Simulation (1997): Schöf, Sonnenschein

Discrete Mathematics for Software Engineers (2015): University of Texas at Dallas, Petri Nets by Dr. Chris Ling

Alternative Computer Programming: Introduction to Qt Creator: http://www.alternative-computer-programming.com/cpp-tutorial-001-qt-creator-introduction.html

KDE TechBase: Development/Tutorials/Using Qt Creator: https://techbase.kde.org/Development/Tutorials/Using_Qt_Creator

Department of Computer Science University of Maryland (1991): Discrete-Event Simulation A. Udaya Shankar

McCormick School of Engineering, Northwestern University: Writing a Discrete Event Simulation in Java: http://www.cs.northwestern.edu/~agupta/_projects/networking/QueueSimulation/mm1.html

en.Wikipedia: Discrete event simulation: https://en.wikipedia.org/wiki/Discrete_event_simulation