



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Justus Kilpatrick Hartig

Entwicklung eines Werkzeugs zur Modellierung und
Source-Code-Erzeugung von steuerungstechnisch
interpretierten Petri-Netzen für speicherprogram-
mierbare Steuerungen

Justus Kilpatrick Hartig

Entwicklung eines Werkzeugs zur Modellierung
und Source-Code-Erzeugung von steuerungs-
technisch interpretierten Petri-Netzen für spei-
cherprogrammierbare Steuerungen

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Regenerative Energiesysteme und Energiemanagement
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ing. André Wenzel
Zweitgutachter: Prof. Dr. Ing. Holger Gräßner

Abgegeben am 28.06.2019

Justus Kilpatrick Hartig

Thema der Bachelorthesis

Entwicklung eines Werkzeugs zur Modellierung und Source-Code-Erzeugung von steuerungstechnisch interpretierten Petri-Netzen für speicherprogrammierbare Steuerungen

Stichworte

SIPN, steuerungstechnisch interpretierte Petri Netze, SPS, Source Code Erzeugung, IEC 61131-3

Kurzzusammenfassung

In der Steuerungstechnik werden Petri Netze eingesetzt, um Anlagenprozesse zu modellieren. Innerhalb dieser Arbeit wird ein Hilfsmittel entwickelt, mit dem steuerungstechnisch interpretierte Petri Netze graphisch erstellt und in Source-Code überführt werden können. Der erzeugte Source-Code soll dabei der internationalen Norm IEC 61131-3 entsprechen, sodass dieser für verschiedene speicherprogrammierbare Steuerungen verwendet werden kann.

Justus Kilpatrick Hartig

Title of the paper

Development of a tool for modelling and source code generation of signal interpreted Petri Nets for programmable logic controllers.

Keywords

SIPN, signal interpreted Petri Net, PLC, source code generation, IEC 61131-3

Abstract

In the control technology Petri nets are used to model plant processes. Within this Thesis, a tool is being developed with which Petri nets can be created graphically and converted into source code. The generated source code should correspond the international standard IEC 61131-3, so that it can be used for various programmable logic controllers.

Inhaltsverzeichnis

<i>Inhaltsverzeichnis</i>	<i>I</i>
<i>Abbildungsverzeichnis</i>	<i>III</i>
<i>Tabellenverzeichnis</i>	<i>IV</i>
<i>Glossar</i>	<i>V</i>
1 <i>Einleitung</i>	1
1.1 Einführung	1
1.2 Ziel der Arbeit	1
1.3 Motivation	1
1.4 Aufbau der Arbeit	2
2 <i>Grundlagen</i>	3
2.1 Internationale Norm IEC 61131-3.....	3
2.1.1 Programmiersprachen für SPS	3
2.1.2 Programmorganisationseinheiten (POEs).....	3
2.1.3 Variablen und Datentypen	4
2.2 Steuerungstechnisch interpretierte Petri Netze (SIPN).....	5
2.2.1 Stellen.....	5
2.2.2 Transitionen.....	5
3 <i>Anforderungsanalyse</i>	8
3.1 Nichtfunktionale Anforderungen	8
3.2 Funktionale Anforderungen	9
3.2.1 Allgemeine Anforderungen.....	9
3.2.2 SIPN zeichnen.....	10
3.2.3 Variablen	13
3.2.4 Fehlervermeidung	14
4 <i>Software design</i>	15
4.1 Die Softwarearchitektur.....	15
4.1.1 Model-View-Controller	15
4.1.2 Layer.....	16
4.1.3 Pipes and Filters.....	17
4.1.4 Auswahl der Architektur	17
4.2 Design konkreter Problemstellungen.....	19
4.2.1 Struktur des Modells.....	19
4.2.2 Exportieren des SIPN als Source Code	20
4.2.3 Modellierung des Petri Netzes	21
4.3 Die Programmiersprache	23
5 <i>Implementierung des Programms</i>	25
5.1 Der View	25
5.2 Das Modell.....	27
5.2.1 Variablen Modell.....	27
5.2.2 Das Zeichenmodell des SIPNs.....	28
5.3 Der Controller	28

5.3.1	Zeichnen der Stelle	29
5.3.2	Zeichnen der Transition.....	30
5.3.3	Zeichnen von gerichteten Kanten	32
5.3.4	Zeichnen von Betriebsköpfen.....	33
5.3.5	Einfügen von Kommentaren.....	34
5.3.6	Löschen von Zeichenobjekten.....	34
5.3.7	Verschieben von Zeichenobjekten	34
5.3.8	Variablenhandling	35
5.3.9	Drucken des SIPNs	36
5.3.10	Exportieren des SIPN als Bild	38
5.4	Fehlervermeidung	39
5.4.1	Variablenfehler	39
5.4.2	Zeichnungsfehler	40
5.4.3	Logische Fehler	40
6	Softwaretests	41
6.1	Testen der Variablen.....	41
6.2	Testen der Schaltbedingungen	42
6.3	Testen des Zeichnens.....	42
7	Validierung	47
8	Fazit	50
9	Ausblick	50
	Literaturverzeichnis	51
	A: SIPN der Pufferstrecke	53
	B: Datenträger	55

Abbildungsverzeichnis

ABBILDUNG 1: DARSTELLUNG DER STELLE.....	5
ABBILDUNG 2: BSP. STANDARDTRANSITION.....	7
ABBILDUNG 3: BSP. ZEITTRANSITION	7
ABBILDUNG 4: BSP. ANWEISUNGSTRANSITION.....	7
ABBILDUNG 5: BSP. LÖSCHTRANSITION.....	7
ABBILDUNG 6: BSP. ITERATIVES SCHALTEN.....	11
ABBILDUNG 7: KOMPONENTEN DER MVC-ARCHITEKTUR.....	16
ABBILDUNG 8: PIPES AND FILTER MUSTER.....	17
ABBILDUNG 9: SOFTWAREARCHITEKTUR	18
ABBILDUNG 10: FASSADENMUSTER	19
ABBILDUNG 11: STRATEGIEMUSTER FÜR DEN EXPORT.....	21
ABBILDUNG 12: KLASSENDIAGRAMM PETRI NETZ OBJEKTE.....	22
ABBILDUNG 13: KLASSENDIAGRAMM ZEICHENOBJEKTE.....	23
ABBILDUNG 14: BENUTZEROBERFLÄCHE	25
ABBILDUNG 15: KLASSENDIAGRAMM DER VIEW	26
ABBILDUNG 16: DARSTELLUNG DER PAKETE IM MODELL.....	27
ABBILDUNG 17: EINGABEMASKE FÜR DIE STELLE.....	29
ABBILDUNG 18: EINGABEMASKE FÜR DIE TRANSITION	30
ABBILDUNG 19: EINGABEMASKE FÜR DIE ZU LÖSCHENDEN STELLEN	30
ABBILDUNG 20: EINGABEMASKE FÜR DIE SCHALTVERZÖGERUNG	30
ABBILDUNG 21: EINGABEMASKE FÜR ANWEISUNGEN	30
ABBILDUNG 22: BERECHNUNG DER ANSCHLUSSPUNKTE	33
ABBILDUNG 23: BEISPIELHAFTER BETRIEBSKOPF	33
ABBILDUNG 24: DIALOGFELD VARIABLENDEKLARATION.....	36
ABBILDUNG 25: EXEMPLARISCHE REIHENFOLGE DER DRUCKBEREICHE	37
ABBILDUNG 26: BEISPIEL SCHLINGE	40
ABBILDUNG 27: TESTPROZESS	43
ABBILDUNG 28: TESTERGEBNIS: SOURCE CODE IN EINER TEXT DATEI.....	45
ABBILDUNG 29: PUFFER STRECKE.....	46

Tabellenverzeichnis

TABELLE 1: AUSWAHL ELEMENTARER DATENTYPEN NACH IEC 61131-3.....	4
TABELLE 2: VAR-SCHLÜSSELWÖRTER NACH IEC 61131-3.....	4
TABELLE 3: DARSTELLUNG DER VERSCHIEDENEN TRANSITIONSARTEN.....	7
TABELLE 4: NICHTFUNKTIONALE ANFORDERUNGEN	8
TABELLE 5: ANFORDERUNGEN BEDIENERFREUNDLICHKEIT	8
TABELLE 6: ALLGEMEINE FUNKTIONALE ANFORDERUNGEN.....	9
TABELLE 7: ANFORDERUNGEN BEI EXPORTIEREN IN CODE	10
TABELLE 8: ANFORDERUNG AN DAS ZEICHNEN DES SIPNS	10
TABELLE 9: ANFORDERUNG AN DIE STELLE	11
TABELLE 10: ANFORDERUNG AN DIE TRANSITION	12
TABELLE 11: ANFORDERUNG AN VERSCHIEDENE TRANSITIONSARTEN.....	12
TABELLE 12: ANFORDERUNGEN AN DIE ERWEITERTE SCHALTBEDINGUNG VON TRANSITIONEN	12
TABELLE 13: ANFORDERUNG AN GERICHTETE KANTE	12
TABELLE 14: ANFORDERUNG AN VARIABLEN	13
TABELLE 15: ANFORDERUNG AN VERSCHIEDENEN VARIABLENARTEN.....	13
TABELLE 16: ANFORDERUNG AN ELEMENTAREN DATENTYPEN	14
TABELLE 17: ANFORDERUNG ZUR FEHLERVERMEIDUNG.....	14
TABELLE 18: VOR UND NACHTEILE DER EINZELNEN DRUCKMETHODEN	37
TABELLE 19: TESTFÄLLE BEI DER ERWEITERTEN SCHALTBEDINGUNG.....	42
TABELLE 20: VALIDIERUNG DER ALLGEMEINEN ANFORDERUNGEN	47
TABELLE 21: VALIDIERUNG DER ZEICHNUNGSANFORDERUNGEN.....	48
TABELLE 22: VALIDIERUNG DER ANFORDERUNGEN AN DIE VARIABLEN.....	48
TABELLE 23: ANFORDERUNGEN AN DAS FEHLERHANDLING	49

Abkürzungsverzeichnis

AWL	Anweisungsliste
BMP	Windows Bitmap
GIF	Graphics Interchange Format
JPEG	Joint Photographic Experts Group
JVM	Java Virtual Machine
MVC	Model View Controller
PNG	Portable Network Graphics
POE	Programm-Organisationseinheit
SIPN	Steuerungstechnisch interpretiertes Petri Netz
SPS	Speicherprogrammierbare Steuerung
ST	Strukturierter Text
WBMP	Wireless Application Protocol Bitmap
XML	Extensible Markup Language

Glossar

Codesys: Herstellerunabhängige Entwicklungsumgebung für SPS nach IEC 61131-3 (Vgl. Codesys)

TIA-Portal: kurz für „Totally Integrated Automation“, Automatisierungssoftware der Firma Siemens (Vgl. Siemens)

1 Einleitung

Die Aufgabe ist, ein Hilfsmittel für die speicherprogrammierbare Steuerung (SPS) zur Automatisierung von Anlagenprozessen zu entwickeln. Dabei handelt es sich um eine Software für die Source Code Generierung von steuerungstechnisch interpretierten Petri Netzen (SIPN).

1.1 Einführung

Es gibt verschiedene Methoden und Beschreibungsmittel, um Anlagenprozesse zu automatisieren. Ein Beschreibungsmittel ist das Petri Netz. Petri Netze sind gerichtete Graphen. (Schnieder, 1999) Sie stellen die Zustände und die Übergänge in andere Zustände eines Prozesses dar. In der Steuerungstechnik wird das Petri Netz um eine Kommunikation mit der Umwelt des Systems erweitert. Das Petri Netz bekommt Eingänge und kann Ausgänge schalten. Diese spezielle Form von Petri Netzen werden steuerungstechnisch interpretierte Petri Netze (SIPN) genannt. (Aspern, 2003) Um den Prozess einer Anlage zu steuern, wird das SIPN in einen normgerechten Programmcode geschrieben, sodass dieser auf einer speicherprogrammierbaren Steuerung (SPS) laufen kann. Die Umwandlung erfolgt nach festen Codierungsregeln.

1.2 Ziel der Arbeit

Ziel der Arbeit ist, ein Werkzeug zu entwickeln, welches aus dem gezeichneten SIPN Source Code generiert. Der Source Code soll der internationalen Norm „IEC 61131-3“ entsprechen und als Funktionsblock in Codesys importiert werden können.

Zielgruppe des Werkzeuges sind Studierende und Professoren, die das Werkzeug nutzen können, um simple Petri Netze zu entwickeln und sich mit der Methode der steuerungstechnisch interpretierten Petri Netze vertraut zu machen. Der Anwender soll bei der Erstellung fehlerfreier SIPN unterstützt werden.

1.3 Motivation

Die Petri Netze werden entweder mit klassischen Zeichentools wie zum Beispiel „Microsoft Visio“ oder manuell mit dem Stift gezeichnet. Die Umwandlung in den Source Code für die SPS muss der Anwender selbst vornehmen. Dabei können Fehler entstehen, die zu einem Fehlverhalten der Anlage führen.

Je nach der Größe des steuerungstechnisch interpretierten Petri Netzes ist ein hoher Zeitaufwand für die konventionelle Umwandlung in den Source Code notwendig. Eine Software dafür könnte die Umwandlung innerhalb weniger Sekunden durchführen.

Derzeit werden die beim Testen der Steuerung aufgetretenen Fehler meist direkt im Source Code geändert. Dadurch stimmt das gezeichnete Petri Netz nicht mit der Steuerung überein. Wenn der Anwender die Möglichkeit hat, Änderungen im Petri Netz ohne großen Mehraufwand auf die Steuerung zu laden, kann das Petri Netz und der Source Code konsistent gehalten werden.

1.4 Aufbau der Arbeit

Zu Beginn der Arbeit werden die Grundkonzepte von SIPN erklärt und untersucht, welche Codierungsregeln für die Umwandlung gelten. Mit dem Wissen über das Konzept der SIPN, werden in Kapitel 3 Anforderungen an die Software gestellt. Im vierten Kapitel, dem Softwaredesign, werden anhand der Anforderungen die notwendigen Komponenten und deren Kommunikation untereinander entwickelt, aus denen sich die Software zusammensetzt. In dem darauffolgenden Kapitel wird die Implementierung einzelner Funktionen genauer erläutert. Mit Softwaretests werden grundlegende Funktionen getestet und überprüft, ob die Anforderungen erfüllt wurden. Danach folgt eine Validierung des Programms. Zum Schluss steht eine Zusammenfassung der Arbeit und ein Ausblick, wie das Programm weiterentwickelt werden könnte.

2 Grundlagen

An dieser Stelle wird beschrieben, wie ein SIPN aussieht und wie dieses in Source Code umgewandelt wird. Dafür wird die internationale Norm IEC 61131-3 betrachtet, die die Grundstruktur des Source Codes für die Steuerung beschreibt. Des Weiteren werden die einzelnen Objekte eines SIPN betrachtet und die geltenden Codierungsregeln dargelegt.

2.1 Internationale Norm IEC 61131-3

Die IEC 61131 beschreibt die Anforderungsdefinitionen für SPS-Systeme. (Tiegelkamp, et al., 2009) Die Norm besteht aus mehreren Teilen. Für die normgerechte Source Code Generierung ist der dritte Teil der Norm von Bedeutung.

„Dieser Teil der IEC 61131 legt die Syntax und Semantik von Programmiersprachen für speicherprogrammierbare Steuerungen (SPS) fest, wie sie in IEC 61131-1 definiert sind.“ (DIN EN 61131-3:2014-06)

2.1.1 Programmiersprachen für SPS

Laut der Norm werden dem Anwender fünf verschiedene Programmiersprachen angeboten: Ablaufsprache (AS), Anweisungsliste (AWL), Funktionsbausteinsprache (FUP), Kontaktplan (KOP) und Strukturierter Text (ST). (DIN EN 61131-3:2014-06)

Der Strukturierte Text (ST) ist eine textuelle Sprache, die der Programmiersprache Pascal ähnelt. (Petry, 2011) Diese Sprache gilt als Hochsprache und besitzt typische Anweisungen wie IF-Anweisungen und Schleifen. Die Anweisungsliste ist ebenfalls textbasiert und ähnelt einer Assembler Sprache. Die anderen Programmiersprachen sind graphische Sprachen. (Petry, 2011)

2.1.2 Programmorganisationseinheiten (POEs)

Um das SPS-Programm strukturieren zu können, werden dem Anwender vier Programmorganisationseinheiten zur Verfügung gestellt:

- 1) Das „Programm“ enthält alle POEs und Variablen, die für die Steuerung eines Prozesses notwendig sind. Dort werden zudem die Eingangsvariablen und die Ausgangsvariablen der SPS zugewiesen. (DIN EN 61131-3:2014-06)
- 2) Der „Funktionsbaustein“ dient zur Modularisierung des Programms und besitzt Ein- und Ausgangsvariablen und interne Variablen. Die internen Variablen werden gespeichert und bis zum nächsten Aufruf beibehalten. (DIN EN 61131-3:2014-06)
- 3) Die „Funktion“ ist ähnlich wie ein Funktionsbaustein, jedoch wird der Zustand nicht gespeichert. (DIN EN 61131-3:2014-06)

- 4) Die „Klasse“ dient als Unterstützung für die objektorientierte Programmierung. (DIN EN 61131-3:2014-06)

2.1.3 Variablen und Datentypen

Um Informationen zu speichern werden Variablen verwendet. Jede Variable besitzt einen festen Datentyp. In der IEC 61131-3 werden elementare Datentypen festgelegt. Beispielhaft sind einige dieser Datentypen in Tabelle 1 dargestellt.

Tabelle 1: Auswahl elementarer Datentypen nach IEC 61131-3

Beschreibung	Schlüsselwort
Boolesche	BOOL
Ganze Zahl	INT
Doppelte ganze Zahl	DINT
Vorzeichenlose ganze Zahl	UINT
Reelle Zahl	REAL
Zeitdauer	TIME
Datum (nur)	DATE
Bitfolge der Länge 8	BYTE
Bitfolge der Länge 16	WORD

Quelle: (Vgl. DIN EN 61131-3:2014-06)

Zusätzlich kann der Anwender eigene Datentypen definieren. (Vgl. Tiegelkamp, et al., 2009)
Die Deklaration einer Variablen erfolgt am Anfang einer Programmorganisationseinheit. Die Variablen werden je nach Verwendung innerhalb der entsprechenden „VAR-Sektion“ deklariert. Alle möglichen „VAR-Sektionen“ sind in Tabelle 2 dargestellt:

Tabelle 2: VAR-Schlüsselwörter nach IEC 61131-3

Schlüsselwort	Gebrauch der Variable
VAR	Innerhalb des Elements (Funktion, Funktionsbaustein, usw.)
VAR_INPUT	Von außen geliefert, nicht veränderbar innerhalb des Elements
VAR_OUTPUT	Vom Element an externe Elemente geliefert
VAR_IN_OUT	Von externen Elementen geliefert, kann innerhalb des Elements verändert werden und an ein externes Element geliefert werden
VAR_EXTERNAL	Geliefert von der Konfiguration über VAR_GLOBAL
VAR_GLOBAL	Globale Variablen-Deklaration
VAR_ACCESS	Zugriffspfad-Deklaration
VAR_TEMP	Temporärer Speicher für Variablen in Funktionsbausteinen, Methoden und Programmen
VAR_CONFIG	Instanz spezifische Initialisierung und Zuweisung der Ortsangabe
(END_VAR)	Beendet die verschiedenen oben genannten VAR-Sektionen

Quelle: (DIN EN 61131-3:2014-06)

Abhängig von der Art der Programmorganisationseinheit können verschiedene „VAR-Sektionen“ verwendet werden. Der Funktionsbaustein erlaubt die Verwendung von folgenden „VAR-Sektionen“: „VAR“, „VAR_INPUT“, „VAR_OUTPUT“, „VAR_IN_OUT“ und „VAR_TEMP“. Eine VAR-

Sektion wird mit dem Schlüsselwort begonnen und mit „END_VAR“ beendet. (Vgl. DIN EN 61131-3:2014-06)

2.2 Steuerungstechnisch interpretierte Petri Netze (SIPN)

SIPN bestehen aus Stellen und Transitionen. In der Literatur werden Stellen auch häufig als Plätze bezeichnet. Stellen und Transitionen werden Knoten genannt. Die Knoten des SIPN werden mit gerichteten Kanten verbunden.

