Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Christian Meier

Evaluierung und Anwendung unterschiedlicher Methoden zur automatischen Codegenerierung bei ereignisdiskreten Steuerungen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Automatisierung
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Florian Wenck
Zweitgutachter: Prof. Dr.-Ing Lutz Leutelt

Abgegeben am 28.07.2013

Christian Meier

Thema der Masterarbeit
Evaluierung und Anwendung unterschiedlicher Methoden zur automatischen Codegenerierung bei ereignisdiskreten Steuerungen
Stichworte
DES, Codegenerierung, AWL, Anweisungsliste, ereignisdiskrete Systeme
Kurzzusammenfassung
Diese Thesis baut auf einer vorhandenen lauffähigen Arbeit auf, in der ein Software-Tool zur automatischen Codegenerierung von ereignisdiskreten Systemen entwickelt wurde. Diese Anwendung generiert automatisch einen AWL-Code aus einer in DESTool modellierten Anlage. In dieser Arbeit wird zunächst nach verschiedenen Ansätzen zur Codegenerierung für ereignisdiskrete Systeme recherchiert. Sodann wird eine Auswahl dieser Ansätze genauer analysiert und evaluiert, Vor- und Nachteile werden dabei gegenüber gestellt. Auf Basis dieser Evaluierung werden dann ausgewählte Ansätze in das Software-Tool eingepflegt. Es soll auch bewertet werden inwiefern sich der generierte Code von demjenigen unterscheidet, der mit dem original Software-Tool generiert wurde.

Christian Meier

Title of the paper
Evaluation and application of different methods for automatic code generation for discrete event control
Keywords
DES, code generation, AWL, instruction list, discrete event systems
Abstract
This Thesis builds on an existing executable work, in which a software tool for automatic code generation of discrete event systems has been developed. This application automatically generates an AWL code of a discrete event system modeled in DESTool. First in this paper is a research by different approaches for code generation for discrete event systems. Furthermore a selection of these approaches is analysed and evaluated, the advantages and disadvantages are compared with each other. Based on this evaluation the selected approaches will be entered into the existing software tool. It should also be analysed in which form the new generated code differs from the code generated by the basic software tool.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Ziel der Arbeit

1.1 Einführung

Automatische Codegenerierung ist ein weit verbreitetes Thema, welches in einer Vielzahl von Fachartikeln behandelt wird. Die in diesen Artikeln verwendete Steuerungssynthese hat den Vorteil, dass Risiken, die bei einer intuitiven Programmierung auftreten, reduziert werden.
Es gibt unterschiedlichste Ansätze, um eine Steuerung, die mittels Steuerungssynthese entwickelt wird, in eine Programmiersprache einer SPS zu konvertieren. Die Theorie dieser Ansätze kann auf verschiedene Arten variieren. Es können Unterschiede in der Programmiersprache auftreten, die von „Ladder Logic" über „Function Charts" bis zu „Funktionsplan" variieren kann, oder beim Ansatz der Modellierung (Petri-Netze, Automaten) des zu konvertierenden Modells.
Für die Evaluierung der in dieser Arbeit enthaltenen Ansätze werden in Kapitel 2 die Grundlagen der ereignisdiskreten Systeme sowie die Grundlagen der Supervisory Control Theory eingeführt. In Kapitel 3 wird der Begriff der Codegenerierung erläutert und ausgewählte Methoden zur Codegenerierung vorgestellt. Kapitel 4 beschreibt eine Fertigungszelle, dessen Spezifikationen für den Vergleich herangezogen werden. Diese Spezifikationen sind der Arbeit von Urland [Url12] entnommen. In Kapitel 5 wird die Adaption des Programmes „DES2IEC" beschrieben und die Ergebnisse der Generierung dem Ergebnis aus [Url12] gegenübergestellt. Kapitel 6 beinhaltet die Evaluierung des generierten AWL-Codes. Abschließend werden in Kapitel 7 die Ergebnisse zusammengefasst und ein Ausblick gegeben.

1.2 Aufgabenstellung

Diese Thesis baut auf einer vorhandenen lauffähigen Arbeit auf, in der ein Software-Tool zur automatischen Codegenerierung von ereignisdiskreten Systemen entwickelt wurde. Diese Anwendung generiert automatisch einen AWL-Code aus einer in DESTool modellierten Anlage. In dieser Arbeit werden zunächst verschiedene Ansätze zur Codegenerierung für ereignisdiskrete Systeme recherchiert. Sodann wird eine Auswahl dieser Ansätze genauer analysiert und evaluiert, Vor- und Nachteile werden dabei gegenüber gestellt. Auf Basis dieser Evaluierung werden dann ausgewählte Ansätze in das Software-Tool eingepflegt.
Es soll auch bewertet werden inwiefern sich der generierte Code von demjenigen unterscheidet, der mit dem original Software-Tool generiert wurde.

1.3 Zielsetzung

Das Ziel dieser Abschlussarbeit ist es, nach besseren Ansätzen zur Codegenerierung von ereignisdiskreten Systemen zu suchen. Die Modulierung soll möglichst mit Automaten realisiert werden. Die Ansätze werden in Hinsicht auf Codemenge, Codeverständlichkeit, Struktur, Zykluszeit und Speicherbedarf untersucht.

Kapitel 2

Supervisory Control Theory

In diesem Kapitel wird auf die Grundlagen der ereignisdiskreten Systeme (DES, engl. Discrete Event Systems) und der Supervisory Control Theory (SCT) eingegangen. Zu Beginn werden die mathematischen Grundlagen ereignisdiskreter Systeme vorgestellt und deren Modellierung erläutert, die mittels der Automatentheorie realisiert wird. Des Weiteren werden unterschiedliche Methoden und mathematische Techniken der Supervisory Control Theory behandelt. Abschließend wird auf verschiedene strukturierte Entwurfsverfahren für DES eingegangen. Die Automatentheorie ist grundlegend für die Modellierung von ereignisdiskreten Systemen und die Supervisory Control Theory.

2.1 Mathematische Grundlagen

Zur Modellierung ereignisdiskreter Systeme sind Kenntnisse der Sprachentheorie erforderlich. Zuerst wird die in dieser Arbeit genutzte Sprachklasse erläutert und Ihr Bezug zur Anwendung bei DES dargestellt. Des Weiteren wird die Automatentheorie zur Modellierung ereignisdiskreter Systeme ebenfalls in diesem Kapitel behandelt.

2.1.1 Sprachen

Reguläre Sprachen.
1957 stufte Noam Chomsky formale Sprachen differenziert nach ihrer Ausdrucksmächtigkeit ein. Diese wird durch die Grammatik einer Sprache festgelegt, welche wiederum durch Regeln bestimmt, was durch diese Sprache dargestellt werden kann. Die Grammatik enthält zwei Hauptelemente, die Terminalsymbole (Elemente des Sprachalphabets, z.B. a,b,c) und die Nicht-Terminalsymbole (Hilfsvariablen um Sequenzen von Terminalsymbolen zu erstellen, z.B. A,B,C). Die Grammatik formt Wörter einer Sprache indem diese Regeln angewendet werden. Die Klasse der regulären Sprachen ist folgendermaßen nach [Wen06]

definiert:

> **Definition 2.1 (Reguläre Sprachen (Typ3)).** Eine Sprache heißt regulär, wenn sie durch eine reguläre Grammatik beschrieben ist. Eine reguläre Grammatik lässt nur Ersetzungen von Nicht-Terminalsymbolen zu einem einzelnen Terminalsymbol oder einem Terminalsymbol gefolgt von einem Nicht-Terminalsymbol zu:
> $A \rightarrow a$, $A \rightarrow aB$.

Deterministische und nicht deterministische endliche Automaten erfassen die regulären Sprachen.

Grundlagen der Sprachentheorie.
Formale Sprachen dienen der Darstellung des Verhaltens von DES. Im Kontrast zur Darstellung mittels Automaten kann die Darstellung mittels formaler Sprachen unendlich sein. Nachfolgend werden notwendige Begriffe für die Definition formaler Sprachen erläutert.
Ein Alphabet ist eine endliche, nichtleere Menge, die im allgemeinen mit Σ bezeichnet wird, und später die Ereignismenge eines Automaten darstellt. Die Elemente von Σ werden Symbol, Buchstabe oder Zeichen genannt. Man bezeichnet eine Menge Σ die eine endliche Anzahl von Elementen enthält, als endliches Alphabet. Eine Folge die keine Symbole enthält wird leerer String genannt und durch $\varepsilon \notin \Sigma$ definiert. Werden endlich viele Symbole aneinander gereiht ($\sigma_1\sigma_2\sigma_3$ mit $\sigma_i \in \Sigma$) so spricht man von einer Verkettung oder Konkatenation. Die durch eine Konkatenation entstandenen Buchstabenketten werden Wörter genannt. Durch die Verkettung von beliebigen aber endlich vielen Symbolen kann eine sehr große Anzahl von Wörtern gebilden werden. Die daraus resultierende Menge wird Σ^* genannt. Das leere Wort und das leere Zeichen werden beide ε genannt und komplettieren die Menge Σ^*. Das leere Zeichen kann ohne Konsequenzen an beliebiger Stelle einer Zeichenfolge eingefügt oder gestrichen werden. Das heißt, dass das leere Zeichen keinen Zustandswechsel hervorruft. Die Menge aller Zeichenfolgen des Alphabets Σ ohne das leere Wort wird Σ^+ genannt. Die Kleene-Hülle von Σ ist definiert als $\Sigma^* = \{\varepsilon\} \cup \Sigma^+$. Es ist zu beachten, dass $\{\varepsilon\}$=0 gilt. Die Konkatenation von Strings *cat:* $\Sigma^* x \Sigma^* \rightarrow \Sigma^*$ ist definiert als:

$$cat(\varepsilon, s) = cat(s, \varepsilon = s,\ s \in \Sigma^* \tag{2.1}$$

$$cat(s,t) = st,\ s,t \in \Sigma^* \tag{2.2}$$

Die Länge $|s|$ eines Strings $s \in \Sigma^*$ist über $|\varepsilon| = 0$ und $|s| = k$ für $s = \sigma_1 ... \sigma_k \in \Sigma^+$ definiert. Dabei sei $s = cat(t, \sigma) = t\sigma$ mit $t \in \Sigma^*$, $\sigma \in \Sigma$, dann ist $\overline{s} = t$ der Präfix von s. Für $s = u\sigma v$

mit $u \in \Sigma^*$, $\sigma \in \Sigma$, $v \in \Sigma^*$ ist v der Suffix von s ab dem Zeichen σ. Für $s = utv$ mit $u,t,v \in \Sigma^*$ ist t ein Substring von s.
Die beiden Sprachen Σ^* und $\varnothing$ sind ebenfalls formale Sprachen über Σ, wie Definition 1.1 zeigt. B sei ein deterministischer endlicher Automat. Dann erkennt dieser genau die reguläre Sprache L_B die wiederum alle Strings $s \in \Sigma^*$ beinhaltet, welche zu einem markierenden Zustand $z \in Z_m$ in B führen.

Formale Sprachen.
Die formale Sprache ist wie folgt nach [Wen06] definiert.

> **Definition 2.2 (formale Sprache).** Eine formale Sprache über dem Alphabet Σ ist eine beliebige Teilmenge $L \subseteq \Sigma^*$. Sie ist regulär (Typ 3), genau dann, wenn sie von einem endlichen Automaten erkannt wird.

$$L \subseteq \Sigma^* \tag{2.3}$$

Auf das Alphabet bezogen ist L die formale Sprache über dem Alphabet Σ. Dieser Definition nach sind die leere Menge $\varnothing$ wie auch die Menge $\{\varepsilon\}$ (enthält nur das leere Zeichen) Sprachen.
Die Theorie der formalen Sprachen hat den Vorteil, dass eine Vielzahl von Operationen geboten werden, welche größtenteils aus der Mengenlehre und Arithmetik stammen.
Aus diesem Grund werden zur Beweisführung oft formale Sprachen den Automaten vorgezogen.

Operationen auf Sprachen.
Da Sprachen zur Familie der Mengen gehören, kann ein Teil der Operationen aus der Mengenlehre auf Sprachen angewandt werden. Für zwei Sprachen $L_1, L_2 \in \Sigma^*$ sind Vereinigungs- und Schnittmenge, die Differenz und das Komplement ($L \subseteq \Sigma^*$) wie folgt definiert:

$$L_1 \cup L_2 = \{s | (s \in L_1) \vee (s \in L_2)\} \tag{2.4}$$

$$L_1 \cap L_2 = \{s | (s \in L_1) \wedge (s \in L_2)\} \tag{2.5}$$

$$L_1 \backslash L_2 = \{s | (s \in L_1) \wedge (s \notin L_2)\} \tag{2.6}$$

$$L^c = \Sigma^* \backslash L \tag{2.7}$$

Weitere Operationen die in der Mengenlehre eine nicht so große Bedeutung haben, jedoch in der Sprachentheorie häufig genutzt werden sind nach [Wen06] folgendermaßen definiert:

Definition 2.3 (Konkatenation zweier Sprachen). Es seien $L_1, L_2 \subseteq \Sigma^*$, dann ist ihre Konkatenation definiert als:

$$L_1, L_2 = \{s \in \Sigma^* | (s = s_1 s_2) \vee (s_1 \in L_1) \vee (s_2 \in L_2)\} \tag{2.8}$$

Ein String ist demzufolge in $L_1 L_2$ enthalten, wenn dieser aus einer Konkatenation eines Strings aus L_1 und eines Strings aus L_2 entwickelt wurde.

Definition 2.4 (Präfix-Hülle einer Sprache). Es sei $L \subseteq \Sigma^*$, dann ist ihre Präfix-Hülle definiert als:

$$\overline{L} = \{s \in \Sigma^* | \exists t \in \Sigma^* (st \in L)\}. \tag{2.9}$$

Die Präfix-Hülle $\overline{L}$ enthält somit die Präfixe aller Strings $s \in L$. Der Einschluss $L \subseteq \overline{L}$ ist eine Tautologie. Wenn also $L = \varnothing$ folgt $\overline{L} = \varnothing$und wenn $L \neq \varnothing$gilt, so ist $\varepsilon \in \overline{L}$. Gilt $L = \overline{L}$, so ist die Sprache prafix-abgeschlossen.

Definition 2.5 (Kleene-Hülle einer Sprache). Es sei $L \subseteq \Sigma^*$, dann ist ihre Kleene-Hülle definiert als

$$L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup .... \tag{2.10}$$

Ein String $s \in \Sigma^*$geht aus einer Konkatenation einer endlichen Anzahl von Strings aus L hevor. Die Kleene-Hülle L^* weist darüber hinaus den leeren String $\{\varepsilon\}$ auf. Die Kleene-Hülle der leeren Menge und der nicht leeren Menge des leeren Strings werden durch $\varnothing^* = \{\varepsilon\}$und $\{\varepsilon\}^* = \{\varepsilon\}$ repräsentiert.