2.2.1 Stellen

Stellen repräsentieren den Zustand des Prozesses. Der dargestellte Zustand kann entweder statisch oder dynamisch sein. Ein statischer Zustand liegt dann vor, wenn bestimmte Lagen erreicht wurden. Der dynamische Zustand beschreibt Bewegungen im Prozess. Ein Beispiel dafür kann sein, dass sich ein Motor bewegt oder steht. (Aspern, 2003)

Um den Zustand zu beschreiben, werden Marken eingesetzt. Ist eine Stelle mit einer Marke belegt, ist der Zustand den die Stelle repräsentiert aktiv. Um die Zustände außerhalb der POE nutzen zu können, oder um Aktoren anzusteuern, werden Ausgänge zu den entsprechenden Stellen zugewiesen. Jede Stelle kann mit keinem, einem oder mehreren Ausgängen verbunden sein. Ebenso kann ein Ausgang von einer oder mehreren Stellen gesteuert werden. (Aspern, 2003)

Die Stelle wird auf der SPS durch eine boolesche Variable repräsentiert. Die zugehörigen Stellen werden den Ausgängen in der Ausgangszuweisung, am Ende der POE zugewiesen. Falls ein Ausgang mehrere Stellen besitzt werden die Stellen mit einem „OR“ verknüpft. (Aspern, 2003)

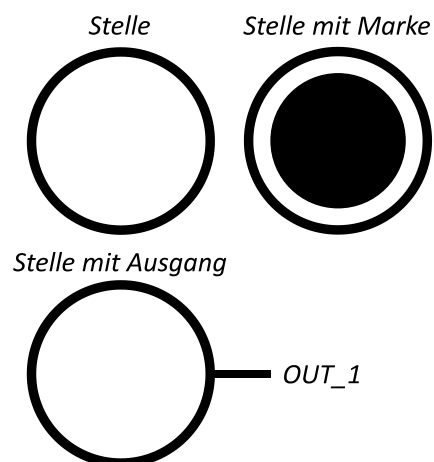


Abbildung 1: Darstellung der Stelle
Quelle: eigene Darstellung

2.2.2 Transitionen

Transitionen repräsentieren das dynamische Verhalten des Petri Netzes. Sie kontrollieren den Übergang vom aktuellen Zustand in den nächsten. Dies geschieht indem die Marke einer Stelle in eine andere Stelle geführt wird. Daher wird auch vom Markenfluss gesprochen.

Die Zustände werden geändert, falls die Schaltbedingungen erfüllt sind. Für eine erfüllte Schaltbedingung ist die Schaltbereitschaft Voraussetzung. Die Schaltbereitschaft ist gegeben, wenn

alle Eingangsstellen mit Marken belegt sind und alle Ausgangsstellen keine Marke besitzen, so dass diese mit Marken belegt werden können. Die Eingangsstellen werden mit Präkanten und die Ausgangsstellen werden mit Postkanten mit der Transition verbunden. (Lunze, 2012)

Die steuerungstechnische Interpretation ermöglicht es, die Schaltbedingung zu erweitern. Es können Signalpräkanten verwendet werden. Diese Kanten tragen nicht zum Markenfluss bei. Bei der Verwendung wird nur das Signal ausgewertet. Dabei wird zwischen zwei Signalkanten unterschieden, der Testkante und der Inhibitorkante. Die Testkante ist das direkte Signal, die Inhibitorkante ist das negierte Signal. (Aspern, 2003)

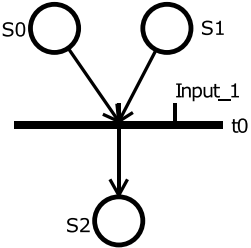
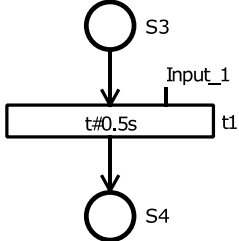
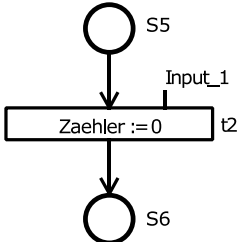
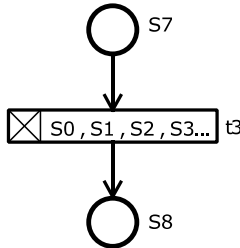
Eine weitere Möglichkeit ist eine Zusatzbedingung. Dabei muss die Transition neben der Schaltbereitschaft einen booleschen Ausdruck erfüllen. Dadurch können Eingangssignale, lokale Variablensignale und Stellensignale miteinander verglichen und logisch verknüpft werden. (Aspern, 2003)

Um komplexe Anlagenprozesse darstellen zu können, werden im steuerungstechnisch interpretierten Petri Netz verschiedene Transitionsarten eingeführt, die wiederkehrende Grundfunktionen darstellen. Die vier Transitionsarten, die während des Studiums am häufigsten verwendet wurden, sind die Standardtransition, die Zeittransition, die Anweisungstransition und die Löschransition. Diese werden im Folgenden noch einmal erläutert. Weitere Transitionsarten sind die Bedingungstransition und die Triggertransition. (Aspern, 2003)

Die Standardtransition kontrolliert nur den Markenfluss und enthält keine Zusatzfunktionen. Für die Verzögerungen im Netz werden Zeittransitionen eingesetzt. Die Zeittransition schaltet, falls die Schaltbedingung für eine bestimmte Dauer erfüllt ist. Die Anweisungstransition führt beim Schaltvorgang die vorgegebenen Anweisungen aus. Die Schaltbedingung entspricht die der Standardtransition. Die Löschransition dient zum Löschen von Marken aus Stellen. Die Codierung einer Löschransition ist mit einer Anweisungstransition zu vergleichen, dessen Anweisungen das Löschen von Stellen ist. Durch das Löschen von Marken aus Stellen können Anfangszustände wiederhergestellt werden. Die Bedingungstransition führt eine Fallunterscheidung durch und hilft den Markenfluss zu kontrollieren. Eine Bedingung ist auch immer eine Anweisung. Die Triggertransition führt einen Programmteil im Moment des Schaltens aus. (Aspern, 2003)

In Tabelle 3 sind die vier häufig verwendeten Transitionsarten mit beispielhaften Überführungen in normgerechten Source Code dargestellt.

Tabelle 3: Darstellung der verschiedenen Transitionsarten

Darstellung:	Zugehöriger Source Code:
<p>Standardtransition</p> 	<pre> IF (S0 AND S1 AND NOT S2) AND Input_1 THEN S0 := 0; S1 := 0; S2 := 1; END_IF </pre> <p><i>Schaltbereitschaft</i> <i>zusätzliche Schaltbedingung</i></p> <p>Markenfluss</p>
<p>Abbildung 2: Bsp. Standardtransition Quelle: eigene Darstellung</p> <p>Zeittransition</p> 	<pre> IF TON_t1.Q THEN S3 := 0; S4 := 1; END_IF TON_t1(IN := (S3 AND NOT S4) AND Input_1 ,PT := t#0,5s); </pre> <p><i>Schaltbereitschaft</i> <i>zusätzliche Schaltbedingung</i></p> <p>Markenfluss</p>
<p>Abbildung 3: Bsp. Zeittransition Quelle: eigene Darstellung</p> <p>Anweisungstransition</p> 	<pre> IF (S5 AND NOT S6) AND Input_1 THEN S5 := 0; S6 := 1; Zaehler := 0; END_IF </pre> <p><i>Schaltbereitschaft</i> <i>zusätzliche Schaltbedingung</i></p> <p>Markenfluss</p> <p>Anweisung</p>
<p>Abbildung 4: Bsp. Anweisungstransition Quelle: eigene Darstellung</p> <p>Löschtransition</p> 	<pre> IF (S7 AND NOT S8) THEN S7 := 0; S8 := 1; S0 := 0; S1 := 0; ... // alle Stellen die gelöscht werden sollen END_IF </pre> <p><i>Schaltbereitschaft</i></p> <p>Markenfluss</p> <p>Löschen von Stellen</p>
<p>Abbildung 5: Bsp. Löschtransition Quelle: eigene Darstellung</p>	

3 Anforderungsanalyse

Im Folgenden wird analysiert, welche Anforderungen an das Programm gestellt werden, um das Ziel der Arbeit zu erfüllen. Dabei wird zwischen nichtfunktionalen und funktionalen Anforderungen unterschieden. Die verschiedenen Ziele werden mit Prioritäten bewertet. Notwendige Aspekte für die fehlerfreie Nutzung des Programms werden mit „++“ bewertet. Das einfache „+“ bedeutet, dass diese Punkte hilfreich sind, um dem Anwender die Bedienung zu erleichtern. Die Priorität „o“ stellt meistens Anforderungen dar, die nur wenig Bedeutung haben. Jedoch wird darauf geachtet, dass eine nachträgliche Implementierung möglich ist.

3.1 Nichtfunktionale Anforderungen

Da jedem Studierenden ermöglicht werden soll, die Software auf dem eigenen Gerät zu nutzen, ist die Plattformunabhängigkeit wichtig. Das Programm sollte so aufgebaut werden, dass einzelne Funktionen nachträglich hinzugefügt werden können, falls in dieser Arbeit nicht alle Funktionen umgesetzt werden können.

Tabelle 4: Nichtfunktionale Anforderungen

ID	Anforderung	Priorität
A1	Plattformunabhängigkeit	++
A2	Erweiterbarkeit	++
A3	Wartbarkeit	+
A4	Zuverlässigkeit	++
A5	Bedienerfreundlichkeit	++

Quelle: eigene Darstellung

Die Zuverlässigkeit ist ebenfalls von großer Bedeutung. Funktioniert das Tool nicht einwandfrei, ist es wahrscheinlich, dass die konventionellen Werkzeuge verwendet werden. In diesem Fall würde sich kein Vorteil für die Verwendung dieses Tools ergeben.

Tabelle 5: Anforderungen Bedienerfreundlichkeit

ID	Anforderung	Priorität
A5.1	Schnell zu bedienen	++
A5.2	Einfach zu bedienen	++
A5.3	Kurze Antwortzeit	+
A5.4	Intuitiv zu bedienen (Anwender lernt die Bedienung schnell und ohne externe Hilfe)	++

Quelle: eigene Darstellung

Damit den Studierenden der Einstieg in die Software gut gelingt, muss das Programm bedienerfreundlich gestaltet sein. Neben dem Aussehen der Bedienoberfläche muss der Anwender das Programm intuitiv bedienen können. Einige Studierende werden dieses Werkzeug während des

Studiums nicht länger als drei Monate verwenden, daher sollten diese ohne Bedienungsanleitungen zurecht kommen. Um die Zeitersparnis zu behalten, die durch die automatische Code Generieren gewonnen wird, muss der Anwender das SIPN schnell zeichnen können.

3.2 Funktionale Anforderungen

In den funktionalen Anforderungen wird genau untersucht, welche Funktionen das Programm erfüllen muss. Dafür werden die Anforderungen zunächst allgemein formuliert und untersucht, was benötigt wird, um diese Anforderungen zu erfüllen.

3.2.1 Allgemeine Anforderungen

Funktional sind insgesamt drei verschiedene Hauptbereiche zu implementieren. Das Variablenhandling, das Fehlerhandling und die Möglichkeit das SIPN zu zeichnen und das gezeichnete SIPN in normgerechten Code umzuwandeln. Beim Variablenhandling und beim Zeichnen gilt es die Herausforderung zu bewältigen, die Komplexität der Benutzereingaben gering zu halten und dem Anwender dennoch Möglichkeit zu geben, das Werkzeug für fast alle Anwendungsfälle nutzen zu können. Um das Fehlerhandling zu betreiben, muss festgelegt werden, was dem Anwender erlaubt wird und was nicht.

Tabelle 6: Allgemeine funktionale Anforderungen

ID	Anforderung	Priorität
B1	Erstellen eines neuen SIPNs	++
B2	Zeichnen des SIPNs	++
B3	Bearbeiten des SIPNs	++
B4	SIPN speichern	+
B5	SIPN laden	+
B6	SIPN drucken	+
B7	Exportieren des SIPN als Bild	+
B8	Exportieren des SIPN in normgerechten Code	++
B9	Normgerechten Code Importieren und SIPN erzeugen	0
B10	Heran- und Herauszoomen des SIPNs	+
B11	Simulation des SIPNs	0
B12	Lebendigkeit des SIPNs prüfen	0
B13	Erkennen und darstellen hierarchischer Strukturen	0
C1	Erstellen/bearbeiten/löschen von Variablen	++
D1	Warnung bei Fehlerhafte Eingabe	+

Quelle: eigene Darstellung

Durch Simulationen des SIPNs oder das Erkennen und Darstellen von hierarchischen Strukturen kann das Programm erweitert werden.

Während des Zeichnens müssen die verwendeten Variablen stetig erweitert und angepasst werden. Zur Nutzung des Petri Netzes aus einer Dokumentation sollte die Zeichnung weiterverwendet werden können. Dies ist möglich bei gedruckten oder als Bild gespeicherten Zeichnungen.

Ziel ist es, beim Exportieren des SIPN in Source Code den Zeitaufwand für die Weiterverwendung gering zu halten. Dies kann dadurch realisiert werden, dass das SIPN direkt in die Entwicklungsumgebung importiert werden kann. Für das Exportieren ist die POE des Funktionsbausteins gut geeignet. Mit dieser kann das Programm gut strukturiert werden. Im Gegensatz zu der POE „Funktion“ werden die Zustände beibehalten. Die Entwicklungsumgebungen wie Codesys und TIA-Portal besitzen verschiedene Schnittstellen, um POEs zu importieren. Daher muss für jede Entwicklungsumgebung eine separate Datei erstellt werden können. Codesys ist frei verfügbar und daher gut für Studierende geeignet. Dementsprechend wird die Priorität einer funktionierenden Exportfunktion für Codesys höher eingestuft. Eine mögliche Erweiterung um TIA ist vorzusehen.

Tabelle 7: Anforderungen bei exportieren in Code

ID	Anforderung	Priorität
B8.1	Exportieren des SIPNs in Codesys als Funktionsblock in ST	++
B8.2	Exportieren des SIPNs in Codesys als Funktionsblock in AWL	o
B8.3	Exportieren des SIPNs in einer Text Datei	++
B8.4	Exportieren des SIPNs in TIA-Portal als Funktionsblock in ST	o
B8.5	Exportieren des SIPNs in TIA-Portal als Funktionsblock in AWL	o

Quelle: eigene Darstellung

3.2.2 SIPN zeichnen

Um das SIPN in funktionierenden Source Code umwandeln zu können, ist es besonders wichtig, dass beim Zeichnen des SIPNs alle notwendigen Objekte (Stellen, Transitionen, Kanten) gezeichnet werden können. Hinzukommt, dass die Zeichnung auch später noch verändert werden kann. Das Einfügen von Kommentaren ist nicht notwendig, kann aber die Übersichtlichkeit und das Verständnis des gezeichneten SIPN verbessern.

Tabelle 8: Anforderung an das Zeichnen des SIPNs

ID	Anforderung	Priorität
B2.1	Stelle hinzufügen	++
B2.2	Transition hinzufügen	++
B2.3	Gerichtete Kante hinzufügen	++
B2.4	Kommentar hinzufügen	+
B2.5	Verschieben einzelner Objekte	++
B2.6	Verschieben mehrerer Objekte	+
B2.7	Kopieren von Objekten	+
B2.8	Löschen von einzelnen Objekten	++
B2.9	Löschen von mehreren Objekten	+

Quelle: eigene Darstellung

3.2.2.1 Stelle

Beim Zeichnen der Stelle müssen alle relevanten Eigenschaften für das SIPN gegeben sein. Jede Stelle besitzt einen Namen. Dieser muss einzigartig sein, da die Stelle im Source Code mit einer Variablen repräsentiert wird. Initialstellen werden mit Marken belegt. Falls die Stelle einem oder mehreren Ausgängen zugewiesen wird, müssen diese visuell dargestellt werden. Um die Übersicht zu verbessern, sollte dem Anwender die Möglichkeit gegeben werden, die Position der Ausgangsvariablen an der Stelle zu bestimmen.

Tabelle 9: Anforderung an die Stelle

ID	Anforderung	Priorität
B2.1.1	Name zuweisen/ändern	++
B2.1.2	Marke hinzufügen/entfernen	++
B2.1.3	Beliebig viele Ausgänge hinzufügen/ändern/löschen	++
B2.1.4	Position der Ausgangsvariablen an der Stelle ändern	+

Quelle: eigene Darstellung

3.2.2.2 Transition

Der Transitionsname muss ebenfalls eindeutig sein. Der Grund ist, dass der Name für die Position im Source Code des SPS-Programms verantwortlich ist. In der Regel bearbeitet das SPS Programm den Source Zeile für Zeile. Dadurch hat die Reihenfolge Einfluss auf das iterative Schalten der Transitionen. (Vgl. Aspern, 2003)

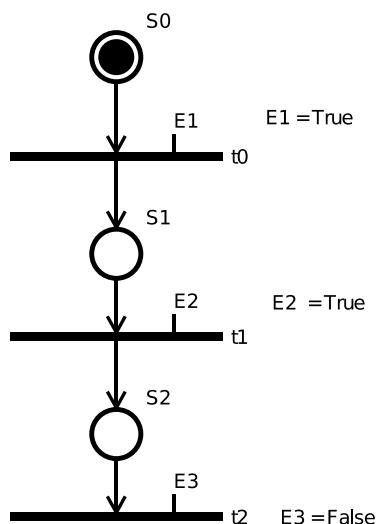


Abbildung 6: Bsp. iteratives Schalten
Quelle: eigene Darstellung

Nach dem iterativen Schalten würde die Marke in Abbildung 6 direkt in die Stelle S_2 springen, da die Transition t_1 die Marke direkt durchschaltet. Denn bei der Bewertung der Transition t_1 ist die Marke in die Stelle S_1 weitergerückt, daher ist die Schalbereitschaft vorhanden.

Ist die Reihenfolge der Transitionen im Programmcode vertauscht (t_1 vor t_0) wird zunächst die Transition t_1 bewertet. Die Stelle S_1 ist in dem Zeitpunkt noch nicht belegt, daher schaltet die Transition nicht. Am Ende des Zyklus ist die Marke bei der Stelle S_1 .

Die sequenzielle Reihenfolge bestimmt auch über Vorränge einzelner Transitionen. Besitzen zwei Transitionen denselben Vorplatz und könnten beide schalten, schaltet nur die Transition, die von der SPS zuerst abgearbeitet wurde.

Das Bearbeiten der Schaltbedingung und der Transitionsarten muss mit den spezifischen Anforderungen realisiert werden.

Tabelle 10: Anforderung an die Transition

ID	Anforderung	Priorität
B2.2.1	Name zuweisen/ändern	++
B2.2.2	Priorisierung der Transitionen verändern	++
B2.2.3	Drehung verändern	+
B2.2.4	Transitionsart verändern	++
B2.2.5	Schaltbedingung hinzufügen/ändern/löschen	++

Quelle: eigene Darstellung

Die Trigger Transition und die Bedingungstransition werden nicht so häufig verwendet. Außerdem ist die Bedingungstransition ebenfalls eine Anweisungstransition. Daher werden diese Punkte nicht mit höchster Priorisierung betrachtet.

Tabelle 11: Anforderung an verschiedene Transitionsarten

ID	Anforderung	Priorität
B2.2.4.1	Standardtransition	++
B2.2.4.2	Anweisungstransition Eingabe der Anweisungen	++
B2.2.4.3	Zeittransition mit Eingabe der Zeit	++
B2.2.4.4	Löschtransition mit Auswahl der zu löschenden Stellen	++
B2.2.4.5	Trigger Transition mit Eingabe des getriggerten Netzelement	+
B2.2.4.6	Bedingungstransition mit Eingabe der Bedingung	+

Quelle: eigen Darstellung

Die erweiterte Schaltbedingung muss ein boolesches Ergebnis liefern. Dies ist möglich mit einer Booleschen Verknüpfung (AND, OR, XOR, NOT) oder einem Vergleich (<, >, >=, <=, ==, <>). Zusätzlich kann die Schaltbedingung Signalkanten enthalten. (Vgl. Kapitel 2.2.2)

Tabelle 12: Anforderungen an die erweiterte Schaltbedingung von Transitionen

ID	Anforderung	Priorität
B2.2.5.1	Boolesche Verknüpfung	++
B2.2.5.2	Inhibitor-/Testkanten	++
B2.2.5.3	Vergleichsoperationen	++

Quelle: eigene Darstellung

3.2.2.3 Gerichtete Kante

Die gerichteten Kanten (Post- und Präkante) sind die Verbindungsstücke der Petri Netz Knoten. Um die Übersichtlichkeit zu verbessern ist es hilfreich, den graphischen Verlauf der Kante zu bestimmen, indem Zwischenpunkte gesetzt werden. Diese Zwischenpunkte sollten einzeln bewegt werden können.

Tabelle 13: Anforderung an gerichtete Kante

ID	Anforderung	Priorität
B2.3.1	Verbinden von zwei Petri Netz Knoten	++
B2.3.2	Zwischenpunkte hinzufügen/bearbeiten	+

Quelle: eigene Darstellung

3.2.3 Variablen

Die verwendeten Variablen sollten während des Zeichnens deklariert werden. Für die Übersicht sollten alle bereits deklarierten Variablen in der Benutzeroberfläche angezeigt werden. Um die Bedienbarkeit zu verbessern, können die Variablen, die aufgrund der Position im SIPN fest definiert sind, automatisch erstellt werden. Beispiele sind die Stellennamen, die Lokal als „BOOL“ angelegt werden und die Ausgänge, die mit einer Stelle verknüpft sind. Diese werden als Ausgangsvariable vom Typ „BOOL“ deklariert.

Eine weitere Möglichkeit wäre das automatische Erstellen von Variablen, falls diese eine bestimmte Namenskonvention für die Art und den Typ besitzen.

Tabelle 14: Anforderung an Variablen

ID	Anforderung	Priorität
C1	Anzeigen aller verwendeten Variablen	+
C2	Automatisches Erstellen von Variablen (z.B. Stellenname als Lokalen Boolean)	+
C3	Automatisches Erstellen von Variablen nach Namenskonvention	o
C4	Verwenden verschiedener Variablenarten	++
C5	Verwenden verschiedener Variablentypen	++

Quelle: eigene Darstellung

Die verschiedenen Variablenarten, die verwendet werden können, entsprechen den Arten, die in der IEC 61131-3 definiert wurden. Die Variablen „GLOABL“, „ACCESS“ und „CONFIG“ sind im Funktionsbaustein nicht erlaubt und nur dann relevant, wenn das SIPN in eine andere POE exportiert werden soll.

Die lokalen Variablen sind notwendig, da diese für die Stellen benötigt werden. Für die Kommunikation mit dem Umfeld des Systems werden die Eingangs- und Ausgangsvariablen benötigt.

Tabelle 15: Anforderung an verschiedenen Variablenarten

ID	Anforderung	Priorität
C4.1	LOKAL	++
C4.2	INPUT	++
C4.3	OUTPUT	++
C4.4	IN_OUT	+
C4.5	EXTERNAL	+
C4.6	GLOBAL	o
C4.7	ACCESS	o
C4.8	TEMP	+
C4.9	CONFIG	o

Quelle: eigene Darstellung

Wichtige Datentypen sind der BOOL Datentyp für die Stellen und der „TIME“ Datentyp für die Zeittransition. Mindestens ein Datentyp sollte eine Zahl repräsentieren können. Somit können die

meisten Prozesse beschrieben werden. Beim Aufbau sollte die Verwendung weiterer Datentypen einfach integriert werden können.

Auf die Möglichkeit eigene Datentypen zu definieren, wird in dieser Arbeit nicht eingegangen, da die elementaren Datentypen für die meisten Anwendungsfälle ausreichen.

Tabelle 16: Anforderung an elementaren Datentypen

ID	Anforderung	Priorität
C5.1	BOOL	++
C5.2	INT	++
C5.3	DINT	+
C5.4	UINT	++
C5.5	REAL	+
C5.6	TIME	++
C5.7	DATE	+
C5.8	BYTE	+
C5.9	WORD	+

Quelle: eigene Darstellung

3.2.4 Fehlervermeidung

Bevor das SIPN als Funktionsbaustein exportiert werden kann, muss geprüft werden, ob der entstehende SPS-Code fehlerfrei ist. Dafür muss die Software mögliche Fehler, die während des Zeichnens entstehen, erkennen und beheben können.

Die gerichteten Kanten können entweder eine Post- oder eine Präkante sein, wodurch Verbindungen von Stelle zu Stelle oder von Transition zu Transition nicht möglich sind.

Wie bereits in Abschnitt 3.2.2 erwähnt muss der Stellenname und der Transitionsname jeweils einzigartig sein.

Für die SPS muss der Variablentyp und die Variablenart in ihrer Verwendung übereinstimmen. Jedoch liegt die Entscheidung im Wesentlichen beim Anwender, welche Ein- und Ausgänge der Funktionsblock besitzen soll und welche lokalen Variablen angelegt werden. Die Variablenverwendung sollte dennoch bestmöglich überprüft werden. Beispielsweise kann überprüft werden, ob der Wertebereich eingehalten wird.

Tabelle 17: Anforderung zur Fehlervermeidung

ID	Anforderung	Priorität
D1.1	Kanten können nur von Transition zu Stelle oder von Stelle zu Transition gerichtet werden	++
D1.2	Prüfen, ob die Stellennamen einzigartig sind	++
D1.3	Prüfen, ob die Transitionsnamen einzigartig sind	++
D1.4	Variablentyp und Variablenart passend bei der jeweiligen Verwendung	+
D1.5	Das Petri Netz auf Schlingen überprüfen	+
D1.6	Das Petri Netz besitzt mindestens eine Initialstelle	++

Quelle: eigene Darstellung

4 Softwaredesign

In diesem Abschnitt wird die Architektur des Programms aus den Anforderungen abgeleitet und dargestellt wie die einzelnen Komponenten und Schnittstellen gewählt werden, um die ermittelten Anforderungen zu erfüllen.

Zunächst wird die Architektur passend zum Anwendungsfall entworfen. Danach werden konkrete Designentscheidungen getroffen.

4.1 Die Softwarearchitektur

In der Softwarearchitektur werden Strategien ausgewählt, die beschreiben, wie ein System konstruiert wird. (Goll, 2017) Innerhalb der Architektur werden die Komponenten, deren Schnittstellen und Beziehungen zueinander beschrieben. (Hasselbring, 2006) Nachdem eine Architektur gewählt wurde, werden die Bausteine eines Systems entwickelt, die für die Anwendung benötigt werden.

4.1.1 Model-View-Controller

Eine häufig verwendetes Architekturmuster ist der Model-View-Controller. Dieses Musters wird bei interaktiven Systemen angewendet. Es wurde in den späten 70er Jahren von Trygve Reenskaug entwickelt. (Goll, et al., 2013) Der Grundgedanke ist die Zerlegung der Software in drei Komponenten, dem Model, der View und dem Controller.

„Das **Model** umfasst die Kernfunktionalität und kapselt die Verarbeitung und die Daten des Systems.“ (Goll, et al., 2013)

„Eine **View** stellt die Daten für den Benutzer dar. Sie erhält die darzustellenden Daten vom Model.“ (Goll, et al., 2013)

„Ein **Controller** ist für die Entgegennahme der Eingaben des Benutzers und ihre Interpretation verantwortlich.“ (Goll, et al., 2013)

Dabei wird jeder Controller und jeder View einem Modell zugeordnet. Jedes Model kann durch ein oder mehrere Views dargestellt werden. Dafür wird das Modell unabhängig von der Benutzeroberfläche entwickelt. Das bringt den Vorteil, dass der View geändert werden kann, ohne das Model anpassen zu müssen. Die Abhängigkeiten sind in Abbildung 7 dargestellt.

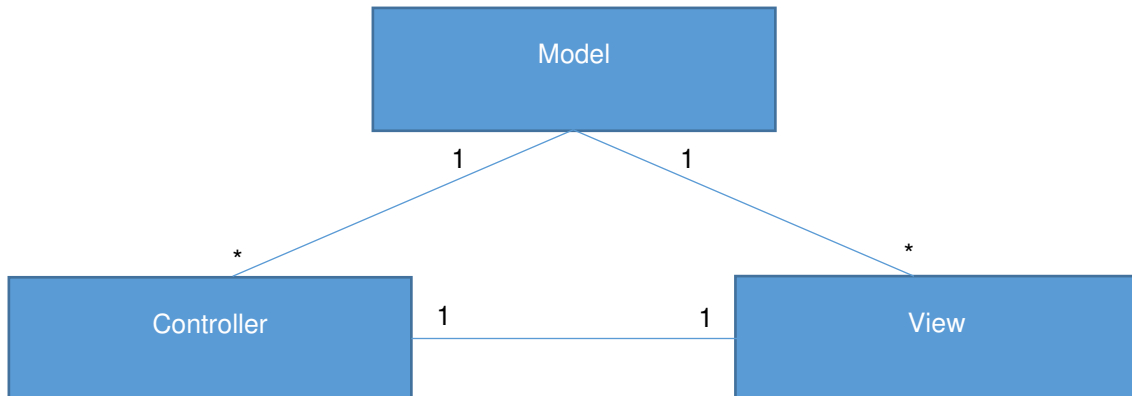


Abbildung 7: Komponenten der MVC-Architektur
Quelle: eigene Darstellung

Häufige Aktualisierungen der Ansicht können zu Performanceverlusten führen. (Goll, et al., 2013) Bei dem Zeichnen wird das Modell häufig angepasst, dementsprechend wird die Ansicht in der View stetig aktualisiert. Bei der Verwendung dieses Architekturmusters sind Lösungen zu finden, um mögliche Performanceverluste zu reduzieren. Weiter ist darauf zu achten, dass beim Ändern des Modells bei der Verwendung mehrerer verschiedenen Ansichten, all diese Darstellungen konsistent gehalten werden müssen. (Eilbrecht, et al., 2019)

4.1.2 Layer

„Das Layer-Muster beschreibt das am meisten verbreitete Prinzip, um auf Architekturebene ein System aufzuteilen.“ (S.30, Buschmann, et al., 1998) Im Architekturmuster Layer (dt. Schicht) werden Schichten gebildet. Dies „[...] dient zur Strukturierung eines Systems in Teilsysteme [...]“ (Eilbrecht, et al., 2019). Jede Schicht enthält eine Unteraufgabe des Systems. Für das Layer Muster gilt das Client-Server-Prinzip. Das Client-Server-Prinzip baut darauf auf, dass es eine Komponente gibt, die Dienste anbietet (Server) und eine Komponente, die diese Dienste in Anspruch nimmt (Clients). Die Clients bekommen eine Antwort von den gefragten Diensten zurück. (Schaller, 2017) Daher spricht man hier von serviceorientierter Programmierung. Ziel bei der Anwendung dieses Architekturmusters ist, die Abhängigkeiten einzelner Komponenten einzuschränken und die Komplexität des Systems zu gliedern. (Goll, et al., 2013)

Eine Schicht ist von der höheren Schicht unabhängig und kann auf die untere Schicht zugreifen. Für die höhere Schicht werden Dienste angeboten. Jede Schicht kann mehrere Komponenten besitzen. Für die Zugriffsmöglichkeiten auf die unteren Schichten werden Schnittstellen definiert. (Goll, 2017) Beim „Strict Layering“ darf eine Schicht nur auf die direkt benachbarte Subschicht zugreifen. Beim „Layer Bridging“ darf eine Schicht auf alle Subsysteme zugreifen. (Goll, 2017)

Wesentlicher Vorteil dieses Musters ist, dass die Abhängigkeiten zwischen den einzelnen Subsystemen minimiert werden können. (Balzert, 2011) Die Schicht ist nur von der Schnittstelle abhängig, nicht von der Implementierung der einzelnen Subsysteme. Zudem kann eine Schicht mit

einer anderen Schicht ausgetauscht werden, sofern diese die gleichen Methoden und Zugriffsmöglichkeiten anbietet. (Balzert, 2011)

Anfragen werden von Schicht zu Schicht weitergereicht. Das kann ähnlich wie beim MVC Muster zu Performanceverlust führen. (Goll, 2017) Bei der Erweiterung und Änderungen eines Programms mit einem Layer Architekturmuster können mehrere Schichten von der Änderung betroffen sein, was zu Mehraufwand führt. (Goll, 2017) Bei der Verwendung ist es daher von Bedeutung, eine passende Anzahl an Schichten festzulegen. (Balzert, 2011)

4.1.3 Pipes and Filters

Bei dem Pipes and Filters Muster wird eine Anfrage in mehreren Arbeitsschritten aufgeteilt. Daraus ergibt sich eine Struktur mit Filtern, die einen Teilschritt darstellen und Rohrleitungen, die die Teilergebnisse weiterleiten.



Abbildung 8: Pipes and Filter Muster
Quelle: (Eilbrecht, et al., 2019)

Ähnlich wie beim Layer Muster wird das System in Komponenten oder Arbeitsschritten aufgeteilt. Im Gegensatz zum Layer Muster ist das Pipes and Filters Muster datenstromorientiert und nicht serviceorientiert. Das bedeutet, dass die Ausgabe eines Filters, die Eingabe des nächsten Filters ist. (Goll, 2017)

Der Vorteil dieser Struktur ist, dass einzelne Filter schnell ausgetauscht werden können und dadurch das Programm leicht zu warten ist und schnell erweitert werden kann.

4.1.4 Auswahl der Architektur

Die beschriebenen Softwarearchitekturen werden mit den Anforderungen verglichen. Das Petri Netz Modell besteht seit 1963 (Aspern, 2003). Seitdem hat sich das Modell nur wenig verändert. Es ist davon auszugehen, dass das Modell nur selten angepasst werden muss. Die Benutzeroberfläche und der Controller werden bei der Weiterentwicklung häufiger verändert. Eine denkbare Weiterentwicklung ist die innerhalb des Kapitels 3.2.1 beschriebene Anforderung (B14): Darstellung von hierarchischen Strukturen. In diesem Fall müsste lediglich die Komponente, die für der View zuständig ist, geändert werden. Je größer die Zeichnung wird desto größer werden die Latenzzeiten. Da jedoch meist kleine bis mittelgroße SIPN mit dieser Software erstellt werden, ist der mögliche Performanceverlust bei großen Zeichnungen nicht entscheidend.

Mit dem Layer Muster können ähnliche Ergebnisse erzielt werden. Um das zu erreichen, könnten drei Schichten eingeführt werden. Jede Schicht übernimmt dabei eine Funktion des MVC-

Mustern. Dabei entsteht allerdings kein Vorteil gegenüber dem MVC-Muster. Im MVC-Muster die Kommunikation zwischen den einzelnen Komponenten simpler.

Da die Pipes und Filters Architektur datenorientiert ist, wird dieses Architekturmuster nicht verwendet. Der Zugriff soll serviceorientiert erfolgen. Dadurch können einzelne Anwenderfunktionen schneller implementiert werden.

Weil sich das MVC-Muster gut für diesen Anwendungsfall eignet, baut die Software auf diesem Muster auf. Ein Lösungsansatz für die Erhöhung der Performance wird in Abschnitt 5.1 beschrieben. In Abbildung 9 ist die Architektur dargestellt.

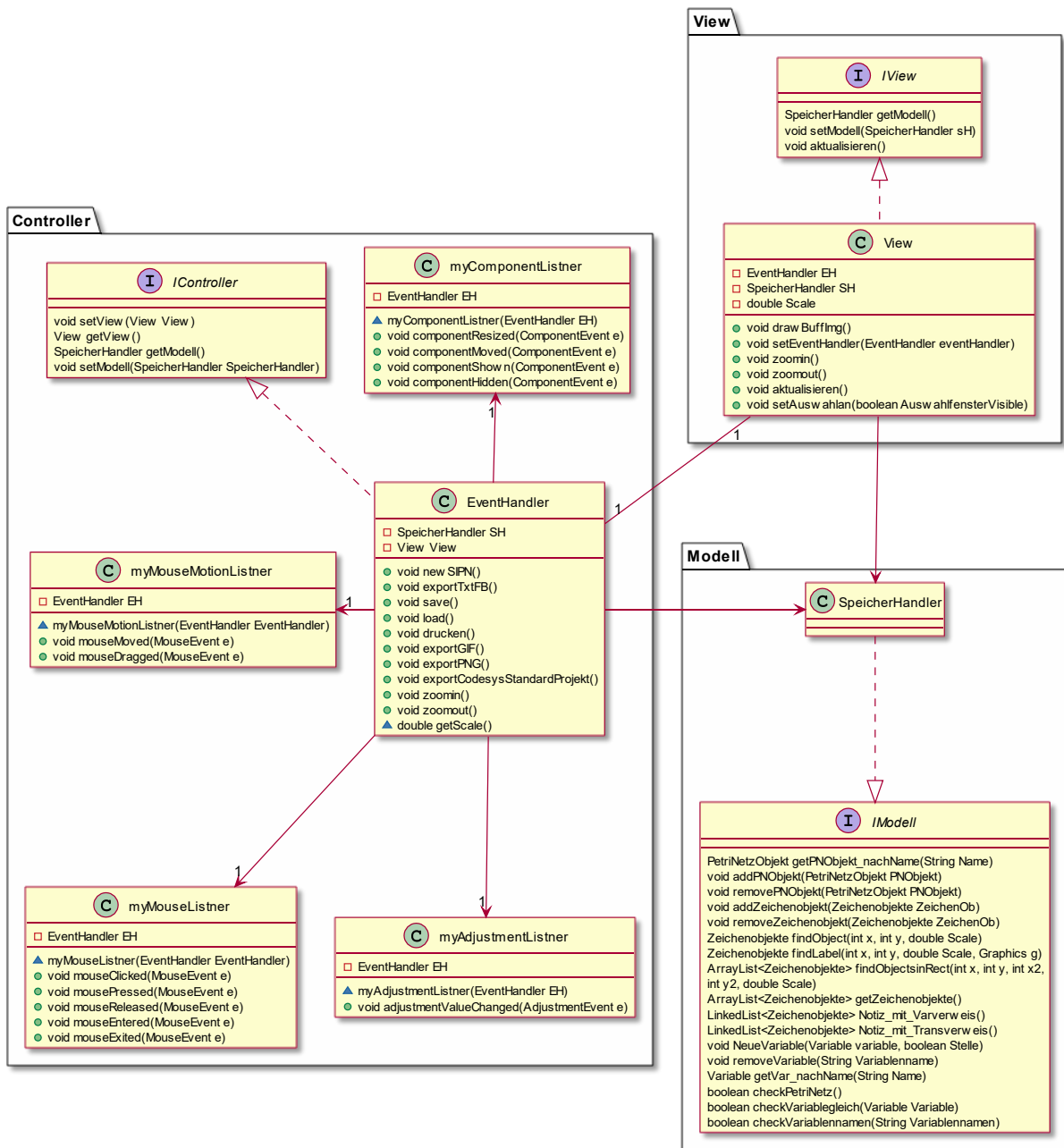


Abbildung 9: Softwarearchitektur
 Quelle: eigene Darstellung

4.2 Design konkreter Problemstellungen

Mit den drei Komponenten, die aus der Erstellung der Softwarearchitektur entstanden sind, lassen sich noch nicht alle Anforderungen erfüllen. Die Probleme für das Erfüllen der Anforderungen werden im folgenden Abschnitt konkreter beschrieben.

4.2.1 Struktur des Modells

Durch das Verwenden des MVC-Musters entstehen drei Komponenten. Für die Übersicht und die Wartbarkeit ist es sinnvoll, weitere Subsysteme innerhalb dieser Komponenten einzuführen. Die Verwendung vieler Subsysteme erschweren jedoch den Zugriff auf die Komponenten, da der Client viele Details des Subsystems kennen muss. Es ist eine Lösung zu erarbeiten, die der Komponente eine einfache Schnittstelle gibt, um auf die Subsysteme zuzugreifen.