Die Klasse der regulären Sprachen besitzt Abschlusseigenschaften betreffend einiger der oben genannten Operationen. Der anschließende Satz summiert diese Operationen und richtet sich nach [Sch95].

Satz 2.1 (Abgeschlossenheit regulärer Sprachen) Die Klasse der regulären Sprachen ist abgeschlossen unter Vereinigung, Schnitt, Komplement, Konkatenation, Bildung der Kleene- und der Präfix-Hülle.

Die oben definierten Abschlusseigenschaften erleichtern die Anwendung der Operationen von regulären Sprachen. Somit gilt für die Sprachen L_1L_2 (über Σ), dass $L_1 \cup L_2$, $L_1 \cap L_2$, L_1^c, L_1L_2, L_1^* und $\overline{L_1}$ wieder reguläre Sprachen (über Σ) bilden. Diese Merkmale sind wichtig im Hinblick auf die Analyse und die Komposition von logischen DES und deren Steuerungsentwurf.

2.1.2 Automaten

In diesem Abschnitt wird die Automatentheorie als Beschreibungsform von DES erläutert. Sie befasst sich mit der formalen mathematischen Beschreibung und Untersuchung von Automaten, das heißt von Modellen diskreter, sequentieller, informationsverarbeitender Systeme. In dieser Arbeit helfen Automaten bei der Darstellung der Systemstruktur logischer DES. Der Zustand stellt den „Status" des Systems dar, und im Kontrast zu einem passiven Erkenner generiert der Generator aktiv Ereignisse, welche Zustandsübergänge hervorrufen können. Nachfolgend liegt der Augenmerk auf den deterministischen Automaten *[Lun12].*

> **Definition 2.6 (autonomer deterministischer Automat).** Ein autonomer Automat A wird durch seine Zustandsmenge $Z = 1,2,3...N$ und seine Zustandsübergangsfunktion $G : Z \to Z$ beschrieben. Die Zustandsübergangsfunktion G gibt für jeden Zustand $z \in Z$ den Nachfolgezustand $z' \in Z$ an: $z' = G(z)$. Da G eine Funktion ist, ordnet sie jedem Zustand z eindeutig einen Nachfolgezustand z' zu und so nennt man den Automaten deterministisch.

Deterministische endliche Automaten.
Automaten werden endlich genannt wenn sie eine endliche Zustandsmenge besitzen. Die in dieser Arbeit verwendeten deterministischen endlichen Automaten werden im weiteren Verlauf dieser Arbeit Automaten oder Generatoren genannt und sind folgendermaßen nach [Wen06] definiert und :

> **Definition 2.7 (deterministische endliche Automaten (DFA)).** Ein DFA G ist ein Quintupel $G = (X, \Sigma, \delta, x_0, X_m)$ mit der endlichen Zustandsmenge X, dem endlichen Eingabealphabet Σ, der Zustandsübergangsfunktion $\delta : X \times \Sigma \to X$, dem Anfangszustand $x_0 \in X$ und der Menge der akzeptierenden Zustände $X_m \subseteq X$.

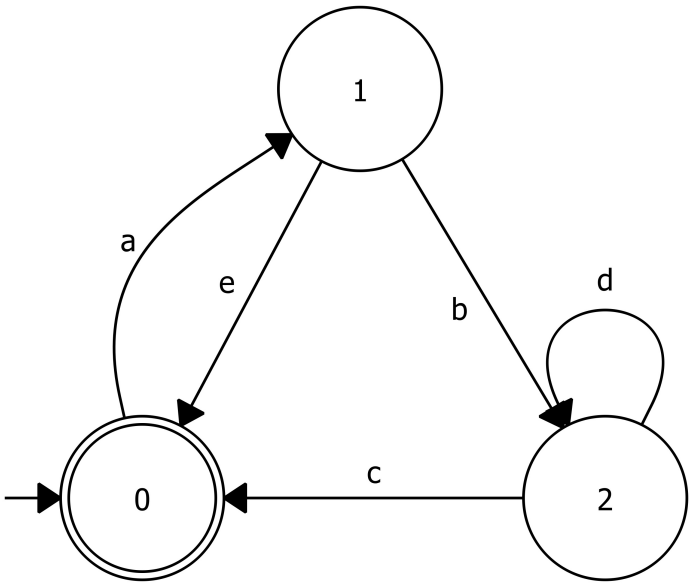


Abbildung 2.1: Beispiel eines Generators

Die Zustandsmenge X wird gemeinhin durchnummeriert, somit ist $X = 1,2,3,...,N$. Alle möglichen Ereignisse werden in Σ gesammelt und werden Ereignismenge genannt. Die Ereignismenge besteht aus allen Ereignissen $\sigma \in \Sigma$ [Lun12]. Mit der Zustandsübergangsfunktion werden dieselbigen beschrieben, sie stellt demnach das dynamische Verhalten des Systems dar. Es weist jedem Ereignis $x \in X$ mit jedem Ereignis $\sigma \in \Sigma$ einen Folgezustand zu. Dadurch gilt:

$$x' = \delta(x,\sigma), x' \in X. \tag{2.11}$$

Durch die feste Zuordnung eines Zustands x zu genau einem Folgezustand x' infolge der Funktion δ, heißt ein Generator deterministisch. Für den Fall, dass das Verhalten nicht eindeutig beschrieben werden kann, werden nichtdeterministische Generatoren verwendet. Hier wird an Stelle von Zustandsübergangsfunktionen, eine Zustandsübergangrelation gebildet, welche eine mögliche Menge an Folgezuständen festlegt.

Adjazenzmatrix.
Zustandsübergänge können grafisch dargestellt werden (siehe 2.1), oder mit Hilfe der Adja-

zenzmatrix A. Sie hat die Größe $N \times N$ Elemente, wobei durch N die Anzahl der Zustände gekennzeichnet ist. Ein Element g_{ij} der Matrix A ist genau dann σ wenn für den Übergang von Zustand j nach Zustand i eine Funktion δ, definiert ist. Das Ausrufezeichen steht nachfolgend für „definiert".

$$x_i = \delta(x_j, \sigma)! \tag{2.12}$$

Ist keine Zustandsübergangsfunktion δ von j nach i bestimmt, so gilt

$$x_i = \neg\delta(x_j, \sigma)!. \tag{2.13}$$

In diesem Fall steht an entsprechender Stelle in A eine Null. Aus dem Generator in Abbildung 2.1 gehen folgende Zustandsübergangsfunktionen hervor

$$x_0 = \delta(x_2, c), \tag{2.14}$$
$$x_1 = \delta(x_0, a), \tag{2.15}$$
$$x_0 = \delta(x_1, e), \tag{2.16}$$
$$x_2 = \delta(x_2, d), \tag{2.17}$$
$$x_2 = \delta(x_1, b), \tag{2.18}$$

Aus den Zustandsübergangsfunktionen des Generators wird die Adjazenzmatrix gebildet. Ist eine Schleife an einem Zustand x_i wird diese „Selfloop" genannt und das entsprechende Diagonalelement in A ist ungleich Null.

$$A = \begin{pmatrix} 0 & a & 0 \\ e & 0 & b \\ c & 0 & d \end{pmatrix} \tag{2.19}$$

Definiert wird die reguläre Sprache $L(G)$ und damit verbunden das Verhalten von G durch die Vollständigkeit aller Strings, die G von x_0 aus im Stande ist zu generieren. Infolge dessen gilt

$$L(G) = \{s \in \Sigma^* | \delta(x_0, s)!\}. \tag{2.20}$$

Alle Strings, die von x_0 einen markierenden Zustand zur Folge haben, definieren das markierende Verhalten $L_m(G)$

$$L_m(G) = \{s \in L(G) | \delta(x_0, s) \in X_m\}. \tag{2.21}$$

2.2 Modellierung und Analyse von DES

Der nächste Abschnitt geht auf die Modellierung ereignisdiskreter Systeme mit Hilfe von Generatoren ein.

Definition 2.8 (Generator). Ein Generator

$$G = (X, \Sigma, \delta, z_0, X_m) \tag{2.22}$$

ist ein DFA mit einer nicht notwendigerweise totalen Zustandsübergangsfunktion $\delta : X \times \Sigma \to X$.

Die nachfolgend behandelten Operationen dienen dazu, aus Generatoren sogenannte Kompositionsgeneratoren (Generatorennetze) zu erstellen. Mittels der Kompositionsoperatoren wird aus Teilmodellen ein zusammengefügtes Gesamtmodell gestaltet. Die einzelnen Generatoren werden mittels gleicher Ereignisse synchronisiert und stellen so eine Verbindung der Teilgeneratoren dar. Aufgrund der Kompositionsoperatoren ist es möglich eine Strecke in modularer Form zu modellieren. Nachfolgend wird die Produktkomposition und die parallele Komposition erläutert.

Im Weiteren Verlauf wird auf die oben genannten Operationen für zwei Komponenten eingegangen. Diese Prozeduren sind jedoch auf beliebig viele Komponenten erweiterbar. Zuerst erfolgt eine Aufteilung der Alphabete in private und gemeinsame Ereignisse. Aus den Alphabeten Σ_1 und $\Sigma_2$folgt somit für gemeinsame Ereignisse

$$\Sigma_{CE} = \Sigma_1 \cap \Sigma_2 \tag{2.23}$$

und für die privaten Ereignisse

$$\Sigma_{PE} = (\Sigma_2 - \Sigma_1) \cup (\Sigma_1 - \Sigma_2). \tag{2.24}$$

Produkt zweier Generatoren (Produktkomposition).
Die Produktkomposition oder auch Strenge Synchronisation genannt (SPC, Strict Product

Komposition) beschreibt das Produkt zweier Generatoren

$$G_x = G_1 \times G_2 \tag{2.25}$$

und synchronisiert die Generatoren G_1 und G_2 auf ihre gemeinsamen Ereignisse. Gleichzeitig wird jedes Auftreten von privaten Ereignissen untersagt. Somit kann G_x nur Ereignisse generieren die in G_1 und in G_2 enthalten sind. Der aus dem Produkt entstandene Generator wird durch die Gleichungen 2.25, 2.26, 2.27 und 2.28 beschrieben:

Das zum Generator zugehörige Alphabet wird durch folgende Gleichung dargestellt:

$$\Sigma_x = \Sigma_1 \cap \Sigma_2 \tag{2.26}$$

Aus dem kartesischen Produkt der Teilgeneratoren resultiert die Zustandsmenge des Produktgenerators. Beiden Teilgeneratoren ist es nur noch möglich im „totalen Gleichschritt" [Wen06] zu schalten.

$$X = X_1 \times X_2 \tag{2.27}$$

In gleicher Weise ergeben sich die markierten Endzustände. Es ist nur möglich einen Endzustand zu erreichen, wenn in beiden Teilgeneratoren markierte Zustände aktiv sind.

$$X_m = X_{m1} \times X_{m2} \tag{2.28}$$

Nach der Produktkomposition wird die Zustandsübergangsfunktion folgendermaßen festgelegt:

$$\delta(x,\sigma) = \begin{cases} \delta(x_1,\sigma) \times \delta_2(x_2,\sigma), & wenn\, \delta_1(x_1,\sigma)! \wedge \delta_2(x_2,\delta)! \\ nicht\, definiert & sonst. \end{cases} \tag{2.29}$$

Die Gleichung 2.29 besagt, dass nur dann ein Zustandsübergang möglich ist, wenn die Zustandsübergangsfunktionen δ_1 und $\delta_2$für genau dieses σ in beiden Komponenten festgelegt ist. Ein Zustandswechsel kann nicht hervorgerufen werden, wenn eine der beiden Funktionen δ_1 oder δ_2 nicht definiert ist. Die Sprache des Kompositionsgenerators wird wie folgt definiert:

$$L(G_1 \times G_2) = L(G_1) \cap L(G_2). \tag{2.30}$$

Sie besteht exklusiv aus gemeinsamen Zeichenketten, die von beiden Generatoren generiert werden können. Private Ereignisse sind ausgeschlossen. Die Produktkomposition wird häufig bei dem Zusammenführen von Strecke und Steuerung genutzt.

Parallele Komposition (SYPC, Synchronous Product).
Die parallele Komposition generiert, genauso wie die Produktkomposition, gemeinsame Ereignisse nur wenn beide Generatoren diese in ihrem aktiven Zustand auszulösen im Stande sind. Es besteht hingegen ein Unterschied in der Behandlung mit privaten Ereignissen. Das Auftreten privater Ereignisse $\sigma \in \Sigma_{PE}$ wird durch die parallele Komposition nicht unterbunden. Die Teilgeneratoren können zu jedem Zeitpunkt private Ereignisse generieren, ohne dass die jeweils andere Komponente diesen Zustandsübergang verbieten kann. Die parallele Komposition synchronisiert die Komponenten G_1und G_2 lediglich auf ihre gemeinsamen Ereignisse. Der aus der parallelen Komposition resultierende Kompositionsgenerator kann nach [Url12] durch das Quintupel

$$G_{||} = (X, \Sigma_{||}, \delta_{||}, x_0, X_m) \tag{2.31}$$

beschrieben werden. Aus der Vereinigungsmenge der Kompositionskomponenten resultiert das Alphabet des parallelen Generators

$$\Sigma_{||} = \Sigma_1 \cup \Sigma_2 \tag{2.32}$$

in dem auch alle privaten Ereignisse vorhanden sind. Die Mengen der Zustände und der markierten Endzustände werden wie bei der Produktkomposition mit dem kartesischen Produkt der Komponenten berechnet. Die Zustandsübergangfunktionen werden nach [Wen06] folgendermaßen definiert

$$\delta(x,\sigma) = \begin{cases} \delta_1(x_1,\sigma) \times \delta_2(x_2,\sigma), & wenn\, \delta_1(x_1,\sigma)! \wedge \delta_2(x_2,\sigma)! \\ \delta_1(x_1,\sigma) \times \{x_2\}, & wenn\, \delta_1(x_1,\sigma)! \wedge \neg\delta_2(x_2,\sigma)! \wedge \sigma \notin (\Sigma_1 \cap \Sigma_2) \\ \{x_1\} \times \delta_2(x_2,\sigma), & wenn\, \neg\delta_1(x_1,\sigma)! \wedge \delta_2(x_2,\sigma)! \wedge \sigma \notin (\Sigma_1 \cap \Sigma_2) \\ nicht\, definiert, & sonst. \end{cases} \tag{2.33}$$

Die Gleichung 2.29 zeigt, dass ein Zustandswechsel generiert wird, wenn beide Zustandsübergangsfunktionen δ_1 und $\delta_2$definiert sind. Es ist somit ein gleichzeitiges Schalten der Generatoren möglich. Ist eine der beiden Zustandsübergangsfunktionen nicht definiert, ist

es nur einem der beiden Generatoren möglich einen Zustandswechsel zu vollziehen. Im Gegenteil zur Produktkomposition findet er dennoch statt, wenn für das Ereignis $\sigma \in \Sigma_{PE}$ gilt.