Eine bekanntes Entwurfsmuster dafür ist das Fassadenmuster. (Eilbrecht, et al., 2019) Hierbei wird eine zusätzliche Klasse erstellt, die den Zugriff auf die Subsysteme kapselt. Die Fassadenklasse greift auf die einzelnen Subsysteme zu. Der Client muss nur noch über diese zusätzliche Klasse kommunizieren. (Eilbrecht, et al., 2019)

Bei dem Fassadenmuster muss beachtet werden, dass der Client die Fassadenklasse nicht umgeht und direkt auf das Subsystem zugreift. Dadurch würde die Fassade wertlos werden. (Eilbrecht, et al., 2019) Die Fassade ist in Abbildung 10 dargestellt. Die Fassadenklasse wird durch die Klasse „SpeicherHandler“ dargestellt.

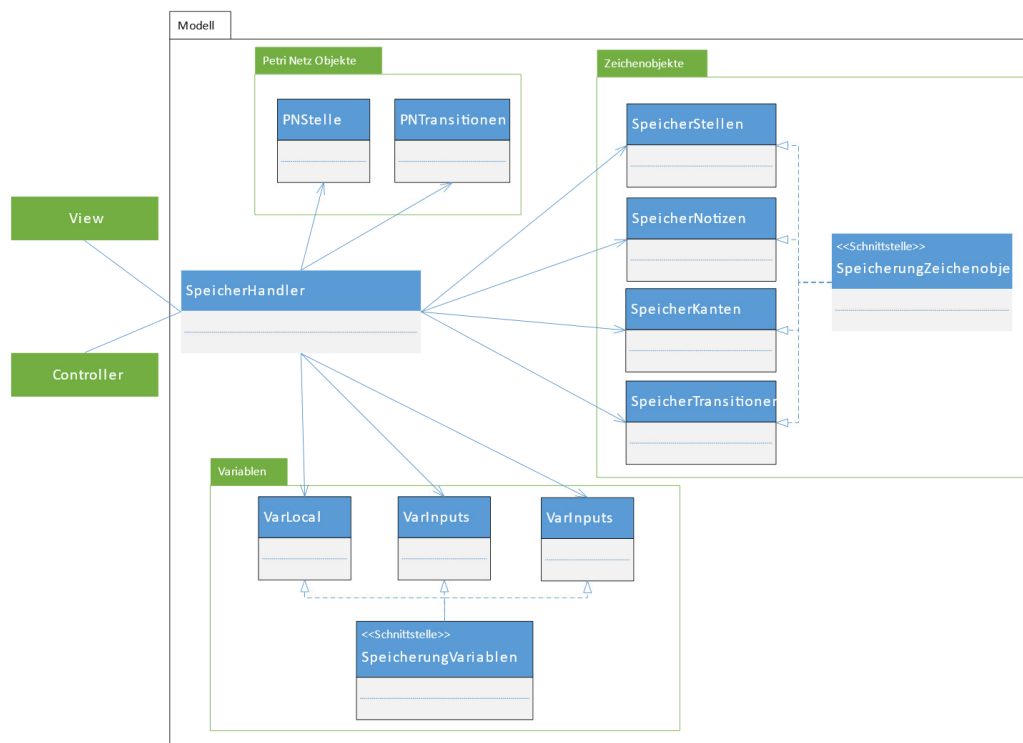


Abbildung 10: Fassadenmuster
Quelle: eigene Darstellung

4.2.2 Exportieren des SIPN als Source Code

Wie bereits in Kapitel 3.2.1 erwähnt, bieten die unterschiedlichen Entwicklungsumgebungen für das Programmieren von SPS verschiedene Schnittstellen an, um POEs zu importieren. Häufig wird das Extensible Markup Language (XML) Format verwendet. Ein für den Anwender besser lesbares Format ist eine Text Datei.

Für jede Schnittstelle werden unterschiedliche Dateien benötigt, die das gleiche SIPN beschreiben. Nun gilt es eine Lösung zu finden, die es ermöglicht, die notwendigen Export-Formate zu erstellen. Dabei ist wichtig, dass dies ohne großen Aufwand geschieht, damit weitere Formate schnell implementiert werden können. Dabei ist zu beachten, dass alle wichtigen Inhalte (Variablendeklaration, Codierung der Transitionen, Ausgangsdeklaration) vorhanden sind. Dafür werden zwei Entwurfsmuster analysiert.

Als erstes wird das Schablonenmuster betrachtet. In diesem Muster wird die Struktur eines Algorithmus definiert. (Eilbrecht, et al., 2019) Die Unterklassen verwenden diese definierte Struktur und legen die Art und Weise der Implementierung der Methoden fest. Dadurch wird die Abfolge mehrere Arbeitsschritte für verschiedene Klassen festgelegt. Die Schablone ist nicht abhängig von ihren Unterklassen. Das hat den Vorteil, dass beim Erstellen neuer Unterklassen, die Schablone nicht angepasst werden muss. (Goll, et al., 2013)

Durch das Verwenden des Schablonenmusters kann die feste Struktur der Codierung festgelegt werden. Der erste Arbeitsschritt wäre die Variablendeklarationen für die unterschiedlichen Variablenarten. Der folgende Arbeitsschritt ist dafür verantwortlich, die Transitionen zu codieren. Abgeschlossen wird die Schablone mit der Ausgangszuweisung. Somit würde eine feste Struktur bestehen, die den logischen Ablauf der Implementierung beschreibt. Wird der Aufbau der einzelnen Exportdateien genauer untersucht, fällt auf, dass die Struktur unterschiedlich sein kann. In Codesys zum Beispiel wird die Variablendeklaration am Ende beschrieben. Da keine feste Struktur vorgegeben werden kann, ist das Schablonenmuster wenig hilfreich.

Ein weiteres geeignetes Entwurfsmuster, ist das Strategie Muster. Es werden mehrere Strategien erstellt. Die unterschiedlichen Strategien sind in diesem zu lösendem Problem die unterschiedlichen Dateiformate, um das SIPN zu codieren. Im Gegensatz zum Schablonenmuster wird in diesem Muster der Algorithmus für jede Strategie komplett ausgetauscht. (Goll, et al., 2013) Es werden verschiedene Unterklassen erstellt, die vom selben Interface erben. Dadurch bekommen die einzelnen Strategien dieselben Funktionalitäten. Die Funktionalität ist in diesem Fall, das Generieren von Source Code. Zum Auswählen der unterschiedlichen Strategien wird ein Kontextobjekt erstellt. Der Nachteil ist, dass dieses Kontextobjekt die einzelnen Strategien kennen muss. (Goll, et al., 2013)

Nach den Abwegen der Vor- und Nachteile der beiden Muster wird sich für das Strategiemuster entschieden.

In Abbildung 11 sind die benötigten Komponenten dargestellt, die dem Design entsprungen sind. Die Schnittstelle enthält eine Methode zur Codegenerierung und bekommt dafür alle notwendigen

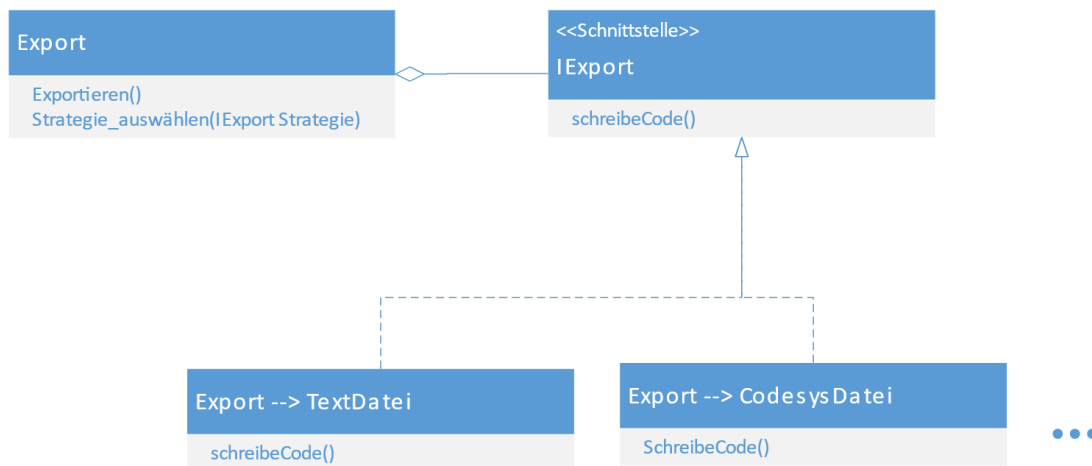


Abbildung 11: Strategiemuster für den Export
Quelle: eigene Darstellung

Informationen des SIPNs. Jede Strategie verwendet diese Schnittstelle und enthält einen Algorithmus, der die Codierung in das gewünschte Format vornimmt. In dem Kontextobjekt „Export“ kann die Strategie ausgewählt und mit dieser die Exportdatei erstellt werden.

4.2.3 Modellierung des Petri Netzes

Nach dem Entscheidungen getroffen wurden, wie das Grunddesign des Programms aussehen soll, wird nun die Darstellung des Petri Netzes entworfen. Das Petri Netz besteht aus einer Menge an Stellen und Transitionen. Jede Stelle und Transition kann durch ein Objekt der jeweiligen Klasse repräsentiert werden. Dabei ist das Ziel, dass das Modell das „Open-Closed-Prinzip“ befolgt. Das hat den Vorteil, dass die Stabilität der Bauteile erhöht wird und bestehende Bauteile wiederverwendet werden können. Dafür benötigt das Modell einen höheren Abstraktionsgrad. (Goll, 2017) Wie Robert Cecil Martin erwähnt, muss beim Design entschieden werden, welche Teile offen für Veränderungen sind. (Martin, 2002) In diesem Fall ist das Hauptziel, dass die unterschiedlichen Transitionsarten erweitert werden können.

Die bekannteste Methode ist die Verwendung von Vererbung. Bei der Vererbung von Klassen werden bereits bestehende Klassen abgeleitet, um neue Klassen zu erstellen. Die abgeleitete Klasse besitzt die gleichen Attribute und Methoden der Basisklasse.

Ein weiteres Prinzip ist das FCOI, welches im Entwurfsmuster-Buch von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides besonders hervorgehoben wird. (Gammer, et al., 1994) FCOI steht für „Favour object composition over class inheritance“, was auf Deutsch so viel bedeutet wie „ziehe Objektkomposition der Klassenvererbung vor“.

Eine Komposition beschreibt die Beziehung zwischen einem Ganzen und seinen Teilen. Dabei kann ein Teil nur zu einem Ganzen verknüpft sein. (Goll, et al., 2013) Bei der Objektkomposition wird von einer Abstraktion referenziert. Die Abstraktion gibt einen Vertrag vor, der bei der Implementierung eingehalten werden muss. (Goll, 2017) Das Entwurfsmuster „Strategie“, welches in Abschnitt 4.2.2 (Exportieren des SIPN als Source Code) dargestellt wurde, ist ein Beispiel für die Verwendung von Objektkomposition.

Der Vorteil der Objektkomposition ist, dass die Abhängigkeiten, im Vergleich zu der Vererbung, voneinander verringert werden. Änderungen in der Basisklasse haben bei der Vererbung direkten Einfluss auf die abgeleiteten Klassen. Dafür wird der Code der Basisklasse bei den erbdenden Klassen wiederverwendet. Daher lässt sich feststellen, dass die Vererbung nur dann sinnvoll ist, wenn eine sogenannte „is a“ Beziehung besteht. Das bedeutet, dass die abgeleitete Klasse tatsächlich ein Typ der Basisklasse ist.

In diesem Fall lässt sich eine „is a“ Beziehung leicht herstellen. Jede Transitionsart erbt von der Klasse Transition Eigenschaften wie zum Beispiel den Markenfluss. Somit entsteht eine Vererbungshierarchie, die in Abbildung 12 dargestellt ist.

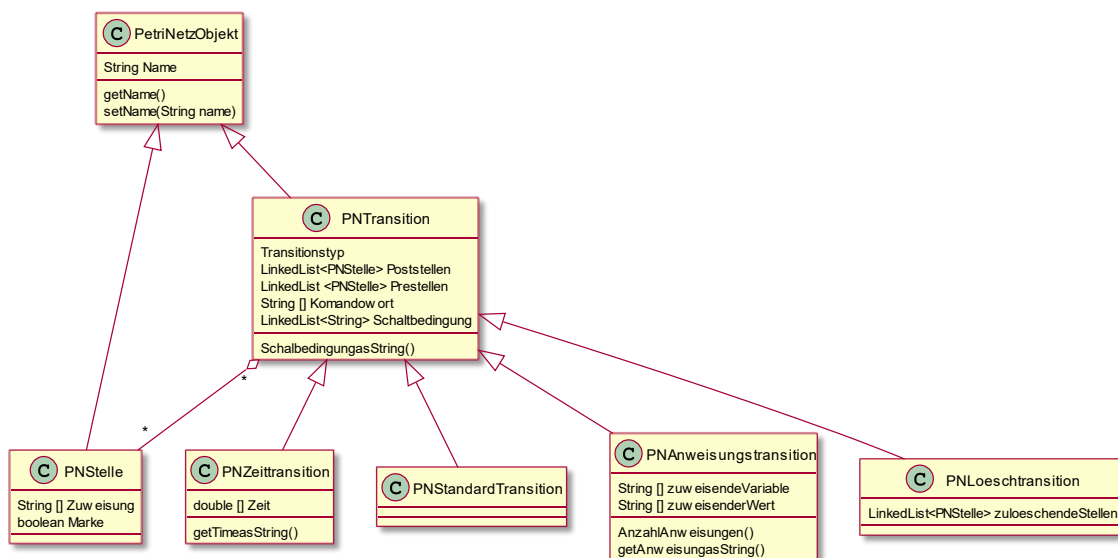


Abbildung 12: Klassendiagramm Petri Netz Objekte
Quelle: eigene Darstellung

Beim Zeichnen der Petri Netz Objekte werden die einzelnen Objekte durch Zeichenobjekte repräsentiert. Die Zeichenobjekte beschreiben die Art und Weise, wie die Zeichenobjekte aussehen. Ähnlich wie bei den Petri Netz Objekten ist bei der Abstraktion der Zeichenobjekte das Ziel, das „Open-Closed-Prinzip“ zu befolgen. Es stellt sich die gleiche Fragestellung wie zuvor. Der Anwendungsfall ist dabei ein ähnlicher.

Der Beziehungen der einzelnen Zeichenobjekten unterscheiden sich nur wenig von den der Petri Netz Objekte. Die unterschiedlichen Transitionsarten müssen alle in der Lage sein die

Schaltbedingung darzustellen und jeder Knoten stellt ein Petri Netz Objekt dar. Daher wurde sich bei der Abstraktion der Zeichenobjekte ebenfalls für die Vererbung entschieden. In Abbildung 13 ist die Vererbungshierarchie der Zeichenobjekte dargestellt.

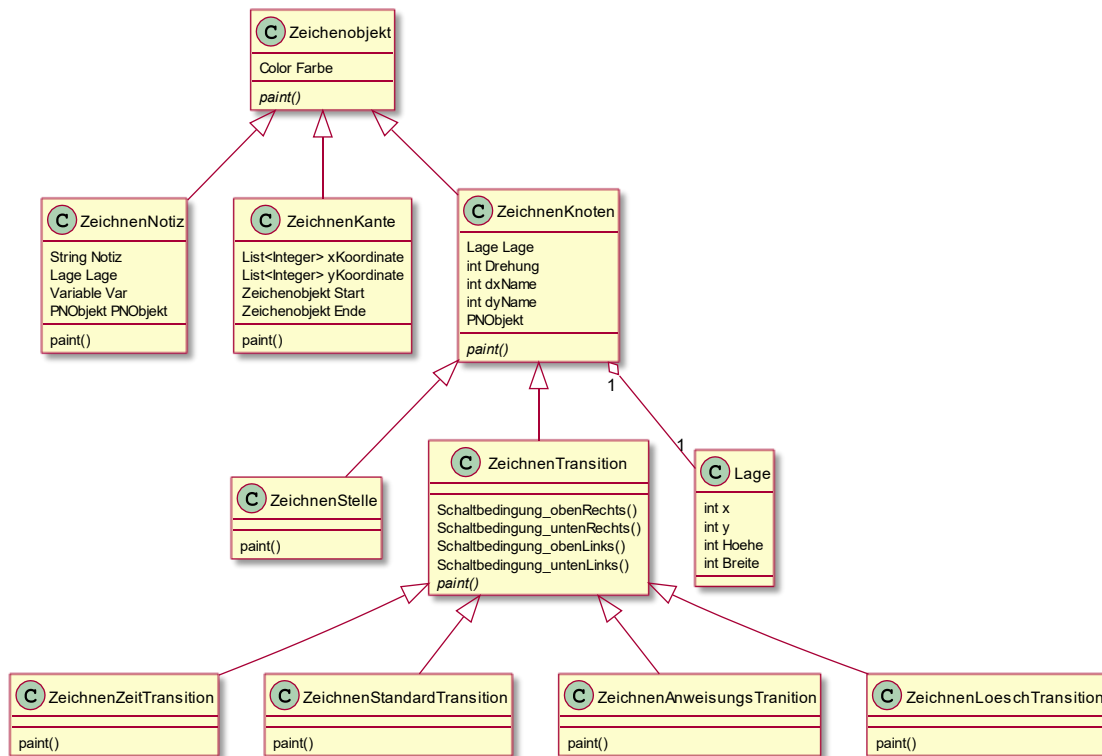


Abbildung 13: Klassendiagramm Zeichenobjekte
Quelle: eigene Darstellung

4.3 Die Programmiersprache

Nachdem die Architektur festgelegt wurde, wird eine Programmiersprache gewählt. Diese Programmiersprache muss die genannten Muster unterstützen. Eine geeignete Auswahl der Programmiersprache ist außerdem notwendig, um weitere Anforderungen wie die Plattformunabhängigkeit zu lösen (A1).

Die Muster setzen eine Programmiersprache voraus, die die objektorientierte Programmierung unterstützt, dazu gehören Aspekte wie Klassen, Objekte, Vererbung und Polymorphie. Die beiden populärsten Programmiersprachen, die die objektorientierte Programmierung unterstützt, sind (laut Carbonnelle, 2019) Python und Java.

In Java wird der Programmcode mittels eines Compilers in ein Byte Code übersetzt. Dieser Byte Code kann auf einer virtuellen Maschine gelesen und ausgeführt werden. Die Java Virtuelle Maschine (JVM) ist für alle bekannten Betriebssysteme vorhanden. Somit kann ein Java Programm unabhängig vom Betriebssystem des Anwenders genutzt werden. (Vgl. Abts, 2014) Python ist ein Interpreter Sprache und ebenfalls plattformunabhängig.

Beide Programmiersprachen unterstützten neben der objektorientierten Programmierung weitere Konzepte. Beide Programmiersprachen unterstützen die Parallelisierung von mehreren Prozessen (Multithreading). Dazu können in beiden Sprachen Bibliotheken verwendet werden, die Visualisierung und das Event-Handling.

Es lässt sich feststellen, dass das Programm in beiden Programmiersprachen mit ähnlichem Aufwand zu programmieren ist. Die Wahl trifft auf Java, da diese Programmiersprache Bestandteil des Studiengangs war.

5 Implementierung des Programms

Zunächst werden die durch das MVC-Muster vorgegeben Komponenten erstellt. Innerhalb der View wird derzeit nur eine Darstellung ermöglicht. Daher ist dies die kleinste Komponente.

5.1 Der View

Wie bereits in Abschnitt 4.1.1 erläutert, ist der View dafür verantwortlich, das SIPN darzustellen. Außerdem enthält Sie die Knöpfe zum Eingeben von Befehlen.

Die Gestaltung der Bedieneroberfläche stellte eine große Herausforderung dar. Sie soll leicht und intuitiv zu bedienen sein und alle funktionalen Anforderungen erreichen können. Die Hauptaufgabe ist das Zeichnen eines SIPNS. Daher wird der Größte Teil der Bedienoberfläche für das Zeichenfeld verwendet, indem das SIPN dargestellt ist. Die Knöpfe auf der Bedienoberfläche sind für die am häufigsten verwendeten Funktionalitäten, wie zum Beispiel das Zeichnen von Stellen und Transitionen. Zur besseren Übersicht werden die bereits verwendeten Variablen ebenfalls dargestellt.

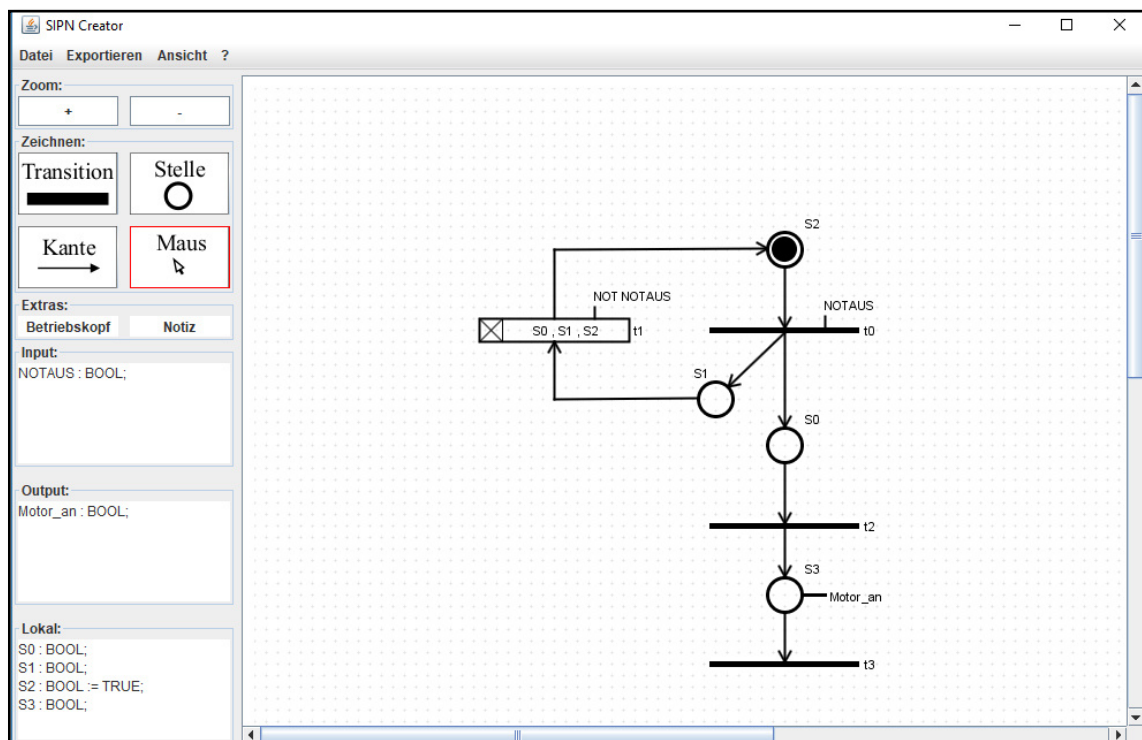


Abbildung 14: Benutzeroberfläche
Quelle: eigene Darstellung

Das Zeichenfeld besteht aus zwei Panels, die übereinander liegen. Das vordere Panel ist dafür verantwortlich, die Zeichenobjekte darzustellen. Dafür wird in der `paint()`-Methode die Liste aller Zeichenobjekte durchgegangen und nacheinander auf das Panel gezeichnet.

Falls mehrere Objekte ausgewählt werden, wird das Auswahlfenster ebenfalls in das Zeichenfeld gezeichnet.

Das zweite Panel ist dafür verantwortlich, den Hintergrund zu zeichnen. Zu dem Hintergrund zählen das Raster und die Druckbereiche. Damit dieses Panel zu sehen ist, muss das Zeichenfeld transparent gestaltet werden.

Je nach Art der Skalierung werden unterschiedlich viele Linien gezeichnet. Bei einer großen Darstellung hat der Hintergrund merkbar Einfluss auf die Antwortzeit des Programms. Dieses Problem wurde bereits in Kapitel 4.1 erwähnt und es müssen nun Methoden entwickelt werden, um diesen Einfluss zu reduzieren.

Es gibt verschiedene Möglichkeiten damit umzugehen.

Die Idee, die Hintergrundlinien nur dann neu zu zeichnen, wenn sich der sichtbare Bereich geändert hat, ist nicht umzusetzen. Änderungen im Vordergrund führen stets dazu, dass der Hintergrund ebenfalls neu gezeichnet wird.

Eine Möglichkeit ist, jedes Mal, wenn der sichtbare Bereich geändert wird, ein gebuffertes Bild mit dem Raster und dem Druckbereich zu erzeugen. Dann muss beim Zeichnen des SIPNS nur

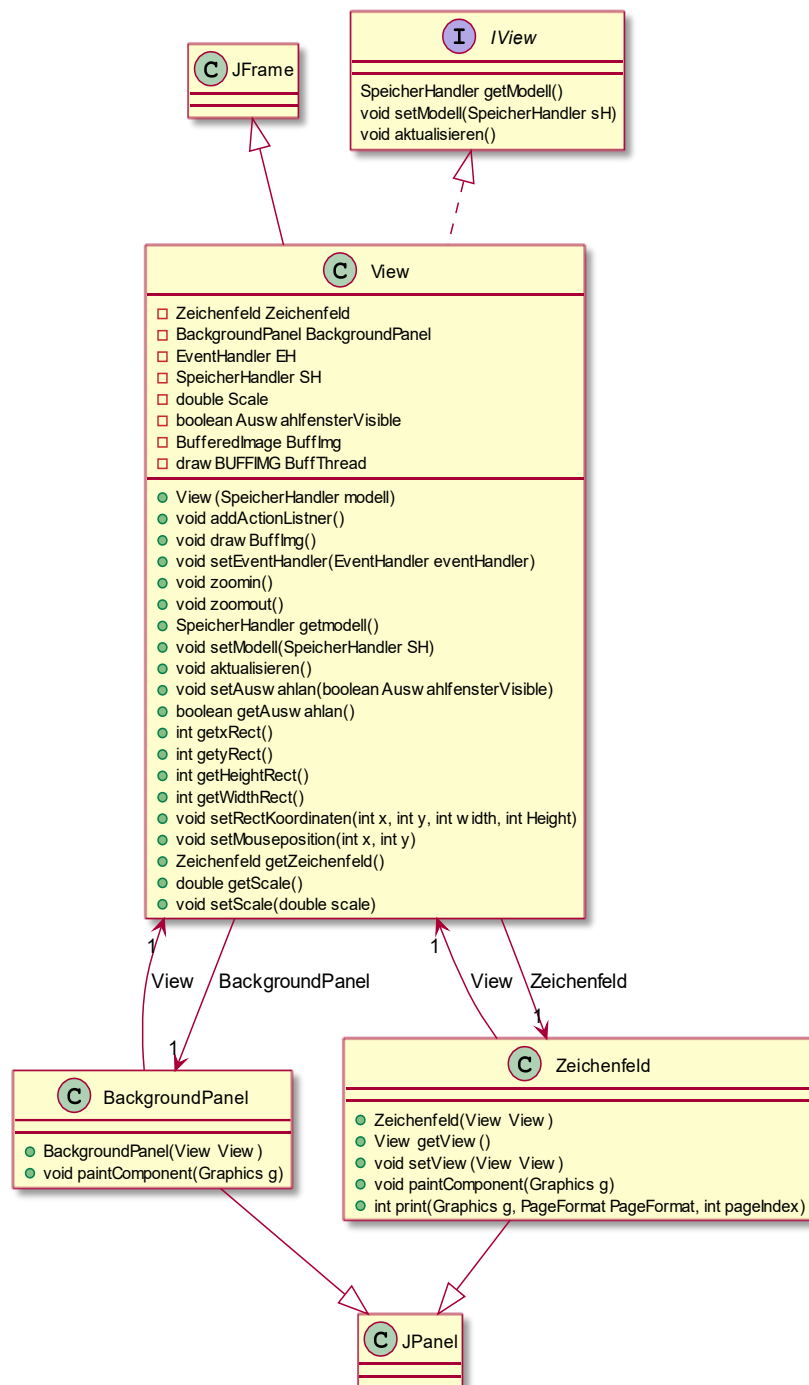


Abbildung 15: Klassendiagramm der View
Quelle: eigene Darstellung

das Bild auf das Panel gerendert werden. Dies geht deutlich schneller. Das Erstellen und Aktualisieren des gepufferten Bildes ist jedoch mindestens genauso zeitaufwändig wie das Zeichnen auf das Panel. Dadurch wird erreicht, dass lediglich beim Ändern des sichtbaren Bereiches hohe Antwortzeiten entstehen.

Das Einführen eines separaten Threads für die Erzeugung des Bildes bietet den Vorteil, dass der Hauptprozess nicht auf die Fertigstellung des Bildes warten muss. Jedoch kann es beim Ändern des sichtbaren Bereiches dazu kommen, dass der Hintergrund unvollständig ist, und es länger dauert, das Hintergrundbild vollständig zu zeichnen. Jedoch kann der Vorgang beim weiteren Verändern des sichtbaren Bereiches abgebrochen werden und mit den geänderten Voraussetzungen neu gestartet werden.

Da der Hintergrund lediglich als Hilfestellung dient und es wichtig ist, dass die Bearbeitung des SIPNs ohne Wartezeiten fortgesetzt werden kann, wird die Möglichkeit mit dem zusätzlichen Thread realisiert.

5.2 Das Modell

Das Modell besteht aus drei verschiedenen Subsystemen. Ein Bereich für die Zeichenobjekte, ein Bereich für die Petri Netz Objekte und ein Bereich für die verwendeten Variablen. Diese einzelnen Bereiche werden über den Speicher Handler kontrolliert. Der Speicher Handler dient als Fassade für die einzelnen Subsysteme. Dieser ist aus dem Programmdesign (Abschnitt 4.2.1) entstanden.

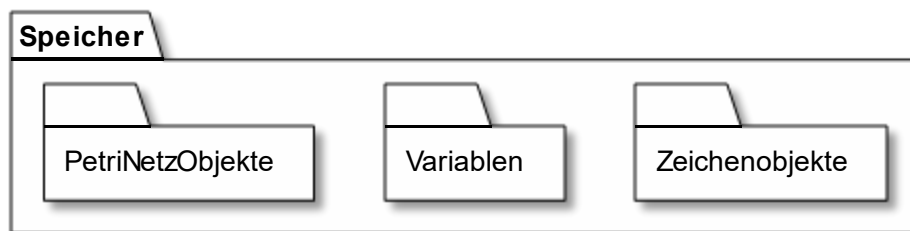


Abbildung 16: Darstellung der Pakete im Modell
Quelle: eigene Darstellung

In den einzelnen Paketen werden alle Objekte gespeichert, die das Modell repräsentieren.

5.2.1 Variablen Modell

Das Variablenmodell besteht aus allen verwendeten Variablen. Eine Variable wird durch die folgenden Eigenschaften repräsentiert:

private String Name;	//Variablenname
private int Typ;	//Variablentyp
private int Art;	//Variablenart
private String initialValue;	//Initialwert

Zudem besitzt die Klasse statische Variablen, die die einzelnen Typen und Klassen repräsentieren.

```
public final static int TypBool = 1;
public final static int TYPBYTE = 2;
public final static int TYPWORD = 3;
public final static int TYPDWORD = 4;
public final static int TypInteger = 5;
public final static int TypDInteger = 6;
public final static int TypReal = 7;
public final static int TypString = 8;
public final static int TYPTON = 9;
//...zu erweitern

public final static int ArtOutput = 101;
public final static int ArtInput = 102;
public final static int ArtLokal = 103;
```

Dadurch wird bestimmt, welche Variablenarten und welche Variablentypen implementiert wurden. Bei Erstellen oder Bearbeiten einer Variablen mit Initialwert, wird geprüft, ob dieser dem Wertebereich der Variablen entspricht.

Jede Variablenart wird in einer separaten Liste gespeichert. Dadurch wird die Übersichtlichkeit erhöht und beim Umwandeln in Source Code können die Listen bei der Variablendeklaration einzeln abgearbeitet werden.

Es können neue Variablen hinzugefügt, oder bei vorhandenen Variablen die Häufigkeit der Verwendung erhöht werden. Außerdem können diese Variablen wieder gelöscht werden. Für jede Variable kann geprüft werden, ob es sich um eine Stelle handelt, da beim Löschen dieser Variable mögliche Referenzen auf die Stelle geändert werden müssen. Zur Speicherung oder zum Exportieren wird dem Programm der Zugriff auf die gesamte Liste gewährt. Außerdem sind einzelne Suchfunktionen implementiert. Zum Beispiel das Ausgeben einer Variablen anhand des Namens.

5.2.2 Das Zeichenmodell des SIPNs

Die einzelnen Zeichenobjekte werden mit der von Java bereitgestellten Klasse `java.awt.Graphics2D` gezeichnet. Diese Klasse unterstützt das Rendern von zweidimensionalen Objekten und Formen. Durch sinnvolles Zusammensetzen der Formen können die Petri Netz Objekte dargestellt werden. Transitionen werden hauptsächlich aus Balken und Stellen aus Kreisen gezeichnet.

Jeder Knoten im Petri Netz bekommt eine Lage zugewiesen. Die Lage bestimmt die Position im Zeichenfeld und beinhaltet die Breite und die Höhe des Zeichenobjekts.

5.3 Der Controller

Die Hauptaufgabe des Controllers ist, die Eingaben des Anwenders zu verarbeiten. Dadurch bietet er dem Anwender des Programms die Möglichkeit über bestimmte Eingaben, Änderungen am

Modell und an der Darstellung vorzunehmen. In den folgenden Abschnitten wird beschrieben, wie die einzelnen Anforderungen implementiert wurden.

5.3.1 Zeichnen der Stelle

Beim Zeichnen einer neuen Stelle wird ein neues Zeichenobjekt erzeugt. Dieses wird auf das Zeichenfeld positioniert. Alle nötigen Eingaben können in der in Abbildung 17 dargestellten Eingabemaske getätigt werden.

Um die Bedienerfreundlichkeit zu erhöhen, wird der Name defaultmäßig gesetzt. Der default Name entspricht dem letzten Namen in der Liste, mit einer um eins erhöhten Zahl.

Beim Bestätigen der Eingabe wird überprüft, ob alle Eingaben valide sind. Voraussetzungen, dass die Eingaben valide sind:

- Der Namenseingabefeld ist nicht leer
- Entspricht der Namenskonvention "S" mit einer Zahl danach
- Der Name wurde noch nicht verwendet

Voraussetzung, dass die Zuweisung valide ist:

- Es darf keine fehlerhaften Zeichen enthalten wie zum Beispiel ,*' oder ,-'
- Falls die Variable bereits vorhanden ist, muss sie ebenfalls eine Ausgangsvariable vom Typ BOOL sein

Abbildung 17: Eingabemaske für die Stelle
Quelle: eigene Darstellung

5.3.2 Zeichnen der Transition

Der Ablauf beim Zeichnen einer Transition ist der Gleiche wie bei einer Stelle. Lediglich die Eingabemaske unterscheidet sich. Diese in Abbildung 18 dargestellte Eingabemaske ermöglicht die Eingabe der Schaltbedingung, die Auswahl der Transitionsart, den Namen und die Drehung der Transition.

Abbildung 18: Eingabemaske für die Transition
Quelle: eigene Darstellung

Je nachdem, welche Transitionsart ausgewählt wird bekommt der Anwender zusätzliche Eingabefelder, um die Transition entsprechend der Art zu beschreiben.

Abbildung 19: Eingabemaske für die zu löschenden Stellen
Quelle: eigene Darstellung

Abbildung 20: Eingabemaske für die Schaltverzögerung
Quelle: eigene Darstellung

Abbildung 21: Eingabemaske für Anweisungen
Quelle: eigene Darstellung

Bei der Löschrtransition werden alle bereits erstellten Stellen angezeigt. Durch Auswählen dieser Stellen wird die Stelle zu der Löschrtransition hinzugefügt. Müssen mehrere Stellen ausgewählt werden, ist es aufwändig jede Stelle einzeln auszuwählen. Durch eine Mehrfachauswahlfunktion könnten alle Stellen im Zeichenfeld ausgewählt werden, die beim Schalten der Löschrtransition gelöscht werden sollen. Somit können mehrere Stellen mit einem Klick ausgewählt werden. Ein Nachteil ist jedoch, dass das nachträgliche Bearbeiten durch weitere Funktionen ermöglicht werden muss. Das führt dazu, dass die Benutzeroberfläche komplexer wird. Eine optimale Lösung ist eine Kombination beider Varianten, um die Vorteile der schnellen Auswahl mit der übersichtlichen Bearbeitung zu vereinen. Allerdings erfüllt die Auswahl mittels Ankreuzfelder alle notwendigen Funktionen und ist intuitiver zu bedienen. Um den Implementierungsaufwand zu reduzieren wird nur diese Methode erstellt.

Bei der Auswahl der Zeittransition wird dem Anwender ermöglicht die Schaltverzögerung zu bestimmen. Um fehlerhafte Eingaben zu vermeiden, werden „Formatted Text Fields“ verwendet. In diesen Feldern kann festgelegt werden, welche Zeichen erlaubt sind. (Oracle, 2017) In dem Programm werden Zahlen, die durch Kommata getrennt werden, erlaubt. Stimmen die Zeichen nicht mit der Vorgabe überein, wird der Inhalt des Feldes nach Bestätigen der Eingabe automatisch angepasst.

Bei der Anweisungstrtransition wird dem Anwender die Möglichkeit gegeben bis zu fünf Anweisungen zu definieren. Durch Auswahl der jeweiligen Ankreuzfelder können Anweisungen hinzugefügt oder entfernt werden.

Unabhängig von der Transitionsart wird bereits beim Ändern der Schaltbedingung die Richtigkeit der Eingabe überprüft. Falls die Variable keine Fehler enthält, kann diese Variable der Schaltbedingung hinzugefügt werden. Dabei wird unterschieden, ob die Variable neu hinzugefügt werden muss oder nicht.

Soll ein Schaltwort eingefügt werden, wird untersucht, ob dies an der gewünschten Stelle möglich ist. Verallgemeinert lässt sich sagen, dass vor einem booleschen Operator eine boolesche Variable ist und vor einem Vergleichsoperator keine boolesche Variable. Sonst kommt es zu Typenfehlern.

Bei dem Bestätigen der Eingabe werden neben der Schaltbedingung, der Name und die spezifischen Eigenschaften der Transitionsarten kontrolliert:

- Standardtransition: Die Standardtransition kann keine speziellen Fehler besitzen.
- Zeittransition: Der Wertebereich für den „TIME“ Datentyp ist in der Norm nicht weiter definiert und ist herstellerspezifisch. (DIN EN 61131-3:2014-06) Die Zeittransition ist lediglich dann fehlerhaft, wenn keine Zeit eingegeben wurde, da in dem Fall eine Standardtransition besser geeignet wäre.
- Löschrtransition: Bei der Löschrtransition muss mindestens eine Stelle ausgewählt werden.

Anweisungstransition: Der Kontrollcheck bei der Anweisungstransition ist komplexer, als bei den vorherigen genannten Transitionsarten. Für jedes Textfeld, indem die zuweisende Variable eingegeben wird, wird auf ungültige Zeichen geprüft. Ist kein ungültiges Zeichen vorhanden, wird dieser Variablenname einer temporären Liste hinzugefügt. Die Eingabe für die Zuweisungswerte werden durch vorhandene Leerzeichen in einzelne Wörter getrennt. Diese Wörter werden ebenfalls der temporären Liste hinzugefügt. Sind alle Textfelder der Anweisungstransition untersucht worden, werden die Variablen falls nötig hinzugefügt. Zahlen, bekannte Variablen und Operationszeichen werden nicht als Variable angelegt.

Während des Zeichnens der Transition kann der Anwender der Software die Priorisierung der Transitionen bestimmen. Dies erfolgt über den Namen der Transition. Jeder in der Software zulässige Transitionsnamen besteht aus einem "t" und einer Zahl. Der Wert der Zahl bestimmt über die Priorisierung der Transition.