Die natürliche Projektion ist die Funktion $P_\Sigma : \Sigma^* \to \Sigma_1^*$. Sie reduziert die Ereignisse aus einem Alphabet Σ die in $\Sigma_1 \subseteq \Sigma$ nicht vorhanden sein sollen. Durch die nachfolgenden Gleichungen wird die natürliche Projektion nach [Lun12] definiert:

$$P_{\Sigma_1}(\varepsilon) = \varepsilon \tag{2.34}$$

$$P_{\Sigma_1}(\sigma) = \begin{cases} \sigma, & wenn\, \sigma \in \Sigma_1 \\ \varepsilon, & sonst \end{cases} \tag{2.35}$$

$$P_{\Sigma_1}(s\sigma) = P_{\Sigma_1}(s)P_{\Sigma_1}(\sigma), s \in \Sigma^*, \sigma \in \Sigma. \tag{2.36}$$

Demgegenüber steht die inverse Projektion $P_{\Sigma_1}^{-1}$. Mit ihr werden alle Ereignisse dem Alphabet Σ hinzugefügt, die nicht in $\Sigma_1$enthalten sind und durch die natürliche Projektion entfernt werden. Die inverse Projektion wird dargestellt durch die Abbildung

$$P_{\Sigma_1}^{-1} : \Sigma_1^* \to 2^{\Sigma^*} \tag{2.37}$$

und wird durch die Gleichung

$$P_{\Sigma_1}^{-1}(t) = \{s \in \Sigma^* | P_{\Sigma_1}(s) = t\} \tag{2.38}$$

definiert.

Sind beide Ereignisalphabete exakt gleich ($\Sigma_1 = \Sigma_2$) so resultiert aus der parallelen Komposition sowie aus der Produktkomposition der gleiche Kompositionsgenerator. Dieser Fall wird nach [Won04] „Meet" genannt.

$$G_1||G_2 = G_1 \times G_2 \tag{2.39}$$

Sind in Σ_1 und in Σ_2 keine gemeinsamen Ereignisse vorhanden, so arbeiten die Generatoren uneingeschränkt asynchron. In diesem Fall gilt

$$\Sigma_1 \cap \Sigma_2 = \varnothing. \tag{2.40}$$

Dieser Fall wird nach [Won04] „Shuffle" genannt.

Erreichbarkeitsanalyse.
Diese Analyse dient dem Ermitteln bestimmter Zustände auf Ihre Erreichbarkeit von einem Anfangszustand x_0 aus. Wenn ein String $s \in \Sigma^*$ vorhanden ist, mit dem der Zustand $x \in X$ von x_0 aus erreichbar ist, so besitzt dieser Zustand das Attribut „erreichbar von x_0". Dies bedeutet, dass eine Funktion $\delta(x_0, s) = x$ definiert ist, mit der sich die Menge aller erreichbaren Zustände wie folgt beschreibt.

$$X_{ac} = \{x \in X \,|\, \exists s \in \Sigma^*(\delta(x_0, s) = x)\} \tag{2.41}$$

Alle Zustände $X - X_{ac}$ sind nicht von x_0 erreichbar und somit für die Darstellung eines Systems nicht relevant. Mit der Ac-Operation werden alle Zustände aus der Zustandsmenge X, der Menge der markierten Zustände X_m und der Zustandsübergangsfunktion auf die erreichbaren Zustände reduziert. Das Alphabet Σ wird dabei nicht verändert. Die folgenden Gleichungen definieren die Ac-Operation nach [CL99]

$$Ac(G) = (X_{ac}, \Sigma, \delta_{ac}, x_0, X_{ac,m}) \tag{2.42}$$

mit

$$X_{ac} = \{x \in X \,|\, \exists s \in \Sigma^*(\delta(x_0, s) = x)\}, \tag{2.43}$$

$$X_{ac,m} = X_m \cap X_{ac}, \tag{2.44}$$

$$\delta_{ac} = \delta\,|_{X_{ac} \times \Sigma \to X_{ac}}. \tag{2.45}$$

Zum Begriff der Erreichbarkeitsanalyse gehört das Konzept der Ko-Erreichbarkeit. Dieses Konzept sucht in einem Generator G nach Zuständen, von denen mindestens eine Ereignisfolge zu einem markierten Zustand führt. Ein Zustand wird demnach ko-erreichbar genannt, wenn ein String $s \in \Sigma^*$ vorhanden ist, so dass $\delta(x, y) \in X_m$ und durch den ein markierter Zustand erreicht wird. Die ko-erreichbaren Zustände werden als Menge folgendermaßen nach [Url12] definiert:

$$X_{CoAc} = \{x \in \Sigma \,|\, \exists s \in \Sigma^*(\delta(x, s) \in X_m)\} \tag{2.46}$$

Wie schon bei den erreichbaren Zuständen existiert für die ko-Erreichbaren Zustände eine Operation zum Entfernen der Zustände, die nicht zu den ko-erreichbaren Zuständen gehören. Die CoAc-Operation reduziert somit die Menge der Zustände auf diejenigen, die auf

einer Trajektorie vom Anfangszustand x_0 bis zu einem Endzustand $x \in X_m$ liegen. Daraus ist ersichtlich, dass diese Operation die Sprache $L(G)$ reduziert, die Menge der markierten Zustände bleibt dabei aber identisch. Nach [Url12] wird die Ko-Erreichbarkeit wie folgt definiert

$$CoAc(G) = (X_{CoAc}, \Sigma, \delta_{CoAc}, X_m)\, mit \tag{2.47}$$

$$X_{CoAc} = \{x \in X \,|\, \exists s \in \Sigma^*(\delta(x,s) \in X_m)\}, \tag{2.48}$$

$$x_{0,CoAc} = \begin{cases} x_0, & wenn\, x_0 \in X_m, \\ nicht\, definiert, & sonst \end{cases} \tag{2.49}$$

$$\delta_{CoAc} = \delta\,|_{X_{CoAc} \times \Sigma \to X_{CoAc}}. \tag{2.50}$$

Hier wird durch die Notation $\delta\,|_{X_{CoAc} \times \Sigma \to X_{CoAc}}$, δ auf die Menge der ko-erreichbaren Zustände limitiert.
Führt man beide Operationen (Ac und CoAc) der Reihe nach aus, ist dies die Trim-Operation. Diese Operation enfernt alle nicht erreichbaren sowie alle nicht ko-erreichbaren Zustände des Generators. Der aus der Trim-Operation resultierende Generator enthält nur noch die Zustände, die existenziell für das markierende Verhalten sind. Die Erreichbarkeit kann ebenso mit Hilfe der Adjazenzmatrix auf algebraischem Weg ermittelt werden.

Blockierung eines Generators (Deadlock, Livelock).
Wenn sich ein Generator in einem Zustand befindet, aus dem er keinen markierenden Zustand mehr erreichen kann, blockiert das System. So kann z.B. ein Produktionsprozess nicht vollendet werden. Wenn ein System blockiert, impliziert dies immer einen Livelock oder einen Deadlock. Ein Deadlock tritt auf, wenn in einem Generator der Zustand $x \notin X_m$ mit $\neg\delta(x,\sigma)!, \forall \in \Sigma$ aktiv ist, in dem keine Ereignisse mehr generiert werden können. Dies hat zur Folge, dass der aktive Zustand nicht mehr verlassen werden kann und es gilt nach [Wen06] die echte Inklusion.

$$\overline{L_m(G)} \subset L(G). \tag{2.51}$$

Tritt ein Livelock auf, so hat der Generator eine Zustandsmenge $\widetilde{X} \subseteq X$ mit $x \in X_m, \forall x \in \widetilde{X}$ erreicht, die er nicht mehr verlassen kann. Bei einem Livelock kann der Generator kein Ereignis mehr generieren, dass zu einem markierenden Zustand führt. Er kann lediglich

zwischen den Zuständen $x \in \widetilde{X}$ schalten. Auch bei einem Livelock gilt die echte Inklusion 2.51. Infolge dessen ist ein Generator blockierend, wenn die echte Inklusion gilt und nicht blockierend, wenn gilt

$$\overline{L_m(G)} = L(G). \tag{2.52}$$

Die in diesem Abschnitt vorgestellten Operationen sind die Grundlage für die Supervisory Control Theory und vereinfachen das Modellieren von Generatoren.

Im Anschluss wird ein Beispiel für ein logisches DES modelliert und analysiert.
Das System, welches nachfolgend modelliert wird, ist eine Fertigungszelle, die aus zwei Maschinen (M1 und M2) und einen Transportsystem (AGV) mit der Kapazität 1 besteht. Die Werkstücke werden zuerst auf M1 und anschließend auf M2 bearbeitet. Die Werkstücke werden M1 aus einem Rohlager zugeführt und aus M2 in ein Endlager befördert. Das AGV wird somit entweder aus M1 oder aus M2 mit einem Werkstück bestückt, um diese dem Endlager bzw. M2 zuzuführen. Abbildung 2.2 zeigt eine schematische Darstellung der Fertigungszelle. Der Tabelle 2.1 ist die Nomenklatur der Ereignisse zu entnehmen.

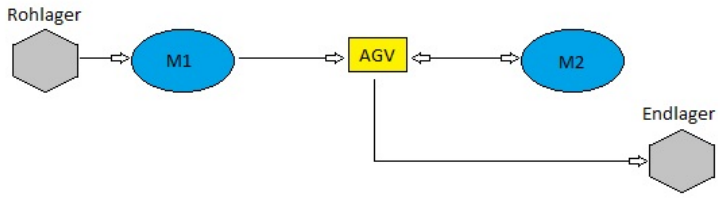


Abbildung 2.2: Fertigungszelle mit AGV

Es wird in diesem Beispiel davon ausgegangen, dass alle Komponenten fehlerfrei arbeiten. Die Maschinen M1 und M2 bestehen aus den Zuständen „idle” und „working”. Die Zustandsübergänge werden hervorgerufen, wenn eine Bearbeitung startet bzw. endet. Die Initial- bzw. markierenden Zustände sind hier jeweils „idle”, da diese signalisieren, dass eine Komponente (wieder) frei ist. Abbildung 2.3 zeigt die Komponentenmodelle der Maschinen M1 und M2 sowie der AGV. Die entsprechenden Ereignisdefinitionen sind Tabelle 2.1 zu entnehmen.

Das Gesamtmodell der Fertigungszelle wird durch das synchrone Produkt der einzelnen Komponenten gebildet.

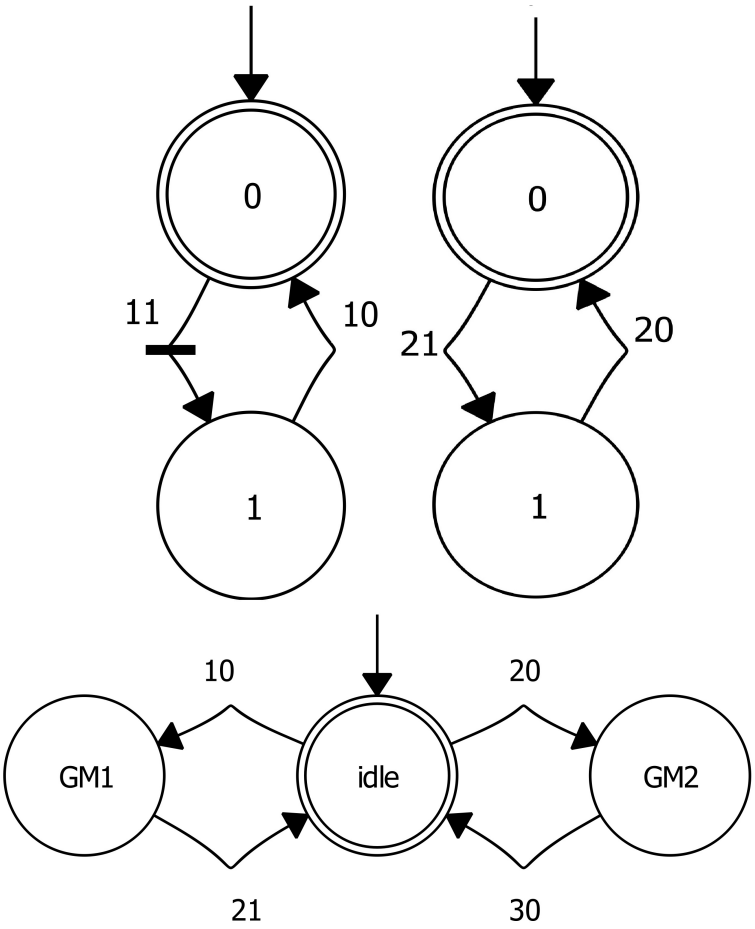


Abbildung 2.3: Komponentenmodelle der Strecke (oben: M1, M2 / unten: AGV)

$$G_{ZELLE} = G_{M1} ||_{SYPC} (G_{M2} ||_{SYPC} G_{AGV}) \tag{2.53}$$

mit der Menge der markierenden Zustände

$$X_m = \{0\} \tag{2.54}$$

und dem Initialzustand

$$x_0 = 0. \tag{2.55}$$

Abbildung 2.4 zeigt das Gesamtmodell in Form eines Generators.

Tabelle 2.1: Ereignisdefinitionen Fertigungszelle

σ	**Beschreibung**	**steuerbar**
11	Start M1 (WS aus Rohlager wird auf M1 bearbeitet	ja
21	Start M2 / Entnahme AGV (WS vom AGV wird auf M2 bearbeitet	ja
10	Ende M1 / Beladen AGV (Bearbeitung M1 beendet, WS wird auf AGV zu M2 transportiert	nein
20	Ende M2 / Beladen AGV (Bearbeitung M2 beendet, WS wird auf AGV zu Endlager transportiert	nein
30	Ablage Endlager (WS wird von AGV ins Endlager gelegt	nein

Aus dem Gesamtmodell G_{ZELLE} ist zu erkennen, dass das System blockiert, wenn die Zustandsübergangsfunktion $\delta(5,10)$ stattfindet. In dieser Situation befinden sich zwei Werkstücke in der Zelle und ein drittes wird eingebunden. Somit sind alle Komponenten belegt und die Fertigungszelle blockiert. Da G_{ZELLE} ausschließlich erreichbare Zustände aufweist, ist die Ac-Operation überflüssig, da gilt:

$$G_{ZELLE} = Ac(G_{ZELLE}) \tag{2.56}$$

Die Zustände (x=7 und x=9) sind jedoch nicht ko-erreichbar und somit ändert die CoAc-Operation das ungesteuerte Verhalten des Systems. Es entfallen die Zustände x=7 und x=9. Der aus der CoAc-Operation resultierende Generator ist in Abbildung 2.5 zu sehen.