Nullen vor der Zahl führen dazu, dass diese Transitionen höher priorisiert werden. Das bedeutet, dass die Transition mit dem Namen t005 im Programmcode vor der Transition mit dem Namen t01 und vor der Transition mit dem Namen t0 erstellt wird.

Beim Umwandeln des Petri Netzes in normgerechten Code muss nur die nach Namen sortierte Liste vom Programm abgearbeitet werden.

5.3.3 Zeichnen von gerichteten Kanten

Beim Zeichnen der gerichteten Kante wird zunächst das Startobjekt ausgewählt. Solange kein Startpunkt gesetzt ist, wird weiter gewartet, bis eine Transition oder eine Stelle ausgewählt wurde. Nachdem der Benutzer des Programms ein Startobjekt ausgewählt hat, kann er beliebig viele Zwischenpunkte setzen. Die Punkte werden miteinander verbunden. Wenn der Benutzer einen weiteren Knoten auswählt, wird überprüft, ob dies möglich ist.

Falls beim Zeichnen der gerichteten Kante kein Fehler aufgetreten ist, werden bei der Transition die Poststellen bzw. die Prästellen aktualisiert. Dazu wird die Richtung der Kante geprüft. Die Stelle, die mit der Transition verbunden ist, wird entsprechend in die Liste der Prästellen oder der Poststellen eingefügt.

Wenn der Anwender eine gerichtete Kante an eine Stelle oder an eine Transition bindet, wird der Punkt, an dem die Kante das Zeichenobjekt berührt vom Programm berechnet. Es unterstützt den Anwender dabei, ohne großen Aufwand ein ordentliches SIPN zu zeichnen.

Der Anschlusspunkt an der Stelle wird über den Kosinus und den Sinus ermittelt:

Um den Winkel α zu bestimmen, wird der Arkustangens verwendet.

$$\alpha = \arctan\left(\frac{\text{deltay}}{\text{deltax}}\right)$$

Mit α kann die relative Position des Anschlusspunktes zu der Mitte bestimmt werden.

Relativ in x-Richtung: $dx = \cos(\alpha)$

Relativ in y-Richtung: $dy = \sin(\alpha)$

Der Punkt an der Transition liegt immer zentral auf

der langen Seite der Transition. Je nachdem, ob die gerichtete Kante unterhalb oder oberhalb der Transition verläuft befindet sich der Punkt auf der unteren oder oberen Begrenzung der Transition.

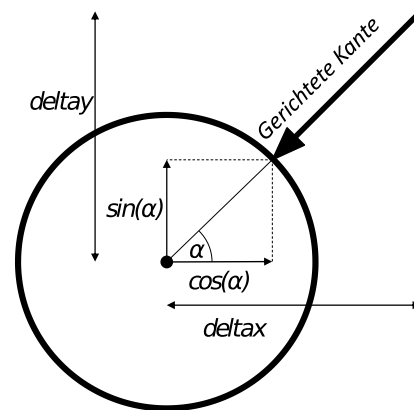


Abbildung 22: Berechnung der Anschlusspunkte

5.3.4 Zeichnen von Betriebsköpfen

Die Betriebsköpfe dienen dazu, untergeordnete Petri Netze zu steuern. Zum Beispiel können sie das Ein- und Ausschalten einer Anlage oder den Wechsel zwischen Hand und Automatikbetrieb kontrollieren. (Aspern, 2003) Wichtiges Merkmal eines Betriebskopfes ist die Löschtransition. Innerhalb der Löschtransition werden alle Stellen des untergeordneten Petri Netzes gelöscht. Somit wird das untergeordnete Netz deaktiviert.

Beim Aktivieren des untergeordneten Petri Netzes durch das Schalten einer Transition werden zwei Marken erzeugt. Eine Statusmarke und eine Arbeitsmarke. Dafür werden zwei freie Stellen benötigt. Um diese vier Zeichenobjekte nicht einzeln erstellen zu müssen, gibt es einen vorgefertigten Betriebskopf, den der Anwender in die Zeichnung einfügen kann. Dieser ist in Abbildung 23 dargestellt.

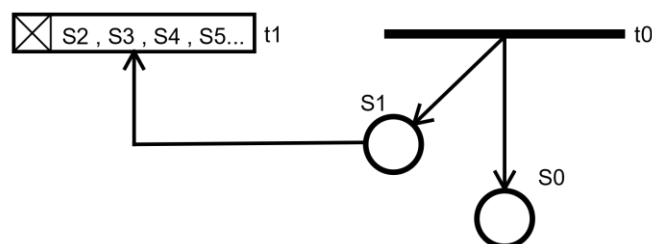


Abbildung 23: Beispielhafter Betriebskopf
Quelle: eigene Darstellung

5.3.5 Einfügen von Kommentaren

Kommentare dienen dem besseren Verständnis von SIPN und sind optional. Die Kommentare können ähnlich wie die anderen Zeichenobjekte erstellt, verschoben und gelöscht werden. Anstelle von Formen wird in der `paint()`-Methode lediglich ein String gezeichnet:

```
g2D.drawString(Notiz, Lage.getX(), Lage.getY());
```

Kommentare, die sich auf Variablen oder Transitionen beziehen, eignen sich dafür, mit in den Source Code übernommen zu werden. Um die Kommentare an der richtigen Position im Text einzufügen, benötigt der Kommentar entweder eine Referenz auf eine Variable oder eine Referenz auf ein Petri Netz Objekt.

5.3.6 Löschen von Zeichenobjekten

Beim Löschen von Zeichenobjekten ist zu beachten, dass auch die zugehörigen Variablen und das zugehörige Petri Netz Objekt entfernt werden, sodass nach dem Löschen des Zeichenobjektes keine ungenutzten Objekte im Speicher bleiben.

Bei Stellen muss in jedem Fall der Stellenname gelöscht werden. Die zugewiesenen Ausgangsvariablen können bestehen bleiben, falls ein anderes Zeichenobjekt die Variablen mit dem gleichen Namen verwendet. Wenn nicht, müssen diese ebenfalls aus dem Speicher entfernt werden.

Ein Problem, welches beim Löschen der Stelle auftreten kann, ist der Umgang mit den Petri Netz Objekten, die Referenzen auf das Petri Netz Objekt der Stelle besitzen. Inhibitorkanten oder Testkanten werden ähnlich wie beim Bearbeiten nicht automatisch aktualisiert, da die direkte Referenz nicht gespeichert wird. Lediglich der Name der Stelle wird verwendet.

Die Transitionen verwenden, innerhalb der Schaltbedingung, Variablen, die gegebenenfalls gelöscht werden müssen. Die bei der Zeittransition erzeugte Timer Instanz muss aus der Liste der lokalen Variablen entfernt werden.

5.3.7 Verschieben von Zeichenobjekten

Ziel ist es, jedes Zeichenobjekt verschieben zu können. Beim Verschieben eines Zeichenobjektes soll es zusammen mit dem Transitions- bzw. dem Stellennamen verschoben werden. Die relative Position des Namens zum Zeichenobjekt bleibt dabei erhalten. Die Namen sollten ebenso separat vom Zeichenobjekt verschoben werden können. Somit kann die Ausrichtung des Namens am Zeichenobjekt bestimmt werden. Eine Herausforderung stellen die gerichteten Kanten dar. Falls sie Zwischenpunkte besitzen, müssen die Zwischenpunkte einzeln ausgewählt und verschoben werden können.

Ein Zeichenobjekt zu verschieben ist durch das Ziehen des Objektes auf eine andere Position möglich. Das Programm überprüft, ob der Anwender ein Zeichenobjekt ausgewählt hat. Wurde

kein Zeichenobjekt direkt ausgewählt, wird vom Programm untersucht, ob ein Namenslabel ausgewählt wurde. Somit können einzelne Zeichenobjekte und Namenslabel verschoben werden. Die Ausnahmen bilden die gerichteten Kanten. Diese sind fest an den Start und Zielobjekte gebunden.

Ein Namenslabel über einem Zeichenobjekt kann nicht direkt ausgewählt werden. Beim Versuch das Namenslabel auszuwählen, wird immer das Zeichenobjekt zusammen mit dem Label verschoben. Für das nachträgliche Ändern müsste das Zeichenobjekt gelöscht und neu erstellt werden.

Eine weitere implementierte Funktion ist die Gruppenauswahl. Durch Halten der Maus im leeren Feld kann ein Auswahlfenster erstellt werden. Beim Loslassen der Maustaste werden alle Zeichenobjekte, die sich innerhalb des Auswahlfensters befinden in eine Auswahlliste gespeichert. Beim Verschieben werden alle Zeichenobjekte innerhalb der Auswahlliste verschoben.

Zusätzlich werden alle Zwischenpunkte der gerichteten Kante in einer weiteren Liste gespeichert. Beim Verschieben werden nur die Zwischenpunkte, die sich innerhalb des Auswahlfensters befinden mitverschoben. Wird mit dem Auswahlfenster nur ein Zwischenpunkt einer gerichteten Kante ausgewählt, kann dieser Zwischenpunkt separat verschoben werden.

Während des Verschiebens werden die Anschlusspunkte wie beim Neuzeichnen einer gerichteten Kante aktualisiert. Dadurch behält der Anwender eine ordentliche Darstellung des SIPNs.

5.3.8 Variablenhandling

Für die Umwandlung in den entsprechenden Source Code ist es notwendig, dass alle Variablen, die im SIPN verwendet werden, eindeutig definiert sind.

Das Ziel ist jedoch, dass die Variablen Definitionen größtenteils im Hintergrund ablaufen. Das bedeutet, dass sich der Anwender so wenig wie möglich mit den Variablen beschäftigen muss. Wird eine Variable verwendet, soll diese automatisch zu den vorhandenen Variablen hinzugefügt werden. Dies ist jedoch nicht immer möglich. Damit das Programm möglichst viele Variablendefinitionen automatisch übernimmt, ist zu bestimmen, wann die Variablen in ihren Verwendungen eindeutig definiert sind.

Die lokalen Variablen, die die Stelle repräsentieren, sind eindeutig definiert. Wenn die Stelle eine Startmarke besitzt, wird der Initialwert gesetzt. Die Ausgangsvariablen, die den Stellen zugewiesen werden, sind ebenfalls eindeutig definiert. Wie die Stellen müssen diese vom Typ „BOOL“ sein und entsprechend der Verwendung von der Variablenart „VAR_OUTPUT“.

In den anderen Fällen ist die Variable manuell zu definieren, da diese Verwendungen nicht eindeutig sind. Bevor eine Variable erstellt wird, überprüft das Programm, ob bereits eine Variable mit gleichem Namen existiert. Ist dies der Fall, müssen der Typ und die Art übereinstimmen. Kann die neue Variablenverwendung mehrere Typen oder Arten annehmen, muss nur die der bereits

vorhandenen Variable dabei sein. Stimmen Typ und Art nicht überein, dann ist die neue Verwendung dieser Variablen ungültig. In diesem Fall muss der Name angepasst werden.

Muss eine neue Variable erstellt werden, öffnet sich ein weiteres Dialogfeld. Dort können die Variablenart und der Variablentyp ausgewählt werden. Bei lokalen und Ausgangsvariablen ist es möglich einen Anfangswert festzulegen. Je nachdem wie die neue Variable eingesetzt wird, ist die Auswahl für den Datentyp und der Variablenart angepasst.

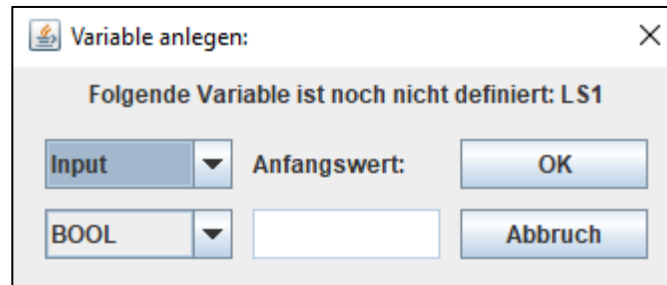


Abbildung 24: Dialogfeld Variablendeklaration
Quelle: eigene Darstellung

5.3.9 Drucken des SIPNs

Für den Nutzer des Programms ist eine einfache Anwendung zu erarbeiten, mit der dieser den Ausschnitt und die Größe bestimmen und drucken kann.

Hierfür werden drei Methoden entwickelt und miteinander verglichen.

Eine Methode ist das SIPN so stark zu verkleinern, dass das komplette SIPN auf einem DIN-A4-Blatt gedruckt wird.

Eine weitere Methode besteht darin, nur die aktuelle Ansicht des Zeichenfelds zu drucken.

Die dritte Methode ist, das Zeichenfeld in mehrere DIN-A4-Blättern zu unterteilen. Die Zeichenobjekte werden von dem Anwender der Software auf die gewünschte Seite geschoben und entsprechend skaliert. Die Druckbereiche sind fest im Zeichenfeld positioniert.

Tabelle 18: Vor und Nachteile der einzelnen Druckmethoden

	Vorteil	Nachteil
Methode 1	<ul style="list-style-type: none"> Kein Mehraufwand für den Anwender Gut geeignet für kleine Petri Netze 	<ul style="list-style-type: none"> Zeichenobjekte kleiner desto größer das Petri Netz
Methode 2	<ul style="list-style-type: none"> Individuelle Gestaltung der Blätter Gut geeignet, um Ausschnitte zu drucken 	<ul style="list-style-type: none"> Mehrfach Drucken notwendig, sonst erhält der Anwender das gleiche Ergebnis wie in Methode 1 beschrieben Abhängig von der Größe des Bildschirms
Methode 3	<ul style="list-style-type: none"> Individuelle Gestaltung der Blätter Drucken aller Blätter mit einem Druckauftrag Gut geeignet für große Petri Netze, die auf mehreren Seiten gedruckt werden 	<ul style="list-style-type: none"> Mehraufwand durch das Verschieben der einzelnen Zeichenobjekte Mehraufwand für die Programmierung, da das Drucken auf mehreren Seiten realisiert werden muss

Quelle: eigene Darstellung

Das Drucken des Petri Netzes soll unabhängig von der Größe funktionieren, daher wurde die Methode 1 verworfen. Mit der Methode 3 können ähnliche Ergebnisse erzielt werden wie mit der Methode 2. Bei beiden Methoden kann der Anwender des Programms die Druckauswahl individuell gestalten. Ziel beim Drucken ist meist das gesamte Petri Netz für die Dokumentation zu drucken. Falls das SIPN auf mehrern Blättern gedruckt werden soll, geht es mit Methode 3 schneller, da nur ein Druckauftrag benötigt wird. Daher wurde sich für die dritte Methode entschieden.

Zum Drucken wird das Java-Interface „Printable“ verwendet. Damit können die Zeichenobjekte ähnlich wie beim Zeichenfeld auf die Blätter gerendert werden.

Die Anzahl an Blättern, die erstellt werden, ist äquivalent zu der Größe des Zeichenfeldes. Die Blätter werden in folgender Reihenfolge gedruckt:

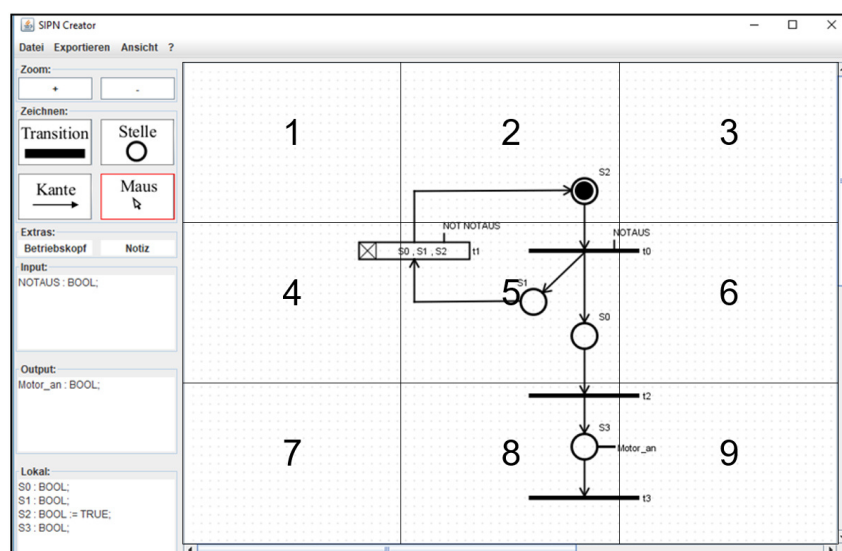


Abbildung 25: Exemplarische Reihenfolge der Druckbereiche
Quelle: eigene Darstellung

Zunächst wird das gesamte Zeichenfeld von links nach rechts abgearbeitet, dann von oben nach unten. Der Anwender kann entscheiden, ob die Druckbereiche im Hochformat oder im Querformat ausgerichtet werden.

Um das SIPN auf das jeweilige Blatt zu zeichnen, müssen diese Zeichenobjekte angepasst werden. Der Grund ist, dass für jedes Zeichenobjekt auf jedem Blatt die relative Position zu der linken oberen Ecke angegeben werden muss.

Zunächst wird geprüft, an welcher Stelle das Blatt im Verhältnis zum gesamten Zeichenfeld liegt. Es wird also geprüft, die wievielte Seite in x-Richtung und die wievielte Seite in y-Richtung das aktuelle Blatt ist. Dann wird untersucht, welche Zeichenobjekte in dem Druckbereich sind. Für jedes Zeichenobjekts, welches sich innerhalb des Druckbereiches befindet, wird die Position auf dem Blatt berechnet.

5.3.10 Exportieren des SIPN als Bild

Um das SIPN als Bild zu exportieren, wird zunächst untersucht, welches Bildformat dafür geeignet ist und welche Funktionen standardmäßig bereitgestellt werden, um Bilder in Java zu exportieren. Gewünscht ist ein Bildformat, welches das SIPN gut darstellen kann und möglichst verlustfrei ist.

Die Java Klassenbibliothek zum Exportieren von Bildern unterstützt die folgenden Bildformate: (Oracle, 2018)

- JPEG (Joint Photographic Experts Group)
- PNG (Portable Network Graphics)
- BMP (Windows Bitmap)
- WBMP (Wireless Application Protocol Bitmap)
- GIF (Graphics Interchange Format)

Bei all diesen Bildformaten handelt es sich um Rastergraphiken (Günter). Rastergraphiken speichern die Werte in Rastern ab. Jedes Feld im Raster bekommt ein Farbwert zugeteilt. Je größer die Pixelanzahl (Felder im Raster), desto größer ist die Bildgröße.

Die bessere Wahl wären Vektorgraphiken, da diese verlustfrei sind. (Wallner, 2016) Ein Beispiel für eine Vektorgraphik ist das Portable Document Format (PDF). Da Java standardmäßig keine Klasse für das Exportieren von Vektorgraphiken besitzt, ist der Aufwand dies in die Software zu implementieren größer als bei einer Pixelgraphik. Da externe Programme bei der Druckfunktion PDF erstellen können wird diese Funktion als ausreichend bewertet.

Um das Installieren von zusätzlicher Software zu vermeiden, wird dem Anwender der Software ermöglicht, das SIPN als Rastergrafik zu exportieren. Das JPEG Format besitzt eine verlustbehaftete Komprimierung und wird aus diesem Grund nicht verwendet. Das BMP und das WBMP Format ist aufgrund der Größe der Datei nur wenig verbreitet. Am besten geeignet ist entweder das PNG Format oder das GIF Format. Beide Formate sind in der Komprimierung verlustfrei. Der

Vorteil von PNG-Dateien ist, dass sie Farben besser speichern können. Dafür ist die .gif Datei meistens kleiner. Da das SIPN eine farblose Zeichnung ist, ist das GIF Format besser geeignet. Jedoch wird das Exportieren in das PNG Formt ebenfalls implementiert, weil bei großen einfarbigen Flächen das PNG Format kleiner sein kann. (Bachmann, 2012)

Falls die Zeichnung exportiert werden soll, werden zunächst alle Zeichenobjekte in ein Buffered Image gezeichnet, welches dann mit der Java Klasse „ImageIO“ gespeichert werden kann.