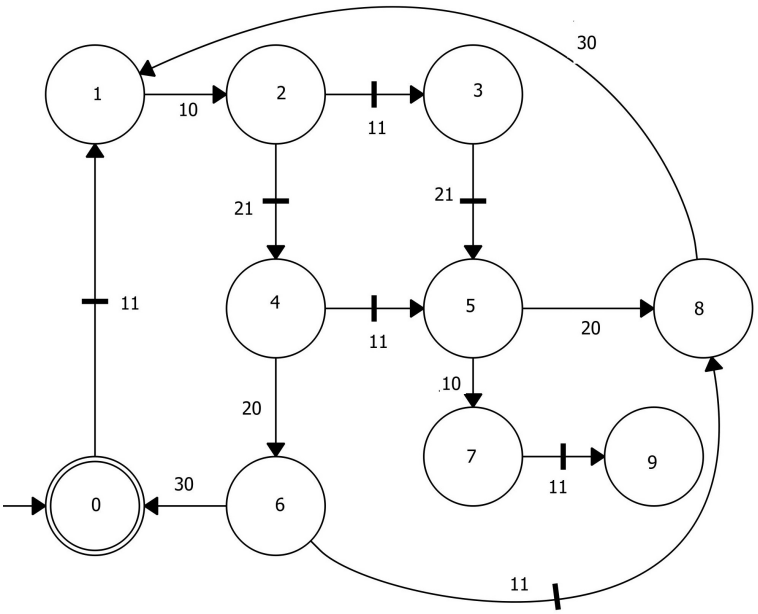


Abbildung 2.4: Gesamtmodell G_{ZELLE} der Fertigungszelle als Generator

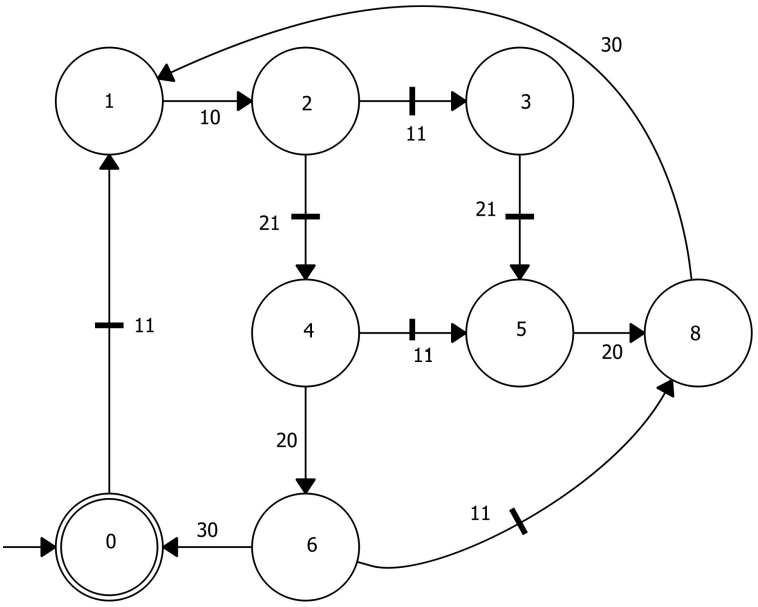


Abbildung 2.5: $CoAc(G_{ZELLE})$

2.3 Methoden der SCT

Die Supervisory Control Theory, erstmalig in der Dissertation von P.J. Ramadge erwähnt, behandelt die Thematik der formalen Steuerungssynthese für ereignisdiskrete Systeme. In einer Vielzahl von Fachartikeln wird die Problematik der Steuerungstechnik mittels SCT untersucht. Die SCT behandelt unterschiedlichste Methoden für logische DES. In dieser Arbeit werden indes nur logische DES genutzt, die wie folgt definiert sind.

> **Definition 2.9 (logische DES).** Ein ereignisdiskretes System ist ein dynamisches System mit einem diskreten Zustandsraum, dessen Zustände sich spontan durch das asynchrone Auftreten von Ereignissen über die Zeit ändern. Ein Ereignis ist eine Erscheinung oder eine Aktion ohne zeitliche Dauer. Werden ereignisdiskrete Systeme auf einen logischen Abstraktionsgrad beschränkt, so spricht man von logischen DES.

Die Dissertation von P.J. Ramadge formalisiert die entscheidende Idee der SCT und dient einer Vielzahl von Veröffentlichungen als Vorlage. Infolge dessen hat sich die SCT stetig weiter entwickelt und ist zu einem wertvollen Werkzeug zum Lösen von Problemen geworden, die durch logische DES auftreten. Dennoch findet die Supervisory Control Theory wenig Anklang in der Industrie, was auf die komplizierte Verfahrensweise bei größeren industriellen Systemen und die in der Industrie wenig bekannte Darstellung durch Generatoren zurückzuführen ist.
In diesem Teil der Arbeit wird der einfachste Ansatz der SCT erläutert. Es wird zunächst auf die Funktion des Generators als grundlegendes Element der SCT eingegangen. Darüber hinaus wird die Frage der Steuerbarkeit beantwortet, die von fundamentaler Wichtigkeit für einen Steuerungsentwurf mit Hilfe der Supervisory Control Theory ist. Abschließend ist der monolithische Steuerungsentwurf definiert.

2.3.1 Streckenmodell und Steuerkreis

Die Strecke wird mittels Modellierung als Generator G nach Definition 2.6 dargestellt. Die SCT formuliert einen Supervisor[1], der auf das markierende sowie auf das freie Verhalten der Strecke einwirken kann. Der Supervisor erlaubt und verbietet Ereignisse und nimmt so Einfluss auf die Strecke. Zunächst wird das Alphabet Σ der Strecke in zwei Teilalphabete sortiert, in steuerbare Ereignisse Σ_c und nicht steuerbare Ereignisse Σ_{uc} weil ein Supervisor nicht auf alle Ereignisse $\sigma \in \Sigma$ einwirken kann. Zwei Beispiele für nicht steuerbare Ereignisse sind ein Sensorsignal bzw. eine Fehlermeldung. Bezüglich dieser Aufteilung wird das Alphabet wie folgt definiert:

[1] Der Begriff Supervisor wird sowohl für Singular als auch für Plural verwendet

$$\Sigma = \Sigma_c \cup \Sigma_{uc} mit \Sigma_c \cap \Sigma_{uc} = \varnothing \tag{2.57}$$

Des Weiteren kann jedes Ereignis $\sigma \in \Sigma$ auch beobachtbar Σ_o bzw. nicht beobachtbar Σ_{uo} sein. Dieses Attribut ist in dieser Arbeit jedoch nicht relevant.
In der grafischen Darstellung von Generatoren werden Transitionen mit der Eigenschaft „steuerbar" mit einem Querstrich gekennzeichnet.
Ein Generator G generiert Ereignisfolgen s. Diese Ereignisfolgen werden von der Steuerung beobachtet, welche dann unpassende Ereignisse für den nächsten Schritt unterbindet. In $S(s)$ werden die Folgeereignisse beschrieben, die im nächsten Schritt bewilligt sind. Alle Ereignisse $\sigma \in \Sigma_{uc}$ werden bei jedem Steuereingriff erlaubt, da der Supervisor keinen Einfluss auf sie nehmen kann. Die von Generator G dargestellte Strecke wird durch Ereignisfolgenrückführung gesteuert. In Abbildung 2.6 ist ein Blockschaltbild einer solchen Ereignisfolgenrückführung gezeigt.

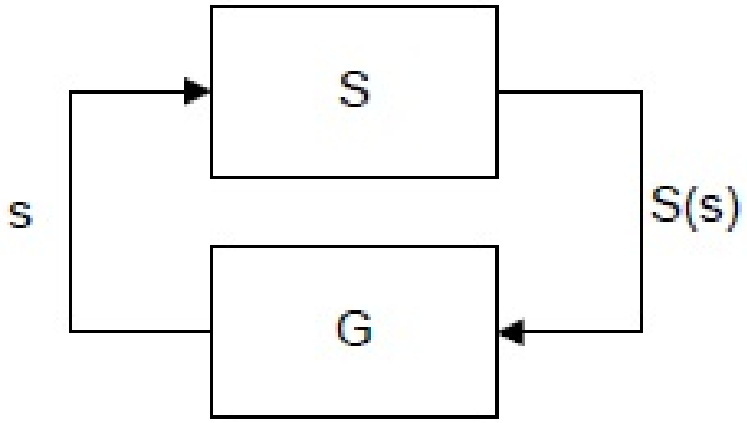


Abbildung 2.6: Steuerkreis S/G

Das gesteuerte Verhalten wird $L(S/G)$ genannt und ist so definiert, dass ε immer in $L(S/G)$ beinhaltet ist. Wenn eine Konkatenation von s und σ in S erlaubt und in G definiert ist, ist auch ein auf s folgendes Ereignis σ möglich. Nur Strings die diese Eigenschaft aufweisen, sind durch die Steuerung zugelassen.
[Won04] definiert das gesteuerte Verhalten $L(S/G) \subseteq L(G)$ von S/G, mit $s \in \Sigma^*$ und $\sigma \in \Sigma$ wie folgt:

1. $\varepsilon \in L(S/G)$,

2. $(s \in L(S/G) \wedge \sigma \in S(s) \wedge s\sigma \in L(G))$ daraus folgt $s\sigma \in L(S/G)$,

3. keine anderen Strings gehören zu $L(S/G)$.

Im Steuerkreis ergibt sich das markierende Verhalten folgendermaßen

$$L_m(S/G) = L(S/G) \cap L_m(G) \tag{2.58}$$

und wird als nichtblockierend bezeichnet wenn gilt

$$\overline{L_m(S/G)} = L(S/G). \tag{2.59}$$

2.3.2 Steuerbarkeit

Die Eigenschaft der Steuerbarkeit ist eine existenzielle Bedingung für eine Steuerung und ist folgendermaßen nach [RW87] definiert:

> **Definition 2.10 (Steuerbarkeit).** $G = (X, \Sigma, \delta, x_0, X_m)$ sei ein Generator und $L(G)$ das ungesteuerte Verhalten von G, $\Sigma = \Sigma_c \cup \Sigma_{uc}$ partitioniert in einen steuerbaren und einen nicht steuerbaren Teil. Eine Sprache $K \subseteq \Sigma^*$ heißt steuerbar bezüglich G, genau dann wenn
>
> $$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}. \tag{2.60}$$

Definition 2.10 besagt, dass kein String $s \in \overline{K}$, $\overline{K}$ verlassen darf, wenn ein Ereignis $\sigma \in \Sigma_{uc}$, welches in $L(G)$ existiert, ausgelöst wird.

Der Begriff der Steuerbarkeit wird nachfolgend anhand des schon eingeführten Beispiels der Fertigungszelle deutlich gemacht.
Eine informelle Spezifikation K1 an die gesteuerte Fertigungszelle ist, dass diese nicht blockieren darf. Um dies zu verhindern, müsste in Zustand x=5 das Ereignis $\sigma = 10$ unterbunden werden. Das Ereignis $\sigma = 10$ ist jedoch nicht steuerbar und kann somit nicht verboten werden. Die Spezifikation (K1) ist demnach nicht steuerbar bezüglich G_{ZELLE}. Eine weitere Steuerung K2 könnte das steuerbare Ereignis $\sigma = 11$ in den Zuständen x=6,

x=4 und x=2 verhindern und damit die Anzahl der Werkstücke im Produktionsprozess auf 1 beschränken, was nicht optimal ist. Die Spezifikation K3 jedoch wäre steuerbar bezüglich G_{ZELLE} und beschränkt die Anzahl der Werkstücke, die sich im Produktionsprozess aufhalten, auf nur 2 Werkstücke. Da K1 nicht steuerbar bezüglich G_{ZELLE} ist, kann auch keine Steuerung entwickelt werden, die K1 erfüllt. Um eine minimal restriktive Steuerung für die Fertigungszelle zu bekommen, berechnet man alle steuerbaren Teilsprachen von K1 bezüglich G_{ZELLE} (Klasse der steuerbaren Teilsprachen).

$$C_{in}(K1, L(G_{ZELLE})) = \{\varnothing, ..., \{\varepsilon\}, \{1110\}, ..., K2, K3, ...\} \tag{2.61}$$

Nach Vereinigung aller Mengenelemente dieser Klasse bekommt man automatisch die größtmögliche (supremale) steuerbare Teilsprache von K1, welche wie folgt definiert wird:

$$K1^{\uparrow C} = K3 \tag{2.62}$$

Das Nichtblockieren der gesteuerten Fertigungszelle wird mit folgender Bedingung geprüft.

$$K3 = \overline{K3} \cap L_m(G_{ZELLE)}) \tag{2.63}$$

Wenn diese Bedingung erfüllt ist, so ist auch K3 L_m-abgeschlossen und ein Nichtblockieren ist gewährleistet.
Die Spezifikation K3 2.7 ist demnach minimal restriktiv und nichtblockierend bezüglich der Fertigungszelle.

2.3.3 Monolithischer Entwurf

Der monolithische Ansatz zur Modellierung einer Steuerung ist die einfachste Art der Entwicklung eines Supervisors. Hierbei wird mit einem unstrukturierten Gesamtmodell der Strecke gearbeitet. Für dieses Gesamtmodell wird ein einzelner Supervisor modelliert, der die Steuerung der Strecke realisiert.
In dieser Arbeit wird das Basic Supervisory Control Problem (BSCP) bzw. BSCP-Nichtblockierend (-NB) betrachtet. Hierbei sind nur Strecken relevant, die nicht steuerbare Ereignisse aufweisen, jedoch keine nicht beobachtbaren Ereignisse. Das BSCP-Problem ist wie folgt definiert:

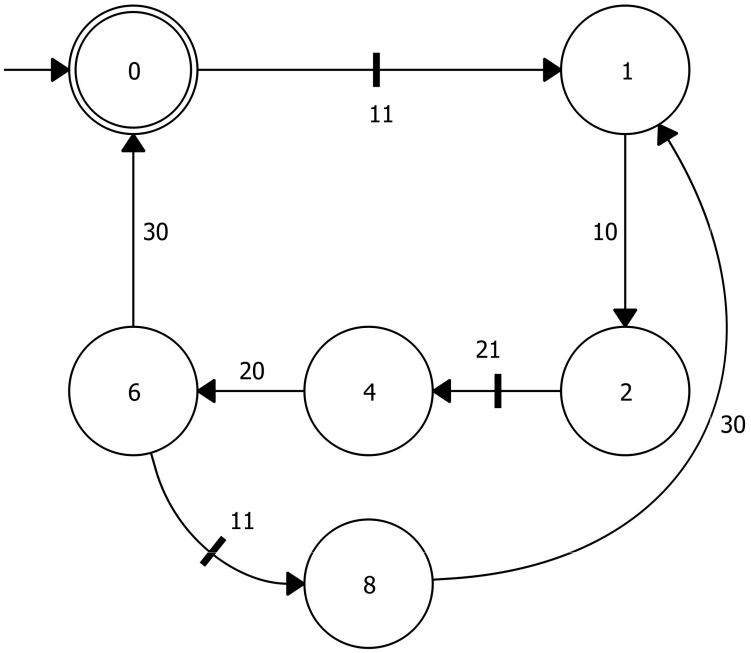


Abbildung 2.7: Spezifikation K3 der Fertigungszelle

BNCP-NB.
Gegeben ist ein Generator G mit einem Ereignisalphabet Σ, $\Sigma_{uc} \subseteq \Sigma$ und eine $L_m(G)$-abgeschlossene Spezifikation $K \subseteq L_m(G)$. Bestimme eine nichtblockierende Steuerung, so dass gilt:
1. $L_m(S/G) \subseteq K$
2. $L_m(S/G)$ „maximal" ist, also für jede andere nichtblockierende Steuerung S' mit $L_m(S'/G) \subseteq K$ gilt:

$$L_m(S'/G) \subseteq L_m(S/G). \tag{2.64}$$

Die gewünschte Steuerung ist dann folgendermaßen zu wählen:

$$L(S/G) = \overline{K^{\uparrow C}}. \tag{2.65}$$

Es gibt zwei Möglichkeiten der Darstellung für eine Steuerung. Die erste ist die Darstellung als Liste. Hierbei wird S durch die Liste seiner Steuereingriffe definiert und $S(s)$ muss

für jeden String $s \in L(G)$ festgelegt sein. Steuereingriffe sind als Menge aller im nächsten Schritt legitimen Ereignisse definiert. Wenn $L(S/G) = \overline{K^{\uparrow c}}$, dann führen folgende Steuereingriffe zur gewünschten Sprache:

$$S(s) = [\Sigma_{uc} \cap \{\sigma \in \Sigma | \delta(\delta(x_0, s)\sigma)\}] \cup \left\{\sigma \in \Sigma_c | s\sigma \in \overline{K^{\uparrow c}}\right\} \tag{2.66}$$

Die zweite Darstellungsform stellt die Steuerung S als Akzeptor R dar, dessen markierende Sprache $\overline{K^{\uparrow c}}$ ist. R wird wie folgt entwickelt:

$$R = \{Y, \Sigma, \zeta, y_0, Y_m\} \, mit \tag{2.67}$$

$$Y = Y_m, \Sigma \, wie \, Strecke, \tag{2.68}$$

$$G = Trim(G) \, und \, \zeta, \, sodass \, L_m(G) = L(G) := \overline{K^{\uparrow c}} = K^{\uparrow c}. \tag{2.69}$$

Nach Konstruktion des Akzeptors R, mit Hilfe der Produktkomposition, gilt für den geschlossenen Steuerkreis:

$$L(G||_{SPC}R) = L(G) \cap L(R) = L(G) \cap \overline{K^{\uparrow c}} = L(S/G) \tag{2.70}$$

$$L_m(G||_{SPC}R) = L_m(G) \cap L_m(R) = \overline{K^{\uparrow c}} \cap L_m(G) = L(S/G) \cap L_m(G) = L_m(S/G). \tag{2.71}$$

R wird hier als Standardrepräsentation von S bezeichnet und ist fehlerfrei bezüglich seiner Steuereingriffe. Bezüglich der Anzahl der Elemente aus Y ist jedoch nicht zwingend ein Minimum erreicht und weitere Reduzierungen sind eventuell möglich.
Angesichts der großen Anlagen der heutigen Zeit, welche eine hohe Modellkomplexität und somit auch eine große algorithmische Komplexität mit sich bringen, ist der monolithische Entwurf nicht zweckmäßig.

Am Beispiel der Fertigungszelle mit AGV wird anschließend eine Steuerung als Generator, und der dazugehörige Steuerkreis entwickelt. In Abbildung 2.8 ist die Steuerung der

Fertigungszelle als Generator abgebildet, der $\overline{K3^{\uparrow C}} = K3$ markiert.

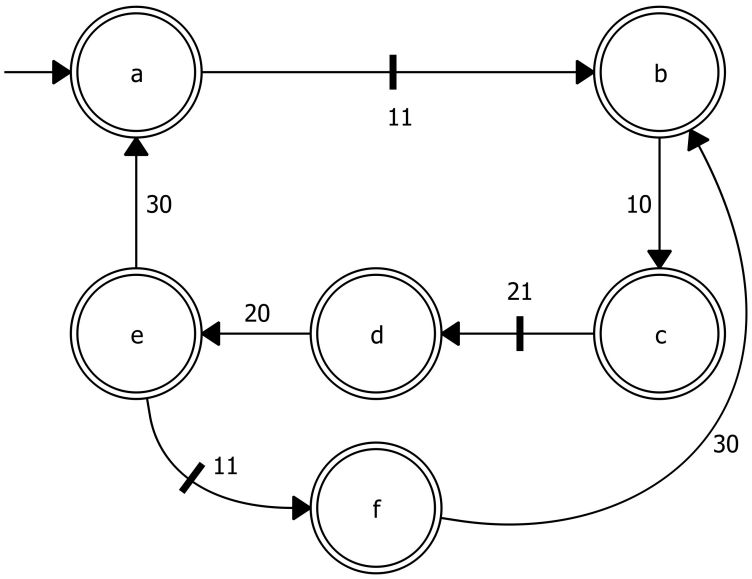


Abbildung 2.8: Steuerung der Fertigungszelle als Generator

Der Steuerkreis für die Fertigungszelle ergibt sich aus der Produktkomposition der Steuerung S und dem Generator G_{ZELLE}

$$S/G_{ZELLE} = S||_{SPC}G_{ZELLE} \tag{2.72}$$

und ist in Abbildung 2.9 als Generator dargestellt.

Es ist deutlich zu erkennen, dass der Steuerkreis nicht blockiert und somit die gewünschte Spezifikation, die an das gesteuerte System gestellt wurde, erfüllt ist. Das Verhalten sowie das markierende Verhalten des gesteuerten Systems sind den nachfolgenden Gleichungen zu entnehmen:

$$L(S/G_{ZELLE}) = L(S) \cap L(G_{ZELLE}) = \overline{K3} \cap L(G_{ZELLE}) = \overline{K3} \tag{2.73}$$

$$L_m(S/G_{ZELLE}) = L_m(S) \cap L_m(G_{ZELLE}) = \overline{K3} \cap L_m(G_{ZELLE}) = K3 \tag{2.74}$$

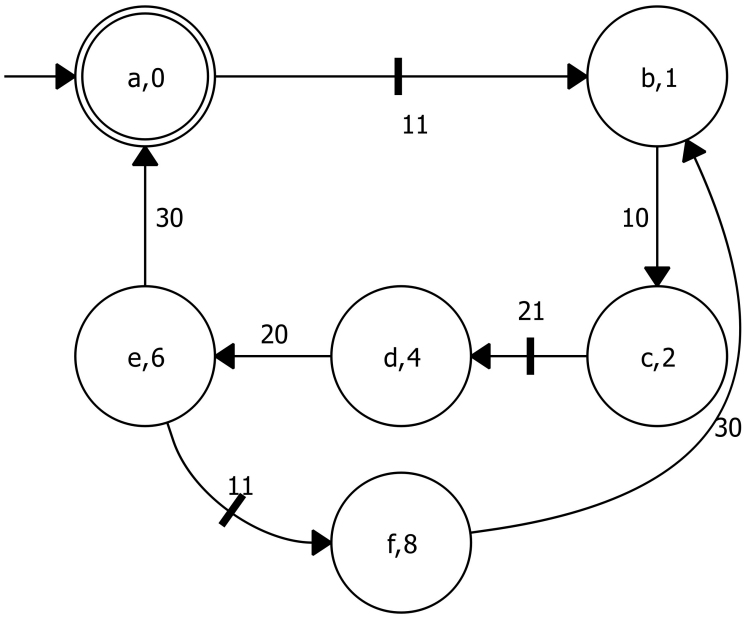


Abbildung 2.9: Steuerkreis der Fertigungszelle

2.4 Strukturierter Entwurf

2.4.1 Modularer Ansatz

Der modulare Ansatz hat das Ziel, die Modellkomplexität im Vergleich zum monolithischen Ansatz zu verringern. Dabei ist der Hauptgedanke die Steuerungsaufgabe auf mehrere Supervisor zu verteilen, und sie zu einem endgültigen Supervisor S_{mod} zusammenzufügen. Die Anwendung des Ansatzes setzt voraus, dass alle Ereignisse $\sigma \in \Sigma$ für jeden Supervisor global vorliegen. Die Architektur wird in Abbildung 2.10 schematisch aufgezeigt.

Die generierten Strings s des Generators G werden von allen Supervisor kontrolliert. Jeder Supervisor unterbindet bzw. erlaubt Ereignisse nach seiner eigenen Steuerungsaufgabe durch die Steuereingriffe S_1und S_2. Das logische „und" in der Abbildung zeigt, dass nur dann ein Folgeereignis $\sigma \in S_{mod}(s)$ aktiv wird, wenn es von keinem der beiden Supervisor unterbunden wird $(\sigma \in S_1(s) \wedge \sigma \in S_2(s))$. Formal ist der modulare Supervisor nach [Wen06] folgendermaßen definiert.

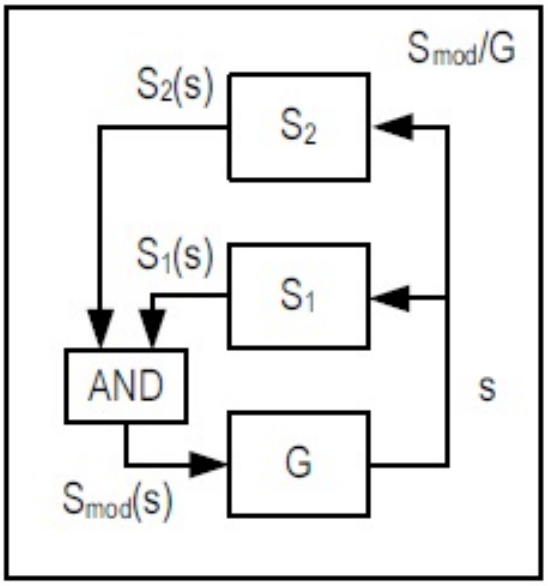


Abbildung 2.10: Architektur des modularen Ansatzes

Definition 2.11 (Modularer Supervisor). S_1, S_2,..., S_n seien zulässige Supervisor bezüglich G und Σ das Ereignisalphabet von G. Der modulare Supervisor $S_{mod} : L(G) \rightarrow 2^{\Sigma}$ ist definiert als

$$S_{mod}(s) = S_1(s) \cap S_2(s) \cap ... \cap S_n(s). \tag{2.75}$$

Informell ist in Definition 2.11 beschrieben, dass die Schnittmenge aller Steuereingriffe der Komponenten-Supervisor die Steuereingriffe von S_{mod} formt. Daraus gehen die Sprachen des Steuerkreises hervor:

$$L(S_{mod}/G) = L(S_1/G) \cap L(S_2/G) \cap ... \cap L(S_n/G) \tag{2.76}$$

$$L_m(S_{mod}/G) = L_m(S_1/G) \cap L_m(S_2/G) \cap ... \cap L_m(S_n/G). \tag{2.77}$$

Die Anforderung an einen mit modularem Ansatz entwickelten Supervisor ist, dass dieser das gleiche Leistungsvolumen hat wie der monolithische Supervisor. Damit dies der Fall

ist muss die Gesamtspezifikation nicht nur das maximal zulässige Verhalten beschreiben, sondern auch in präfix-abgeschlossenen Teilspezifikationen vorhanden sein, bzw. sich in eine solche Form partitionieren lassen. Zur Einhaltung der Steuerbarkeit unter Schnittmengenbildung ist dies eine notwendige Bedingung, denn sind zwei Sprachen L_1und L_2 präfix-abgeschlossen, so gilt nach [WR88]:

$$(L_1 \cap L_2)^{\uparrow C} = L_1^{\uparrow C} \cap L_2^{\uparrow C}. \tag{2.78}$$

Aufgrund der eben erwähnten Bedingung der Präfix-Abgeschlossenheit ist es möglich, die einzelnen Supervisor unabhängig voneinander zu entwickeln. Dadurch gilt für $i = 1, ..., n$:

$$L(S_i/G) = L_{i,a}^{\uparrow C}. \tag{2.79}$$

Zur Umsetzung der Spezifikation L_a kann jetzt ein Supervisor nach der oben genannten Definition 2.11 entworfen werden. Der modulare Supervisor sichert jedoch kein Nichtblockieren betreffend G. Um ein Nichtblockieren des modularen Supervisor zu gewährleisten, muss folgende Bedingung erfüllt sein:

$$\overline{L_m(S_1/G) \cap L_m(S_2/G)} = L_m(S_1/G) \cap L_m(S_2/G). \tag{2.80}$$

Informell sagt Gleichung 2.80 aus, dass ein modularer Steuerkreis nichtblockierend ist, wenn die markierenden Verhaltensweisen der einzelnen nichtblockierenden Teilsteuerkreise konfliktfrei arbeiten.
Die Vorteile des modularen Ansatzes gegenüber dem monolithischen Ansatz sind, dass die resultierenden Supervisor generell kleinere Zustandsräume besitzen. Durch die Modularisierung ist die Steuerung anpassungsfähiger.
Dennoch ist bei der Modellierung mit Hilfe des modularen Ansatzes ein unstrukturiertes Gesamtmodell G nötig. Das Streckenmodell offenbart somit die gleichen Schwachstellen wie das eines monolithischen Ansatzes.

2.4.2 Weitere strukturierte Ansätze

Lokal-modularer Ansatz.
Durch eine Erweiterung des modularen Ansatzes wurde der lokal-modulare Ansatz entwickelt. Dieser Ansatz legt das Augenmerk auf eine Reduzierung der Komplexität der Synthese des Supervisors. Es ist das Ziel nur diejenigen Streckenkomponenten zu vereinen, die erforderlich sind, um die Spezifikation zu gewährleisten. Jeder Supervisor ist

einem bestimmten Teil des Gesamtsystems zugeordnet und übernimmt nur Steuerungsaufgaben für diesen Teil. Als Erstes wird die Strecke G mit dem Ereignisalphabet Σ aus allen Teilmodellen mittels Komposition gebildet. Das daraus folgende System wird in asynchrone Komponenten unterteilt. Bild 2.11 zeigt die Architektur des lokal-modularen Ansatzes. Die einzelnen Komponenten des zerteilten Systems, auch feinstes Produktsystem genannt, werden mit Hilfe der parallelen Komposition zu lokalen Systemen geformt. Zum Schluss werden die Spezifikationen der lokalen Systeme des eingeschränkten und des Gesamtsystems formuliert. Eine detailliertere Beschreibung des lokal-modularen Ansatzes findet sich in [Wen06]. In den Arbeiten von Queiroz und Cury [QC00, QC02] sind Gleichungen, Definitionen und eine Anpassung der Spezifikation mit den resultierenden Supervisor nachzulesen.