Nun muss die Größe des Bildes festgelegt werden. Ein größeres Bild führt dazu, dass die Qualität höher ist. Der Nachteil ist eine zur Größe proportional steigende Arbeitszeit. Zudem muss eine größere Menge zwischengespeichert werden.

Als Lösung wird eine definierte Bildgröße verwendet. (15.000 Pixel * 15.000 Pixel) Das Programm bestimmt die Größe des gezeichneten Bildes. Dafür wird die Größte x-Position und die größte y-Position aller Zeichenobjekte ermittelt. Diese Größe der Zeichnung wird auf die Bildgröße skaliert. Dadurch werden kleine Petri Netze mit sehr hoher Qualität erzeugt. Große Petri Netze besitzen weiterhin eine für die meisten Verwendungen ausreichende Qualität.

Somit hat der Anwender die Möglichkeit eine Pixelgraphik oder durch das Installieren einer externen Anwendung eine Vektorgraphik zu erstellen.

5.4 Fehlervermeidung

Die Fehlervermeidung ist bei der Erstellung des SIPNs elementar. Nur ein fehlerfreies SIPN ist in funktionierenden Source Code umzuwandeln. Damit der Anwender erst gar nicht versucht, aus einem fehlerhaften SIPN in Code umzuwandeln, wird das SIPN auf Fehler untersucht. Zwischen folgenden Fehlern wird unterschieden:

- Variablenfehler
- Zeichnungsfehler
- Logische Fehler

5.4.1 Variablenfehler

Wird eine Variable verwendet, können Fehler auftreten. Bei der Verwendung einer Variablen wird direkt bei der Eingabe untersucht, ob Fehler vorhanden sind. Dabei wird vor allem überprüft, ob die Variablenart und der Variablentyp passend zu der Verwendung sind. Zudem wird geprüft, ob die Variablen ungültige Zeichen aufweisen.

Dadurch, dass die Variablen direkt bei der Eingabe überprüft werden, können Folgefehler ausgeschlossen werden.

5.4.2 Zeichnungsfehler

Wenn beim Zeichnen die Regeln des SIPNs nicht befolgt werden, können Zeichnungsfehler entstehen.

Auf einige Fehler wird der Anwender der Software bereits während des Zeichnens hingewiesen. Zum Beispiel, wenn der Anwender eine Stelle mit einer Stelle, oder eine Transition mit einer Transition verbindet. Außerdem kann es zu Problemen führen, wenn gerichtete Kanten kein Start oder Zielobjekt besitzen. Daher werden beim Löschen von Zeichenobjekten, die anliegenden Kanten ebenfalls mit entfernt, um diese Fehlerquelle auszuschließen.

Auf andere Fehler wird erst vor dem Umwandeln in Source Code geprüft:

Für einen sauberen Ablauf des Prozesses wird mindestens eine Startmarke benötigt. Ohne Startmarke kann kein Markenfluss entstehen und der Anlagenprozess ist statisch.

Das SIPN darf keine Schlingen besitzen (Lunze, 2012). Da bei einer Schlinge die Poststelle und die Prästelle identisch sind, würde eine nach der Regel generierten Transition nicht schalten. Die Zeittransition gilt hierbei als Ausnahme. Mögliche Rückführungen

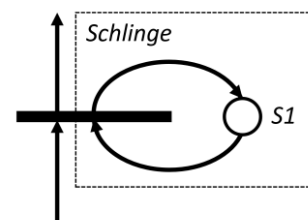


Abbildung 26: Beispiel Schlinge

bei einer Zeittransition werden genutzt, um Taktgeneratoren zu erstellen. Da diese Zeittransitionen anders codiert werden, gelten diese dennoch als fehlerhaft.

Jede Stelle wird überprüft, ob sie mit dem SIPN verbunden ist. Falls eine Stelle nicht mit dem SIPN verbunden ist, wird der Anwender aufgefordert, diese an das SIPN anzuschließen oder zu löschen.

Zudem muss jede Transition mindestens eine Poststelle und eine Prästelle besitzen. Somit wird das SIPN ordentlich gehalten.

Auf diese Fehler wird das SIPN vor dem Umwandeln in normgerechten Code überprüft.

5.4.3 Logische Fehler

Logische Fehler treten dann auf, wenn das gezeichnete SIPN den Prozess nicht ordnungsgemäß repräsentiert. Diese Art von Fehlern können nicht abgefangen werden. Erst durch Testen des automatischen Prozesses fallen diese Fehler auf. Die einzige Möglichkeit den Anwender zu unterstützen, um diese Fehler zu vermeiden, ist eine Simulation hinzuzufügen. Somit muss diese Beschreibung des Prozesses nicht direkt an der Anlage getestet werden. Eine exakte Simulation eines Prozesses ist jedoch sehr aufwändig zu implementieren, da die Zeiten für bestimmte Bewegungen unbekannt sind. Der Nutzen einer simpleren Simulation ist nur bedingt hilfreich. Die Beschreibung muss dennoch an der Anlage getestet werden. Da der Implementierungsaufwand im Verhältnis zum Nutzen eher gering ist, werden logische Fehler nicht abgefangen.

6 Softwaretests

Um bereits während der Erstellung der Software zu überprüfen, ob einige Funktionen korrekt funktionieren, wurden Java -Test Klassen entwickelt. Diese eignen sich besonders zum Überprüfen der Benutzereingaben, da für diese Tests keine Visualisierung notwendig ist.

6.1 Testen der Variablen

Bei den Variablen wird getestet, ob der Anfangswert zum eingegebenen Wert übereinstimmt. Dafür wird das Ergebnis der Methode „checkInitialWert()“ mit der Erwartung verglichen. Für jeden Variablentyp werden Werte innerhalb und außerhalb des zugehörigen Wertebereichs getestet. Dafür wird die „Assert“ (dt. zu Behaupten) Klasse verwendet. In dem Methodenaufruf „assertEquals“ wird der zu erwartende Wert (erster Parameter) mit dem zu erwartenden Wert (zweiter Parameter) verglichen. Im Folgenden sind die Testfälle aufgelistet:

```
assertEquals(true, Variable.checkInitialWert(Variable.TypBool, "TRUE"));
assertEquals(true, Variable.checkInitialWert(Variable.TypBool, "true"));
assertEquals(false, Variable.checkInitialWert(Variable.TypBool, "290"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TypInteger, "+290"));
assertEquals(true, Variable.checkInitialWert(Variable.TypInteger, "-290"));
assertEquals(false, Variable.checkInitialWert(Variable.TypInteger, "-290000"));
assertEquals(false, Variable.checkInitialWert(Variable.TypInteger, "true"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TYPBYTE, "90"));
assertEquals(true, Variable.checkInitialWert(Variable.TYPBYTE, "240"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPBYTE, "260"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPBYTE, "true"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPBYTE, "90.4"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TypDInteger, "+290"));
assertEquals(true, Variable.checkInitialWert(Variable.TypDInteger, "-290"));
assertEquals(false, Variable.checkInitialWert(Variable.TypDInteger, "-1000000000000"));
assertEquals(true, Variable.checkInitialWert(Variable.TypDInteger, "2147483640"));
assertEquals(false, Variable.checkInitialWert(Variable.TypDInteger, "true"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TYPWORD, "90"));
assertEquals(true, Variable.checkInitialWert(Variable.TYPWORD, "+240"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPWORD, "70000"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPWORD, "true"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPWORD, "90.4"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TYPDWORD, "90"));
assertEquals(true, Variable.checkInitialWert(Variable.TYPDWORD, "+240"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPDWORD, "5000000000"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPDWORD, "true"));
assertEquals(false, Variable.checkInitialWert(Variable.TYPDWORD, "90.4"));
```

```
assertEquals(true, Variable.checkInitialWert(Variable.TypReal, "25.264"));
assertEquals(false, Variable.checkInitialWert(Variable.TypReal, "true"));
```

Jeder Testfall hat den Erwartungen entsprochen.

6.2 Testen der Schaltbedingungen

Damit die erweiterte Schaltbedingung getestet werden kann, werden zwei Variablen erstellt. Eine vom Typ BOOL (E1) und eine vom Typ WORD (E2). Beim Testen wird geprüft, ob die einzelnen Variablentypen passend verwendet wurden.

Tabelle 19: Testfälle bei der erweiterten Schaltbedingung

Zu testende Schaltbedingung	Zu erwartendes Ergebnis
(E1 AND E1) OR E1	Korrekt
(E1 AND) OR E1	Fehlerhaft
(E1 AND E2) OR E1	Fehlerhaft, da E2 vom Typ WORD ist
(E2 AND E1) OR E1	Fehlerhaft, da E2 vom Typ WORD ist
NOT	Fehlerhaft
NOT E2	Fehlerhaft, da E2 ein WORD ist
NOT E1	Korrekt
E2 < E2	Korrekt
E2 <)	Fehlerhaft ungleiche Anzahl an Klammern
(E1 AND E1) OR (E2 <> E2)	Korrekt
()	Fehlerhaft

Quelle: eigene Darstellung

Die Testergebnisse stimmen mit den zu erwarteten Ergebnissen überein.

6.3 Testen des Zeichnens

Für den Test, ob die Zeichenfunktionen funktionieren, wird angestrebt ein Prozess mittels des Programms zu zeichnen. Der zu zeichnende Testprozess wird so gewählt, dass jedes Zeichenobjekt, das implementiert wurde, mindestens einmal gezeichnet wird. Um den entstehenden Source Code zu kontrollieren wird dieses SIPN händisch umgewandelt.

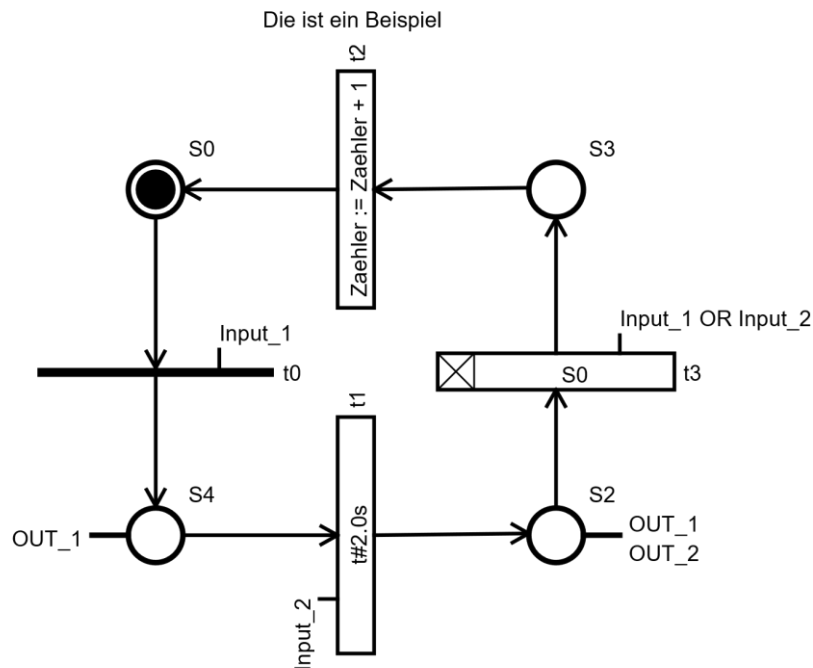


Abbildung 27: Testprozess
Quelle: eigene Darstellung

Die Codierung startet mit dem Funktionsblocknamen.

FUNCTION_BLOCK TEST // Der Name dieser POE ist „TEST“

Am Anfang müssten die Variablen deklariert werden. Wird angenommen, dass der „Zaehler“ als Lokal definiert wurde, ist folgende Variablendeklaration zu erwarten:

```

VAR_INPUT
    Input_1 : BOOL;
    Input_2 : BOOL;
END_VAR
VAR_OUTPUT
    OUT_1 : BOOL;
    OUT_2 : BOOL;
END_VAR
VAR
    S0 : BOOL := 1; //Da diese Stelle eine Marke besitzt, ist der Anfangswert True.
    S2 : BOOL;
    S3 : BOOL;
    S4 : BOOL;
    TON1 : TON; //TON für die Zeittransition
    Zaehler : INT;
END_VAR

```

Es gibt vier Transitionen, die der Reihe nach in den Source Code umgewandelt werden müssen:
Die Transition t0 ist eine Standardtransition und sieht wie folgt aus:

```

IF S0 AND NOT S4 AND Input_1 THEN
    S0 := 0;
    S4 := 1;
END_IF

```

Die Transition t1 ist eine Zeittransition, hierbei muss eine Einschaltverzögerung erstellt werden:

```

IF TON1.Q THEN
    S4 := 0;
    S2 := 1;
END_IF
TON1(IN:=S3 AND NOT S2 AND Input_2, PT:=t#2s);

```

Darauf folgt die Anweisungstransition, die den Zaehler um eins erhöht:

```

IF S3 AND NOT S0 THEN
    S3 := 0;
    S0 := 1;
    Zaehler := Zaehler +1;
END_IF

```

Zum Schluss kommt die Löschransition, in der die Stelle S0 gelöscht werden soll:

```

IF S2 AND NOT S3 AND (INPUT_1 OR INPUT_2) THEN
    S2 := 0;
    S3 := 1;
    S0 := 0;      //zu loeschende Stelle
END_IF

```

Der zu erwartende Programmcode wird mit der Endzuweisung abgeschlossen:

```

OUT_1 := S2 OR S4;
OUT_2 := S2;

```

Wird das in Abbildung 27 dargestellte SIPN in eine Text-Datei exportiert, erhält der Anwender das in Abbildung 28 dargestellte Ergebnis. Der Testergebnis entspricht den Erwartungen. Bei der Text Datei werden vermehrt Klammern gesetzt und die Reihenfolge der Variablendeklaration ist unterschiedlich. Diese Unterschiede verändern die Logik des Source Codes nicht. Das bedeutet, dass dieser Source Code funktionieren würde.

```

// Variablen

FUNCTION_BLOCK TEST
VAR_INPUT
Input_1 : BOOL;
Input_2 : BOOL;
END_VAR
VAR_OUTPUT
OUT_1 : BOOL;
OUT_2 : BOOL;
END_VAR
VAR
Zaehler : INT;
TON_t1 : TON;
S0 : BOOL := TRUE;
S2 : BOOL;
S3 : BOOL;
S4 : BOOL;
END_VAR

-----

// Implementation

// t0
IF (S0 AND NOT S4) AND (Input_1) THEN
    S0 := 0;
    S4 := 1;
END_IF

// t1
IF TON_t1.Q THEN
    S4 := 0;
    S2 := 1;
END_IF
TON_t1(IN:=(S4 AND NOT S2) AND (Input_2) , PT:=t#2.0s);

// t2
IF (S3 AND NOT S0) THEN
    Zaehler := Zaehler + 1;
    S3 := 0;
    S0 := 1;
END_IF

// t3
IF (S2 AND NOT S3) AND (Input_1 OR Input_2) THEN
    S2 := 0;
    S0 := 0;
    S3 := 1;
END_IF

OUT_1 := S2 OR S4;
OUT_2 := S2;

```

Abbildung 28: Testergebnis: Source Code in einer Text Datei

Der Testprozess stellt kein reales System dar und dient nur zum Überprüfen der Zeichenfunktionen. Um zu testen, ob sich das Programm bei komplexeren Systemen gleich verhält, wird das Modell einer Puffer Strecke verwendet. Beispielhaft wird diese in Abbildung 29 dargestellt. Im Anhang sind die Testergebnisse beigefügt. Es zeigt, dass mit dem Programm auch komplexere Anwendungen beschrieben werden können.

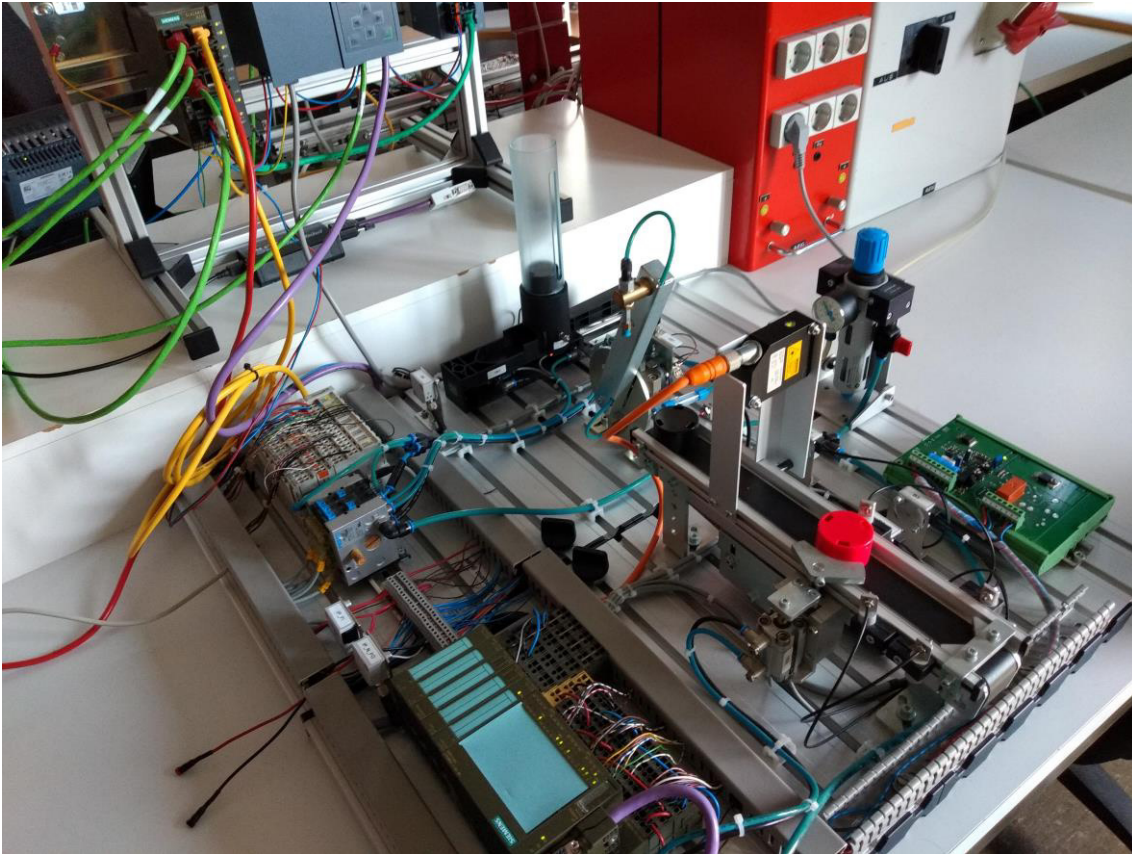


Abbildung 29: Puffer Strecke
Quelle: eigene Darstellung

7 Validierung

Im Folgenden wird untersucht, welche funktionalen Anforderungen umgesetzt wurden und welche nicht. Dafür werden die Testergebnisse mit den Anforderungen verglichen.

Tabelle 20: Validierung der allgemeinen Anforderungen

ID	Anforderung	Prio.		Implementierung
B1	Erstellen eines neuen SIPNs	++	✓	Wie beim Testen gezeigt wurde, kann mit dem Programm ein SIPN gezeichnet und bearbeitet werden.
B2	Zeichnen des SIPNs	++		
B3	Bearbeiten des SIPNs	++		
B4	SIPN speichern	+	✓	Zum Speichern und zum Laden werden Java Klassenbibliotheken genutzt, um das Modell zu speichern.
B5	SIPN laden	+		
B6	SIPN drucken	+	✓	Das Drucken wird wie in Abschnitt 5.3.9 beschrieben ermöglicht.
B7	Exportieren des SIPN als Bild	+	✓	Das Exportieren des SIPN als Bild unterstützt die Bildformate PNG und GIF.
B8	Exportieren des SIPN in normgerechten Code	++	✓	Das SIPN kann in einer Codesys Exportdatei oder in eine Text Datei umgewandelt werden.
B9	Normgerechten Code Importieren und SIPN erzeugen	o	×	
B10	Heran- und Herauszoomen des SIPNs	+	✓	In einem vorgegebenen Skalierungsbereich möglich.
B11	Simulation des SIPNs	o	×	
B12	Lebendigkeit des SIPNs prüfen	o	×	
B13	Erkennen hierarchischer Strukturen	o	×	
C1	Erstellen/bearbeiten/löschen von Variablen	++	✓	Das Programm gibt die Möglichkeit Variablen zu erstellen und zu bearbeiten. Die Möglichkeit eine Variable zu erstellen ist an dessen Verwendung geknüpft.
D1	Warnung bei Fehlerhafter Eingabe	+	✓	Typenfehler und fehlerhafte Schaltbedingungen werden abgefangen.