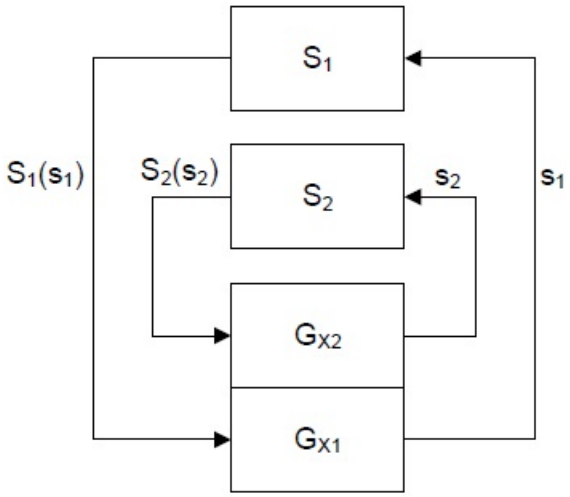


Abbildung 2.11: Architektur des lokal-modularen Ansatzes

Der lokal-modulare Ansatz eignet sich speziell für Systeme mit vielen Nebenläufigkeiten.

Hierarchisch Interface-basierter Ansatz (HISC).
Der Hierarchisch Interface-basierte Ansatz wurde federführend von R.J. Leduc entwickelt [Led05a, Led05b, Led06]. Die Idee besteht in der Kombination aus hierarchischer und modularer Architektur. Abbildung 2.12 zeigt die schematische Darstellung der Struktur des HISC.

Die Modularisierung und Hierarchisierung werden sowohl bei der Modellierung des Systemmodells als auch bei Steuerung angewendet. Das System wird während des Verfahrens in n Komponenten aufgeteilt, die auf zwei Hierarchieebenen modelliert und gesteuert werden. Die Kommunikation zwischen den Ebenen wird mit „Interfaces” realisiert. Die obere Hier-

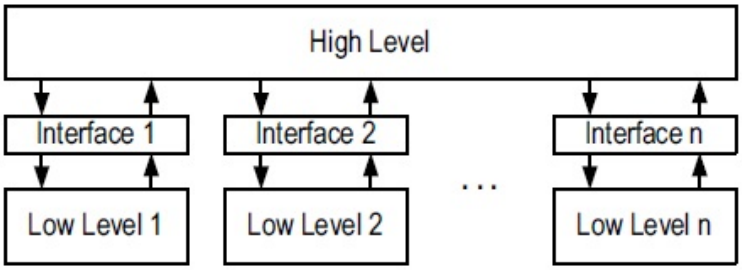


Abbildung 2.12: schematische Darstellung der Struktur des HISC

archieebene, auch „High-Level" genannt, besteht aus einem allgemeinen Streckenmodell, einem dafür modellierten Supervisor und aus n „Interfaces". Das „Low-Level", die untere der beiden Ebenen, besteht aus n Komponenten, die detailliert beschrieben und jeweils einem lokalen Supervisor zugewiesen sind. Des Weiteren besitzt jede Komponente ein an die Anforderungen der Komponente angepasstes „Interface".
Alle Systemmodelle, Supervisor und „Interfaces" können mittels Generatoren modelliert werden. Der Vorteil dieser Art der Modellierung ist, dass keine Modellierung eines Gesamtmodells nötig ist. Steuerbarkeit und Nichtblockieren des Systems werden durch die Prüfung der einzelnen „Low-Level" Komponenten mit dem jeweiligen „Interface" bzw. dem „High-Level" gewährleistet. Dieser Ansatz ist sehr zweckmäßig bei der Modellierung großer Systeme. R.J. Leduc unterscheidet in seinen Arbeiten [Led05a, Led05b] zwischen dem „seriellen Fall" $(n = 1)$ und dem „parallelen Fall" $(n \geq 1)$. Eine detailiertere Beschreibung des HISC ist auch in [Wen06] zu finden.

Kapitel 3

Codegenerierung

3.1 Allgemein

Generatorenmodelle und die dazugehörigen Supervisor, wie sie mit DESTool erstellt werden können, sind eine einfache Form einen industriellen Prozess zu modellieren und eine formale Steuerung zu entwickeln. Die manuelle Programmierung ist im allgemeinen sehr fehleranfällig und stellt bei realen Anlagen aufgrund der steigenden Größe eine schwierige und mühsame Aufgabe dar. Infolge dessen ist eine automatische Codegenerierung sinnvoll, um die Fehlerquellen zu minimieren und die Arbeitszeit effektiver zu nutzen. In der Arbeit [Url12] wurde ein Programm entwickelt, mit dem aus einer in DESTool entworfene Steuerung ein AWL-Code generiert wird. Der Ansatz, der für die Generierung genutzt wird, ist in [UGD09] nachzulesen. Es existieren jedoch viele weitere Publikationen, die sich mit dem Thema der Umwandlung von Automaten in Programmiersprachen einer SPS (nach IEC DIN EN 61131-3) befassen. In diesem Kapitel werden zwei Varianten solcher Ansätze vorgestellt.

3.2 Unterschiedliche Methoden

3.2.1 Ansatz von Liu und Darabi

In der Veröffentlichung [JD02] wird ein Ansatz zur Konvertierung eines Supervisors, modelliert nach der „Ramadge Wonham Methode (RW)", in Kontaktplan vorgestellt. In diesem Paper wird der IDEF3 („Integrated Definition") Standard genutzt, eine Methode zum Erfassen und Modellieren von Prozessen, um die Ereignisse des Supervisor zu klassifizieren.
Die Ereignisse werden in „activities" und „ressources" zerlegt. Eine „activity" ist jede Operation in einem System, welche eine bestimmte Zeit braucht, um fertiggestellt zu werden. Während dieses Zeitraums wird mindestens eine „ressource" verwendet, um die Aktivität durchzuführen. Eine „ressource" ist eine Einheit des Systems, die von mindestens einer Ak-

tivität angefordert wird. „ressourcen" werden mit „ressource-activity latch/unlatch events" für die jeweilige „ressource" reserviert bzw. freigestellt. Des Weiteren wird die Ereignismenge Σ in drei disjunkte Mengen unterteilt. Die „activity starting events" Σ_s, die „activity ending events" Σ_f und die „System events" Σ_{sm}. Das „Starting event" $\sigma \in \Sigma_s$ einer Aktivität wird sofort ausgeführt, nachdem für die Aktivität alle „ressources" reserviert wurden die es für die Durchführung benötigt. Das Startereignis σ_s einer „activity" kann somit auch als die letzte Zuordnung einer „ressource" zu einer Aktivität bezeichnet werden. Ein „Ending event" $\sigma \in \Sigma_f$ tritt auf, wenn eine „activity" beendet wird. Wird jedoch eine „ressource" während der Ausführung der Aktivität derselbigen nicht mehr zur Verfügung gestellt, wird das „Ending event" $\sigma \in \Sigma_f$ gesperrt. Ein Systemereignis ist jeder Zustandswechsel der in dem System ausgelöst wird, und der nicht als „activity starting/ending event" oder „ressource latch/unlatch event" deklariert ist.
Aus den klassifizierten Ereignissen werden zwei Vektoren gebildet. Der „state-ressource" Vektor und der „activity-ressource" Vektor. Der „activity-ressource" Vektor ist wie folgt definiert:

$$AR_a(r) = \begin{cases} 1, & wenn\, die\, Aktivität\ a\ die\, Ressource\ r\ benötigt \\ 0, & sonst. \end{cases} \tag{3.1}$$

Der „state-ressource Vektor" wird nach folgender Vorschrift gebildet, wenn gilt $\sigma = s_a \in \Sigma_s, dann \forall r \in R$

$$SR_x(r) = \begin{cases} 1, & wenn AR_a(r) = 1 \\ SR_y(r), & sonst \end{cases} \tag{3.2}$$

wenn jedoch gilt $\sigma = f_a \in \Sigma_f, dann \forall r \in R$, dann ist

$$SR_x(r) = \begin{cases} 0, & wenn AR_a(r) = 1 \\ SR_y(r), & sonst \end{cases} \tag{3.3}$$

und für die Bedingung $\sigma \in \Sigma_{sm}, dann \forall r \in R$ gilt

$$SR_x(r) = SR_y(r). \tag{3.4}$$

Außerdem wird für jedes Zustandspaar $x_1, x_2 \in X$ für das $x_2 = \delta(f_a, x_1)$ gilt und $f_a \in \Sigma_f$ ein „Ending event" der Aktivität $a \in A$ ist, jedoch kein „Starting event" für a in Σ_s enthalten ist, ein neues Ereignis s_a generiert und zur Menge Σ_s hinzugefügt. Entsprechend dazu wird ein neuer Zustand y der Menge X angefügt und die folgenden Transitionen definiert:

$$y = \delta(s_a, x_1) \tag{3.5}$$

und

$$x_2 = \delta(f_a, y). \tag{3.6}$$

Der „state-ressource" Vektor SR_x wird anschließend an jeden Zustand $x \in X$ des Supervisor angehängt und bildet den sogenannten „extended" Supervisor. Dieser erweiterte Generator wird anschließend nach nachfolgenden Regeln in die Programmiersprache Kontaktplan konvertiert.
Für jeden Zustand $x \in X - x_0$ wird für alle Ereignisse σ des Zustands x ein Netzwerk mit den nachfolgenden Eingängen und Ausgängen konstruiert. Eine Nomenklatur für die Gleichungen 3.7, 3.8, 3.9 ist der Tabelle 3.1 zu entnehmen.

$$I = \begin{cases} \{es_\sigma, x\} & wenn\,(\sigma \in \Sigma_s) \wedge (N > 1) \wedge (N_s \geq 1) \\ \{x\} & wenn\,(\sigma \in \Sigma_s) \wedge (N = 1) \\ \{x, \sigma\} & sonst \end{cases} \tag{3.7}$$

$$O^L = \{\{r | SR_y(r) = 1 \wedge SR_x(r) = 0\}, y | y = \delta(\sigma, x)\} \tag{3.8}$$

$$O^U = \{\{r | SR_y(r) = 0 \wedge SR_x(r) = 1\}, x\} \tag{3.9}$$

Tabelle 3.1: Nomenklatur der Eingangs- und Ausgangsformel

Bezeichnung	**Erklärung**
N	die Anzahl der von Zustand x ausgehenden Ereignisse
I	Eingang der als Schließer fungiert
O^L	Ausgangsstrang der etwas verriegelt
O^U	Ausgangsstrang der etwas entriegelt
N_s	die Anzahl der „activity starting events" in Zustand x
es_σ	ein Konfliktlösungssignal, welches aus allen „activity starting events" die in x verfügbar sind ein Ereignis σ auswählt

Beispiel für die Anwendung des Ansatzes.
Um das Vorgehen für diesen Ansatz deutlicher zu erläutern, wird nachfolgend ein RW Supervisor mittels der oben genannten Regeln in die Programmiersprache Kontaktplan

konvertiert. Abbildung 3.1 zeigt das Schema für den modellierten Produktionsprozess. Die „ressources" und „activities" können der Tabelle 3.2 entnommen werden, die Ereignisse und deren Beschreibung aus der Tabelle 3.3.

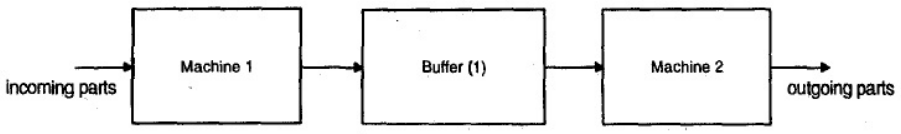


Abbildung 3.1: Schema der Beispielanlage

Der Produktionsprozess, der für dieses Beispiel verwendet wird, besteht aus zwei Maschinen (M1 und M2) die mit einem Puffer der Kapazität „1" verbunden sind. Die Maschinen können die Zustände „bereit", „besetzt" oder „defekt" einnehmen, der Puffer die Zustände „voll" oder „leer". Der Initialzustand ist für den Puffer „leer" und für die Maschinen M1 und M2 „bereit". Die Anlage hat folgende Spezifikationen für Produktion und Reparatur:

- der Puffer darf nicht über- bzw. unterlaufen
- M1 darf keine Bearbeitung starten solange ein Werkstück im Puffer vorhanden ist
- M2 darf keine Bearbeitung starten solange ein Werkstück im Puffer vorhanden ist
- M2 hat Reparaturvorrang gegenüber M1, das heißt, wenn beide Maschinen M1 und M2 defekt sind, muss zuerst M2 repariert und in den Produktionsprozess eingegliedert werden

Tabelle 3.2 beschreibt die Ressourcen und Aktivitäten für den Produktionsprozess so-

Tabelle 3.2: „ressources" und „activities" der Beispielanlage

„ressources"	Maschine 1 (im Supervisor als „1" bezeichnet)
	Maschine 2 (im Supervisor als „2" bezeichnet)
	Reparateur (im Supervisor als „3" bezeichnet)
„activities"	Betrieb der Maschine 1 (1)
	Betrieb der Maschine 2 (2)
	Reparaturarbeiten an Maschine 1 (3)
	Reparaturarbeiten an Maschine 2 (4)

wie deren Abkürzungen, die im Supervisor genutzt werden. Außer den Maschinen M1 und M2 gibt es eine dritte Ressource, die im Artikel „Repairman" genannt wird. Die Ereignisse

der Anlage sind in Ereignisse, die eine Ressource zuweisen, Ereignisse, die eine Ressource entziehen, Ereignisse, die den Start einer Aktivität darstellen, Ereignisse, die das Ende einer Aktivität signalisieren und Systemereignisse unterteilt. In Tabelle 3.3 werden diese Ereignisse aufgelistet, beschrieben und einer Abkürzung zugeordnet.

Die zwei zuvor beschriebenen Spezifikationen sind für die Modellierung der Anlage wichtig. Aus den Gleichungen 3.1,3.2,3.3,3.4 folgen der „activity-ressource" 3.10 Vektor und der „state-ressource" Vektor.

$$\begin{bmatrix} AR_1 \\ AR_2 \\ AR_3 \\ AR_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Für den „state-ressource" Vektor ergeben sich für Zustand „1" und Zustand „2" folgende Vektoren:

$$SR_1 = [000] \tag{3.11}$$

$$SR_2 = [100]. \tag{3.12}$$

In Abbildung 3.2 ist der „extended" Supervisor dargestellt, bei dem die „state-ressource" Vektoren an die entsprechenden Zustände geheftet werden. Aus dem „extended" Supervisor wird anschließend der Kontaktplan erstellt, der in Abbildung 3.2 zu sehen ist. Der Kontaktplan, der den „extended" Supervisor repräsentiert, ist so aufgeteilt, dass für jeden Zustandswechsel ein Netzwerk erstellt wird. Dies bedeutet zum Beispiel, wie in Netzwerk 3, dass wenn der Zustandsmerker von Zustand „2" aktiv ist und das „Ending-event f_1" auftritt, ein Zustandswechsel zu Zustand „3" durchgeführt wird. Infolge dessen wird Zustandsmerker „3" gesetzt und der des vorigen Zustands rückgesetzt. Durch den „state-ressource" Vektor ist klar zu erkennen, welche Ausgänge in dem aktuellen Zustand aktiv sein sollen. Dementsprechend werden in dem Netzwerk die gewünschten Ausgänge aktiviert bzw. deaktiviert.

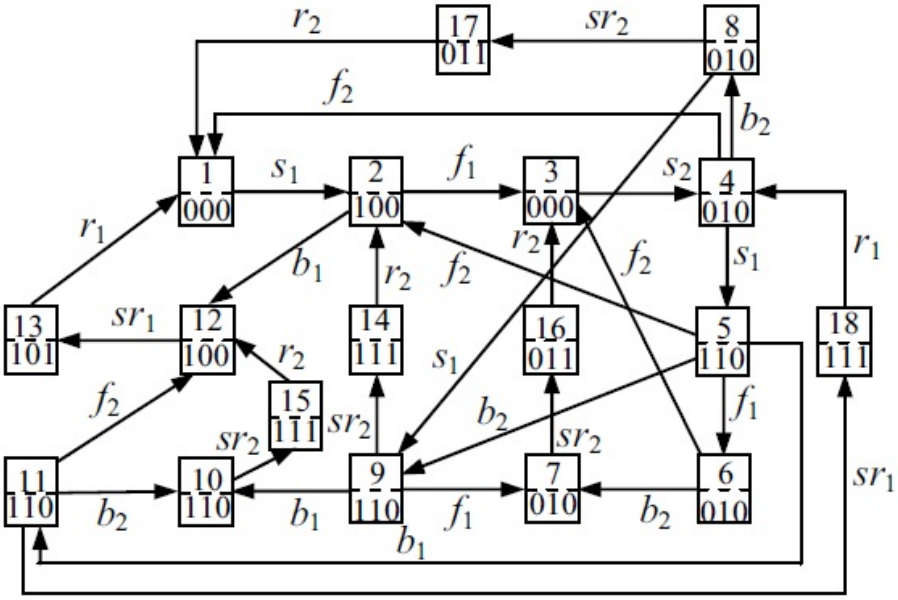


Abbildung 3.2: „extended" Supervisor

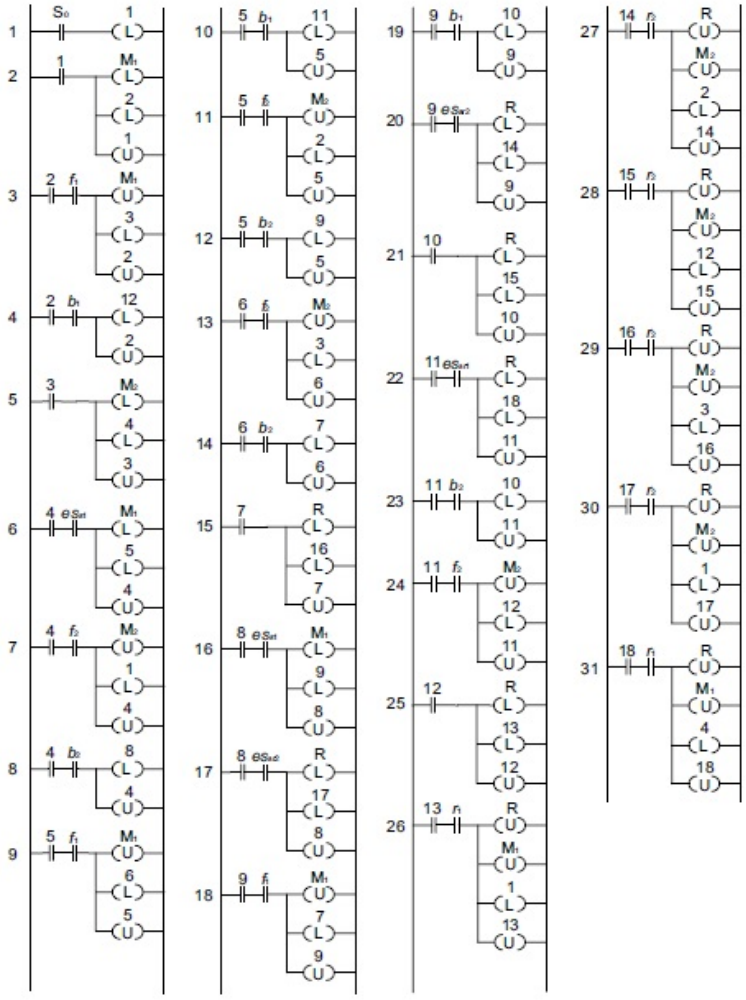


Abbildung 3.3: Kontaktplan für die Beispielanlage

3.2.2 Ansatz von Hasdemir, Kurtulan und Gören

In dem Paper [HKG07] wird ein Ansatz vorgestellt, mit dem ein Supervisor der als Automatenmodell entwickelt wurde, in die SPS Programmiersprache „Kontaktplan" transformiert wird. Der Ansatz verspricht eine Reduzierung des Speicherbedarfs infolge der Nichtbenutzung von „set"- und „reset"- Operatoren. Hierzu werden Bedingungen festgelegt, die eine Verriegelung des jeweiligen Zustands darstellen, indem nur logische „und" und „oder" Operatoren verwendet werden. In den folgenden zwei Gleichungen, in denen die Alternativen für „set"- und „reset"- Operatoren definiert werden, steht ein „." bzw. ein „$\prod$" für ein logisches „und" und ein „$\sum$" bzw. ein „+" für ein logisches „oder".

$$S_i(k) = \sum_{n \in I_{Sq}(i)} q_j(k).\sigma(i,j) \tag{3.13}$$

$$R_i(k) = \sum_{j \in I_{R\sigma}(i)} \sigma_j(k) \tag{3.14}$$

„k" bezeichnet in diesem Fall den „k-ten" Zeitpunkt und

$$\sigma(i,j) = \sum_{n \in I_{T\sigma(i,j)}} \sigma_n \tag{3.15}$$

entspricht allen Ereignissen, die einen Zustandswechsel von Zustand j zu Zustand i forcieren. Aus den Formeln 3.13 und 3.14 wird anschließend die Gleichung für den jeweiligen Zustand als Folgezustand zusammengesetzt. Die daraus resultierende Gleichung ist mit Formel 3.16 definiert.

$$q_i(k+1) = S_i(k) + q_i(k).R'_i(k) \tag{3.16}$$

Zur Vorbereitung werden für jeden Zustand eines Generators Mengen erstellt, die nachfolgend erläutert werden.

In $I_{sq}(i)$ werden alle Zustände eingetragen von denen ein Zustandswechsel zum Zustand i verursacht werden kann. $I_{sq}(i)$wird definiert als

$$I_{Sq}(i) = \{j \in I_q | \exists k \in I_{sigma}, f(q_j, \sigma_k) = q_i\}, \forall i \in I_q. \tag{3.17}$$

Alle Transitionen, die einen Wechsel zum Zustand i forcieren, bilden die Menge $I_{s\sigma}(i)$. Daraus folgt:

$$I_{S\sigma(i)} = \{j \in I_\sigma | \exists k \in I_q, f(q_k, \sigma_j) = q_i\}, \forall i \in I_q. \tag{3.18}$$

Die Menge aller Transitionen, die von einem Zustand i abgehen, sind in $I_{R\sigma}(i)$ definiert.

$$I_{R\sigma}(i) = \{j \in I_\sigma | \sigma_j \in \Gamma(q_i)\}, \forall i \in I_q \tag{3.19}$$

Des Weiteren wird für jedes mögliche Zustandspaar eine Menge mit den Transitionen gebildet, die einen Zustandswechsel von Zustand j zum Zustand i auslösen.

$$I_{T\sigma}(i,j) = \{k \in I_\sigma | \sigma_k \in \Gamma(q_j) \wedge k \in I_{S\sigma}(i)\}, \forall (i,j) \tag{3.20}$$

Darüber hinaus bekommt jeder Zustandsmerker Q_i einen „Hilfsmerker" q_i zugewiesen, der die durch die Nichtbenutzung von „set"- und „reset"- Operatoren verlorengegangene Verriegelung übernimmt. Hilfs- und Hauptzustandsmerker werden am Ende des AWL-Codes synchronisiert. Damit bei einem Neustart der SPS der Initialzustand gesetzt wird, kann ein spezielles Merkerbit genutzt werden. Der Merker SM0.1 wird bei einem Neustart der SPS gesetzt und kann parallel zu anderen Bedingungen in das Netzwerk des Initialzustands eingepflegt werden.

Beispiel für die Anwendung dieses Ansatzes.
Die Abbildung 3.4 zeigt den Generator G , der im weiteren Verlauf dieses Abschnitts in die Programmiersprache Kontaktplan mit Hilfe des Ansatzes aus [HKG07] konvertiert wird.

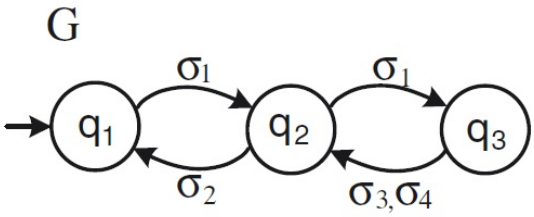


Abbildung 3.4: Beispiel Generator G

Aus den Gleichungen 3.17, 3.18, 3.19, 3.20 ergeben sich die Mengen 3.21 bis 3.33 für den Generator G.

$$I_{Sq}(1) = \{2\} \tag{3.21}$$

$$I_{Sq}(2) = \{1,3\} \tag{3.22}$$

$$I_{Sq}(3) = \{2\} \tag{3.23}$$

$$I_{S\sigma}(1) = \{2\} \tag{3.24}$$
$$I_{S\sigma}(2) = \{1,3,4\} \tag{3.25}$$
$$I_{S\sigma}(3) = \{1\} \tag{3.26}$$

$$I_{R\sigma}(1) = \{1\} \tag{3.27}$$
$$I_{R\sigma}(2) = \{1,2\} \tag{3.28}$$
$$I_{R\sigma}(3) = \{3,4\} \tag{3.29}$$

$$I_{T\sigma}(2,1) = \{1\} \tag{3.30}$$
$$I_{T\sigma}(1,2) = \{2\} \tag{3.31}$$
$$I_{T\sigma}(3,2) = \{1\} \tag{3.32}$$
$$I_{T\sigma}(2,3) = \{3,4\} \tag{3.33}$$

Aus den soeben gebildeten Mengen werden die formalen Ausdrücke für die jeweiligen Zustände hergeleitet. Diese ergeben sich wie folgt:

$$q_1(k+1) = q_2(k).\sigma_2 + q_1(k).\sigma_1' \tag{3.34}$$
$$q_2(k+1) = q_1(k).\sigma_1 + q_3(k).(\sigma_3 + \sigma_4) + q_2(k).\sigma_1'.\sigma_2' \tag{3.35}$$
$$q_3(k+1) = q_2(k).\sigma_1 + q_3(k).\sigma_3'.\sigma_4'. \tag{3.36}$$

Diese formalen Ausdrücke werden daraufhin in Kontaktplan übersetzt. Der Strich „′" an einem Ereignis bedeutet eine Negation des Signals. Abbildung 3.5 zeigt den zu dem Generator G passenden Kontaktplan. Die Netzwerke 4 bis 6 stellen die Synchronisierung der Zustandsmerker mit den Hilfsmerkern dar, die die Verriegelung der Zustände gewährleistet.

3.3 Ansatz von Uzam, Gelen und Dalci

In diesem Abschnitt wird der Ansatz von Uzam et. al [UGD09] beschrieben, der in der vorangegangenen Arbeit von [Url12] für die Generierung eines AWL-Codes verwendet wurde. Die Konvertierung erfolgt, wie bei anderen Ansätzen, mit Hilfe von Merkerbits für jeden Zustand. Ein Zustandswechsel wird entweder durch eine nicht steuerbare oder durch eine steuerbare Transition durchgeführt. Steuerbare Transitionen werden erst am Ende des Programmzyklus gesetzt, um die Nichtberücksichtigung der nicht steuerbaren Ereignisse zu vermeiden. Ein grundlegendes Problem welches Uzam et. al behandeln ist, dass bei Ausgängen die keine Selbsthaltung besitzen der Ausgang nur für die Zykluszeit des Programmes gesetzt ist. Danach wird der Ausgang wieder zurück gesetzt. Für dieses Problem ist in [UGD09] folgende Lösung beschrieben.
Es werden sogenannte Aktionen festgelegt, die die Ausgänge der SPS repräsentieren. Diese Aktionen sind nicht steuerbar und werden als Selfloops an die Zustände gesetzt, in denen der jeweilige Ausgang gesetzt werden soll. Es ist nicht erlaubt die Aktionen als Transitionen zu nutzen. Des weiteren ist eine Rückmeldung für die Beendigung der Aktionen erforderlich, die entweder durch einen Endschalter für eine erreichte Position signalisiert wird oder durch ein Ereignis, das darauf verweist, dass eine Aktion abgeschlossen ist. Abbildung 3.6 zeigt einen Automaten, der mit der Methode von Uzam et. al erweitert wurde.

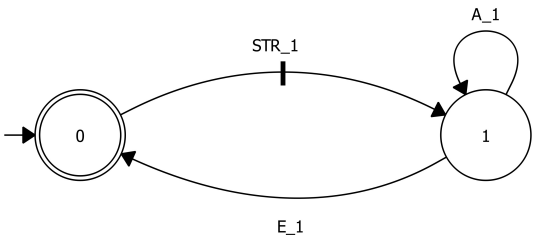


Abbildung 3.6: Beispiel für das Setzen der Ausgänge nach Uzam

Eine detailliertere Beschreibung dieser Methode kann in [UGD09] nachgelesen werden.