Quelle: eigene Darstellung

Bei den allgemeinen Anforderungen sind alle höher priorisierten Anforderungen erfüllt worden. Ähnlich ist das bei den Anforderungen, die an das Zeichnen gestellt wurden. In Tabelle 21 ist zu erkennen, dass die Triggertransition nicht implementiert wurde. Die Software ermöglicht es die Triggertransition nachträglich hinzuzufügen. Ebenfalls wurde das Kopieren von Zeichenobjekten nicht realisiert.

Tabelle 21: Validierung der Zeichnungsanforderungen

	Anforderung	Prio.		Implementierung
B2.1	Stelle hinzufügen	++	✓	Die Stelle wurde mit den Eigenschaften aus der Anforderungsanalyse (B2.1.1, B2.1.2, B2.1.3, B2.1.4) implementiert und kann dem SIPN hinzugefügt werden.
B2.2	Transition hinzufügen	++	✓	Die Transition erfüllt die Eigenschaften (B2.2.1-B2.2.5) aus der Anforderungsanalyse. Nur die Triggertransition und die Bedingungstransition (B2.2.4.5, B2.2.4.6) wurde nicht direkt implementiert. Die Priorisierung der Transitionen wird über den Namen der Transitionen geordnet.
B2.3	Gerichtete Kante hinzufügen	++	✓	Gerichtete Kanten können eingefügt werden.
B2.4	Kommentar hinzufügen	+	✓	Kommentare können in die Zeichnung eingefügt werden und mit Transitionen und Variablen verknüpft werden, sodass die Kommentare übernommen werden.
B2.5	Verschieben einzelner Objekte	++	✓	Das Verschieben von Zeichenobjekten und Labels ist möglich.
B2.6	Verschieben mehrerer Objekte	+	✓	Durch ein Auswahlfenster können mehrere Objekte ausgewählt und verschoben werden.
B2.7	Kopieren von Objekten	+	✗	
B2.8	Löschen von einzelnen Objekten	++	✓	Durch Auswahl der Zeichenobjekte können diese gelöscht werden. Die Auswahl für mehrere Objekte erfolgt wie beim Verschieben (B2.5).
B2.9	Löschen von mehreren Objekten	+	✓	

Quelle: eigene Darstellung

Die Variablentypen, die in der Anforderungsanalyse (3.2.3) beschrieben wurden, konnten alle implementiert werden. Weitere elementare Datentypen können schnell hinzugefügt werden.

Tabelle 22: Validierung der Anforderungen an die Variablen

ID	Anforderung	Prio.		Implementierung
C1	Anzeigen aller verwendeten Variablen	+	✓	Die Verwendeten Variablen werden abhängig vom Variablentyp in der Benutzeroberfläche dargestellt.
C2	Automatisches Erstellen von Variablen (z.B. Stellenname als Lokalen Boolean)	+	○	Falls eine Variable eindeutig in ihrer Verwendung ist werden diese automatisch dem Programm hinzugefügt. Jedoch ist dies nicht immer möglich.
C3	Automatisches Erstellen von Variablen nach Namenskonvention	○	✗	

Quelle: eigene Darstellung

Das Deklarieren der Variablen im Hintergrund kann Aufgrund der fehlenden Logik während der Eingabe nur in bestimmten Fällen realisiert werden.

Das Fehlerhandling lässt sich besser realisieren.

Tabelle 23: Anforderungen an das Fehlerhandling

ID	Anforderung	Prio.		Implementierung
D1.1	Kanten können nur von Transition zu Stelle oder von Stelle zu Transition gerichtet werden	++	✓	Diese möglichen Fehler werden, dem Anwender direkt beim Erzeugen des Fehlers mitgeteilt.
D1.2	Prüfen, ob Stellennamen einzigartig sind	++	✓	
D1.3	Prüfen, ob Transitionsnamen einzigartig sind	++	✓	
D1.4	Variablentyp und Variablenart sind passend bei der jeweiligen Verwendung	+	0	Der Variablentyp stimmt mit der Verwendung überein. Die Variablenart wird nur bei den Stellen und dem Ausgangszuweisungen überprüft.
D1.5	Das Petri Netz auf Schlingen überprüfen	+	✓	Schlingen werden gefunden.
D1.6	Das Petri Netz besitzt mindestens eine Initialstelle	++	✓	Ohne Initialstelle kann das SIPN nicht umgewandelt werden.

Quelle: eigene Darstellung

8 Fazit

Durch das Programm wird Studierenden ermöglicht, ein SIPN zu zeichnen. Dieses SIPN kann in normgerechten Source Code umgewandelt und in unterschiedlichen Entwicklungsumgebungen verwendet werden. Dabei wird das SIPN als Funktionsbaustein für die SPS-Entwicklungsumgebung „Codesys“ bereitgestellt. Damit ist das formulierte Ziel dieser Arbeit erreicht.

Im Vergleich zum manuellen Zeichnen bietet dieses Programm nicht nur Zeitersparnis beim Codieren des SIPNs, sondern auch eine Unterstützung für das fehlerfreie Erstellen des SIPNs. Außerdem ist das nachträgliche Bearbeiten durch das Verschieben, Löschen und Bearbeiten von Zeichenobjekten leichter als beim Zeichnen mit dem Stift.

Auf der anderen Seite ist das manuelle Zeichnen schneller als das Zeichnen mittels dieses Programms. Um dem entgegenzuwirken wurden Methoden entwickelt, wie zum Beispiel das Berechnen von Anschlusspunkten von gerichteten Kanten an den Zeichenobjekten (Kapitel 5.3.3), um den Zeitaufwand das SIPN zu zeichnen, möglichst gering zu halten.

Zudem können die Studierenden das SIPN als Bild speichern oder drucken und so für die Dokumentation des Steuerungsentwurfs verwenden.

9 Ausblick

Das Programm bietet eine Vielzahl von Erweiterungsmöglichkeiten. Einige dieser Erweiterungsmöglichkeiten sind bereits in der Anforderungsanalyse beschrieben worden, wurden jedoch innerhalb dieser Arbeit nicht umgesetzt. Darunter fallen komplexere Funktionalitäten wie zum Beispiel das Simulieren des SIPNs oder das Umwandeln eines Strukturierten Text in die dazugehörige Petri Netz Zeichnung.

- Normgerechten Code Importieren und SIPN erzeugen (Anforderung B9)
- Simulation des SIPNs (Anforderung B11)
- Lebendigkeit des SIPNs prüfen (Anforderung B12)
- Erkennen und darstellen hierarchischer Strukturen (Anforderung B13)
- Erweiterung der Export Formate
- Erweiterung der Variablenarten

Um das SIPN in verschiedene Formate umzuwandeln muss das Programm nur um die notwendige Export Strategie erweitert werden. Eine Erweiterung der Variablenarten stellt ebenfalls keine große Herausforderung dar und wurde deshalb nicht implementiert, da es für den spezifischen Studiengang von geringer Bedeutung war.

Literaturverzeichnis

Abts, Dietmar. 2014. *Grundkurs JAVA*. Ratingen : Springer Vieweg, 2014. ISBN 978-3-658-07968-0.

Aspern, Jens von. 2003. *SPS-Softwareentwicklung mit Petrinetzen*. Berlin : VDE VERLAG GMBH, 2003. 3-8007-2728-5.

Bachmann, Nadine. 2012. Bildformate . *netzmarginaleien.de* . [Online] 28. 11 2012. [Zitat vom: 20. 06 2019.] <https://www.netzmarginalien.de/bildformate>.

Balzert, Helmut. 2011. Architektur- und Entwurfsmuster. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. s.l. : Spektrum Akademischer Verlag, 2011, S. 37-107.

Buschmann, Frank, et al. 1998. *Patternorientierte Softwarearchitektur: Ein Pattern System*. s.l. : Pearson Deutschland GmbH, 1998.

Carbannelle, Pierre. 2019. PYPL. *PYPL PopularitY of Programming Language*. [Online] 06 2019. [Zitat vom: 09. 06 2019.] <http://pypl.github.io/PYPL.html>.

Codesys. Startseite: Codesys. *Codesys*. [Online] [Zitat vom: 08. 05 2019.] <https://de.codesys.com/>.

DIN EN 61131-3:2014-06. Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen. s.l. : Berlin: Beuth.

Eilbrecht, Karl und Starke, Gernot. 2019. *Patterns kompakt*. Karlsruhe : Springer-Verlag GmbH, 2019.

Gammer, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. s.l. : Addison-Wesley, 1994.

Goll, Joachim. 2017. *Pattern-orientierte Software-Architektur: Ein Pattern-System*. Esslingen : Springer Vieweg , 2017. ISBN 978-3-658-20055-8.

Goll, Joachim und Dausmann, Manfred. 2013. Architekturmuster. [Buchverf.] Prof. Dr. Manfred Dausmann Prof. Dr. Joachim Goll. *Architektur- und Entwurfsmuster der Softwaretechnik*. Wiesbaden : Springer Fachmedien, 2013.

Günter, Lisa. motocms.com. [Online] [Zitat vom: 11. 06 2019.] <https://www.motocms.com/blog/de/bildformate-vor-und-nachteile/>.

Hasselbring, Wilhelm. 2006. gi.de. *gi.de Software-Architektur*. [Online] Springer-Verlag 2006, 22. 06 2006. [Zitat vom: 16. 06 2019.] <https://gi.de/informatiklexikon/software-architektur/>.

Lunze, Jan. 2012. Beschreibung diskreter Systeme. *Automatisierungstechnik*. München : Oldenbourg Wissenschaftsverlag GmbH, 2012.

Martin, Robert C. 2002. Chapter 9 OCP: The Open—Closed Principle. *Agile software development: principles, patterns, and practices*. s.l. : Prentice Hall, 2002, S. 99-110.

Oracle. 2017. docs.Oracle. *docs.Oracle Formatted Text Field*. [Online] Oracle, 2017. [Zitat vom: 12. 06 2019.] <https://docs.oracle.com/javase/tutorial/uiswing/components/formattedtextfield.html>.

—. **2018.** oracle.com. *docs.oracle/package-summary*. [Online] Oracle, 2018. [Zitat vom: 11. 06 2019.] <https://docs.oracle.com/javase/7/docs/api/javax/imageio/package-summary.html>.

Petry, Jochen. 2011. *IEC 61131-3 mit CoDeSys V3: Ein Praxisbuch für SPS-Programmierer*. Kempten : 3S-Smart Software Solutions GmbH, 2011.

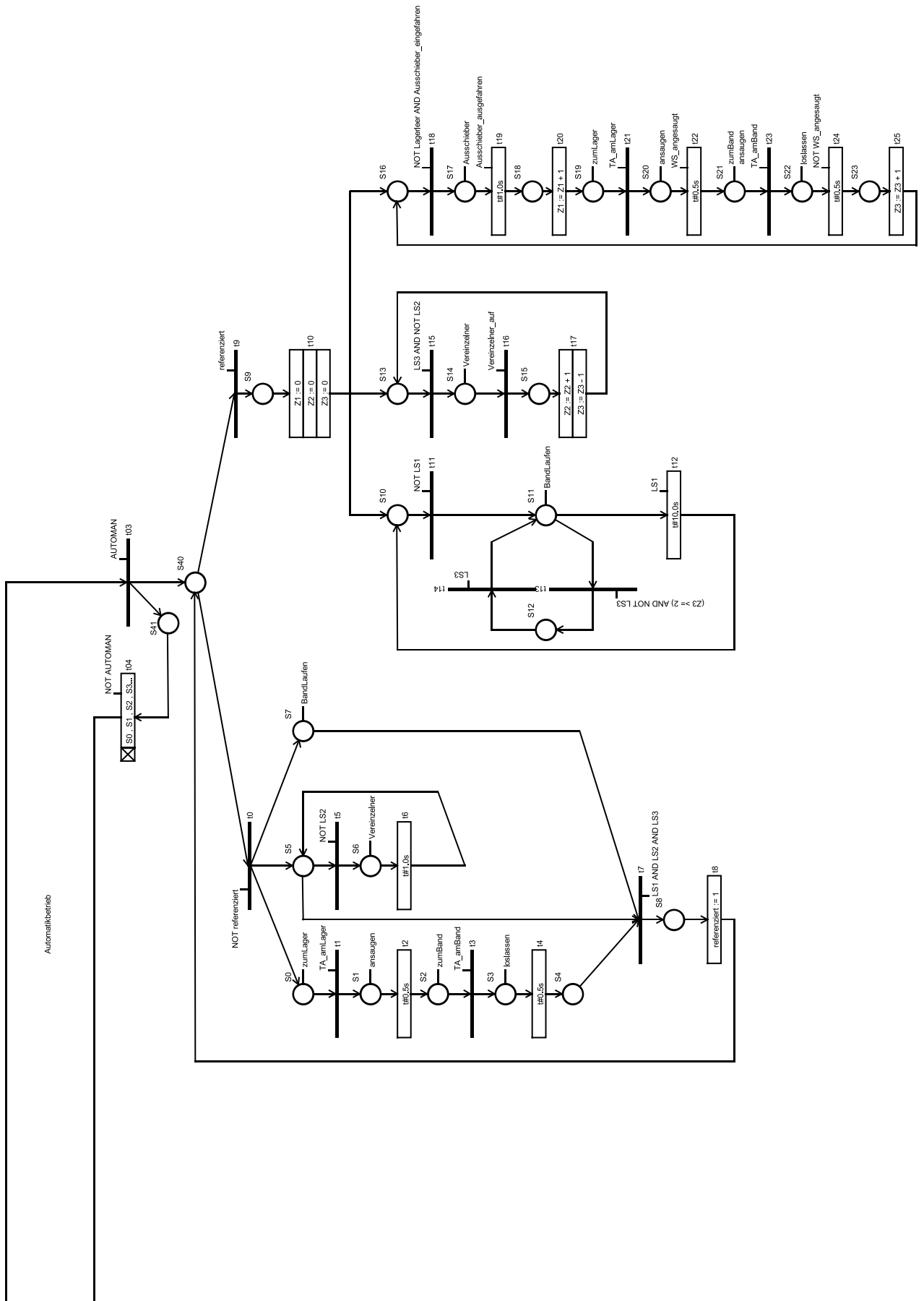
Schnieder, Eckehard. 1999. *Methoden der Automatisierung*. Braunschweig/Wisbaden : Vieweg, 1999.

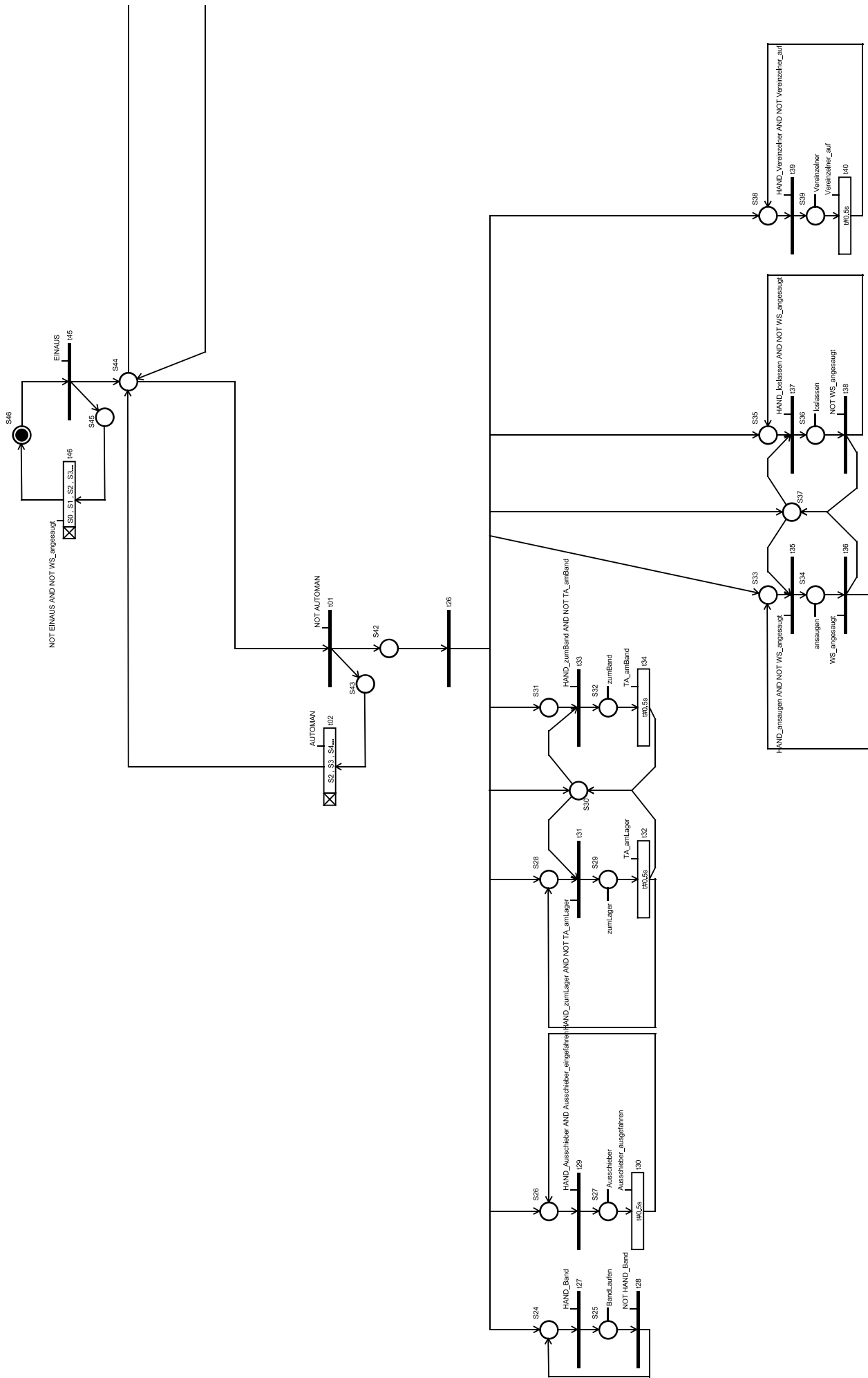
Siemens. TIA Portal: Siemens. *Siemens*. [Online] Siemens. [Zitat vom: 08. 05 2019.] <https://new.siemens.com/global/de/produkte/automatisierung/industrie-software/automatisierungs-software/tia-portal.html>.

Tiegelkamp, Michael und John, Karl-Heinz. 2009. *SPS-Programmierung mit IEC 61131-3*. s.l. : Springer-Verlag Berlin Heidelberg, 2009. ISSN: 2512-5281.

Wallner, Michael. 2016. netzstrategen.com. [Online] 20. 01 2016. [Zitat vom: 11. 06 2019.] <https://netzstrategen.com/blog/webwissen-ist-eine-vektorgrafik-und-wie-erstelle-ich-sie>.

A: SIPN der Pufferstrecke





B: Datenträger

Der Datenträger enthält:

- Bachelorthesis in digitaler Form im PDF „BA_Hartig.pdf“
- Der in dieser Arbeit entwickelte Programmcode Ordner „SIPN_ST“
- Die aus Abschnitt 6.3 entstandenen Testergebnisse Ordner „Testergebnisse“

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, Justus Kilpatrick Hartig, gemäß der allgemeinen Prüfungsordnung § 16 Abs. 5 APSO-TI-BM, dass ich die vorliegende Bachelorarbeit mit dem Thema „Entwicklung eines Werkzeugs zur Modellierung und Source-Code-Erzeugung von steuerungstechnisch interpretierten Petri-Netzen für speicherprogrammierbare Steuerungen“ ohne fremde Hilfe verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg, 28.06.2019

Ort, Datum

Unterschrift