Tabelle 3.3: Ereignisbeschreibung der Beispielanlage

Ereigniskategorie	**Ereignisse**	**Beschreibung**
Verriegelungsereignis (Zuweisung einer „ressource")	M_1	Maschine 1 ist zugewiesen
	M_2	Maschine 2 ist zugewiesen
	R_1	Reparateur ist Maschine 1 zugewiesen
	R_2	Reparateur ist Maschine 2 zugewiesen
Entriegelungsereignisse (Entziehen einer „ressource")	$\overline{M_1}$	Maschine 1 ist entzogen
	$\overline{M_2}$	Maschine 2 ist entzogen
	$\overline{R_1}$	Reparateur ist Maschine 1 von Reparaturarbeiten entzogen
	$\overline{R_2}$	Reparateur ist Maschine 2 von Reparaturarbeiten entzogen
Startereignisse (Start einer Aktivität)	s_1	Betrieb der Maschine 1 startet
	s_2	Betrieb der Maschine 2 startet
	sr_1	Reparatur der Maschine 1 startet
	sr_2	Reparatur der Maschine 2 startet
Endereignisse (Ende einer Aktivität)	f_1	Betrieb der Maschine 1 beendet
	f_2	Betrieb der Maschine 2 beendet
	r_1	Reparatur der Maschine 1 beendet
	r_2	Reparatur der Maschine 2 beendet
Systemereignisse	b_1	Maschine 1 ist defekt
	b_2	Maschine 2 ist defekt

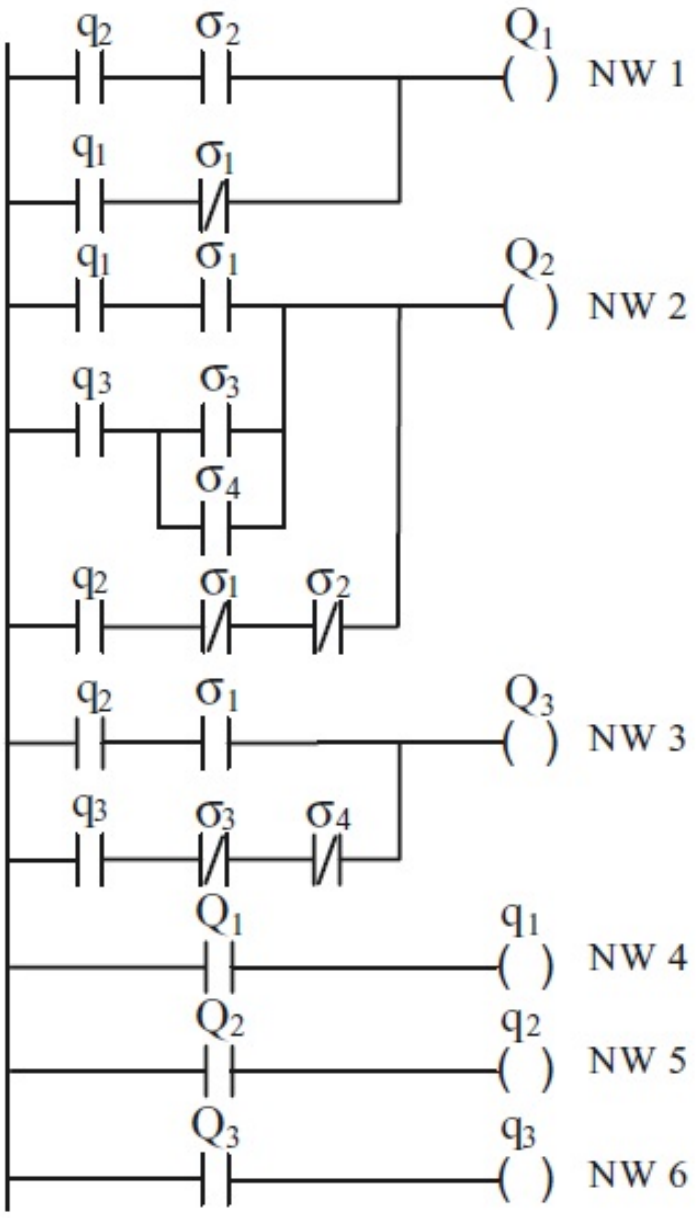


Abbildung 3.5: Kontaktplan für Generator G

Kapitel 4

Fertigungszelle

In dieser Arbeit werden die Streckenmodelle und Spezifikationen einer Fertigungszelle genutzt. Diese Fertigungszelle automatisiert einen Stückgutprozess, der unterschiedliche Werkstücke nach ihrer Farbe und Qualität ordnet. Die Anlage besteht aus den folgenden Komponenten:

- einer Lagereinheit mit Schwenkarm
- einer Transporteinrichtung
- einem Handling Portal und
- einer Robotereinheit.

Diese Anlagenteile werden mittels Speicherprogrammierbarer Steuerung automatisiert. Im folgenden Kapitel werden die für den Prozess notwendige Hardware und die mit dem modularen Ansatz entwickelten Spezifikationen beschrieben. Einige Bilder, die im weiteren Verlauf verwendet werden, wurden der Arbeit von [Url12] entnommen.

4.1 Hardware und Produktionsprozess

Lagereinheit mit Schwenkarm.
Die Lagereinheit beinhaltet einen Ausschieber, der die Werkstücke aus einem Magazin heraus in Position bringt, um sie anschließend mit einem Schwenkarm der Transporteinheit zuzuführen. Dem durch das Magazin mit Ausschieber dargestellten Lager sind folgende Sensorsignale zugeordnet:

- Endschalter des Ausschiebers für eingefahrenen Zustand
- Endschalter des Ausschiebers für ausgefahrenen Zustand
- Lichtschranke, die signalisiert, wenn das Magazin leer ist.

Der Schwenkarm besitzt eine Vakuumeinheit, mit der er die Werkstücke festhalten kann. Auf diese Weise nimmt der Schwenkarm die Werkstücke, die aus dem Lager geschoben werden, auf und befördert sie zur Transporteinheit. Die untere Auflistung zeigt die Sensoren des Schwenkarms.

- Endschalter, wenn Position Lagereinheit erreicht ist
- Endschalter, wenn Position Transporteinrichtung erreicht ist
- Sensor zum Registrieren eines Werkstücks an der Vakuumeinheit

Um die Lagereinheit steuern zu können müssen folgende Aktoren aktiviert werden:

- Ventil zum Ausfahren des Ausschiebers (das Einfahren wird durch eine Feder realisiert, wenn das Ventil verriegelt wird),
- zwei Ventile zum Anfahren der Positionen „Lagereinheit" und „Transporteinrichtung" des Schwenkarms
- Vakuumeinheit zum Ansaugen des Werkstücks
- Drucklufteinheit zum Abstoßen des Werkstücks.

Abbildung 4.1 zeigt eine Darstellung der Lagereinheit und des Schwenkarms.

Transporteinrichtung.
Die nächste Komponente im Produktionsprozess ist die Transporteinrichtung. Mit ihr werden die Werkstücke zum Handling Portal befördert. Die Transporteinheit ist mit einer isel-Doppelspureinheit, einem Schrittmotor und einem Kugelgewindetrieb mit Schlitten ausgestattet. Die Doppelspureinheit ist aus einem Aluminiumprofil gefertigt und der Schrittmotor gewährleistet durch seine Genauigkeit eine Positionsreproduzierbarkeit. Der Schlitten ist so gebaut, dass fünf Werkstücke darauf abgelegt werden können. Der Motor wird durch die Baugruppe FM353 aktiviert, welche im weiteren Verlauf dieses Kapitels näher beschrieben wird. Abbildung 4.2 zeigt eine Zeichnung der Transporteinheit.

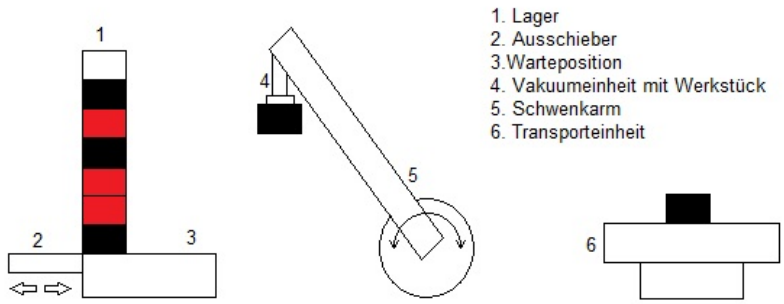


Abbildung 4.1: Lagereinheit inklusive Schwenkarm

Handling Portal.
Das Handling Portal ist eine Einheit zum Transportieren der Werkstücke. Es nimmt die Werkstücke auf, die sich auf dem Transportschlitten befinden wenn dieser die Endposition „Handling Portal" erreicht hat. Dafür ist das Portal mit zwei Achsen ausgestattet, die den Greifer in horizontaler und vertikaler Richtung bewegen können. Welche der Positionen (Transporteinrichtung, Rutsche 1, Rutsche 2) erreicht ist, wird von Endschaltern signalisiert. In jeder Endposition kann der Greifer in vertikaler Richtung bewegt, geöffnet und geschlossen werden. Im Folgenden ist die Sensorik des Handling Portals aufgelistet:

- Endschalter wenn Position Transporteinrichtung erreicht ist
- Endschalter wenn Position Rutsche 1 erreicht ist
- Endschalter wenn Position Rutsche 2 erreicht ist
- Endschalter wenn Position Greifer unten erreicht ist
- Endschalter wenn Position Greifer oben erreicht ist
- Sensor zum Registrieren eines roten Werkstücks am Greifer

Der Sensorik des Greifers ist es zwar nur möglich rote Werkstücke zu detektieren, jedoch reicht dies aus um zwischen roten und schwarzen Werkstücken zu unterscheiden. Die horizontale Achse wird mittels Pneumatik zwischen Transporteinrichtung, Rutsche 1

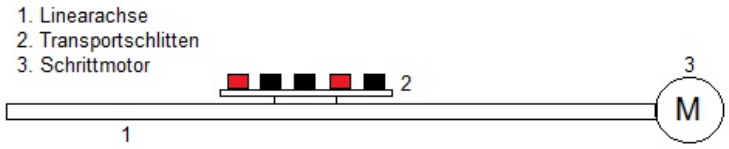


Abbildung 4.2: Transporteinheit

und Rutsche 2 positioniert. Durch Öffnen der entsprechenden Ventile wird der Greifer aus seiner Ruheposition („oben", „geschlossen") in Position „unten" gebracht bzw. geöffnet. Eine schematische Darstellung des Handling Portals ist in Abbildung 4.3 zu sehen.

Robotereinheit.
Die Einheit besteht aus einen mit fünf Freiheitsgraden ausgestatteten Vertikal-Roboter, an dessen Arm eine steuerbare Hand angebracht ist. Die Anbindung des Roboters erfolgt über eine Drive Unit, die per serieller Schnittstelle (RS-232) mit einem Steuerungs-PC verbunden ist. Dieser PC kann mit Hilfe von Profinet mit der SPS Steuerung kommunizieren. Die Programmierung des Roboters wird über den Steuerungsrechner entwickelt, der dann über Merkerbits der SPS den Roboter ansteuern kann.
Bevor mit dem Roboter gearbeitet werden kann muss eine Nestfahrt durchgeführt werden, in der sich der Vertikal-Roboter initialisiert. Während dieser Fahrt werden alle Endlagen, die der Roboter erreichen kann, angefahren und so ein exakter Startzustand festgelegt. Hiernach ist es dem Roboter möglich folgende Fahrten durchzuführen:

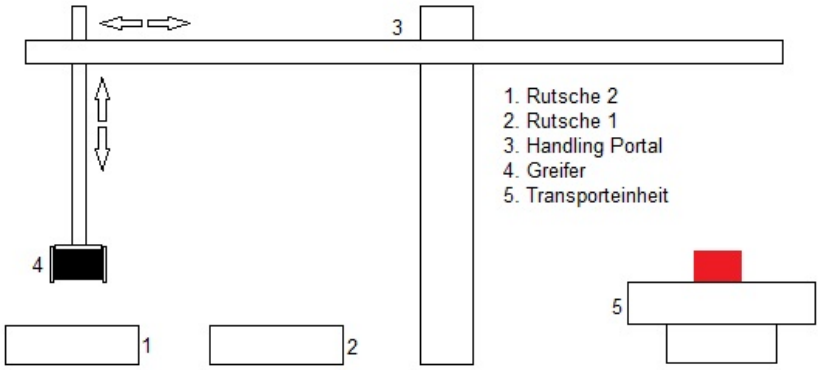


Abbildung 4.3: schematische Darstellung des Handling Portals

- Werkstück von Rutsche 1 zu RFID Station befördern
- Werkstück von Rutsche 2 zu RFID Station befördern
- Werkstück von RFID Station zu Endlager „ok" befördern
- Werkstück von RFID Station zu Endlager „Ausschuss" befördern
- Werkstück von RFID Station zum Rohlager befördern
- Ruheposition anfahren

Die einzelnen Fahrten werden über eine Software der Firma Crescent Software Inc. direkt an die Drive Unit gesendet, welche dann für die Lageregelung der Motoren verantwortlich ist und die gewünschten Positionen anfährt. In Abbildung 4.4 ist ein Bild des vertiakl-Roboters zur besseren Verständlichkeit dargestellt.

RFID Station.
Mit der von Siemens gefertigten RFID Station ist es möglich jedes Werkstück zu markieren oder zu beschreiben. Die Werkstücke verfügen über einen SIMATIC RF320T Transponder, der einen 20 Byte großen Datenspeicher beinhaltet. Die RFID Station kann mit der SIMATIC RF340R Einheit die Werkstücke beschreiben und auslesen. Hierfür darf das Werkstück

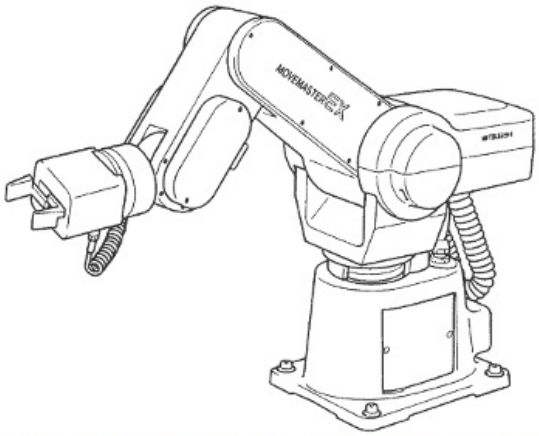


Abbildung 4.4: Vertikal-Roboter

einen maximal 60mm großen Abstand von der Station haben. Für die Kommunikation steht die Komponente SIMATIC RF180R zur Verfügung, die wiederum über Profinet mit der SPS kommuniziert. Eine detailliertere Beschreibung ist in dem Systemhandbuch [SIE] zu finden.

Weitere technische Hardware.
Der Kern der Automatisierung wird von einer Siemens SIMATIC S7-300 Station geformt, die mit einer CPU 315-2 PN/DP ausgestattet ist. Die CPU hat einen Merkerbereich von 2048 Byte und kann maximal 1024 Bausteine speichern. Sie verfügt auch über eine PN Schnittstelle für Profinet und eine DP Schnittstelle, um weitere Peripherien anzuschließen. Als Ein-, bzw. Ausgabebaugruppen wurden zwei digitale Baugruppen gewählt, die jeweils über 16 Ein- und 16 Ausgänge verfügen. Das für die Kommunikation zuständige Netzwerk wird mit Profinet realisiert und besitzt eine Sterntopologie, die durch einen SCALANCE X208 Switch arrangiert wird.

Der Produktionsprozess.
Die Anlage soll einen Produktionsprozess simulieren, in dem Werkstücke aus einem Lager über eine Transporteinrichtung und eine Analyseeinheit in ein Endlager gebracht werden bzw. dem Produktionsprozess wieder zugeführt werden, falls dies nötig ist. Die Anlage wird schematisch in Abbildung 4.5 gezeigt. Dort wird die Bewegung der Werkstücke durch Pfeile angegeben.

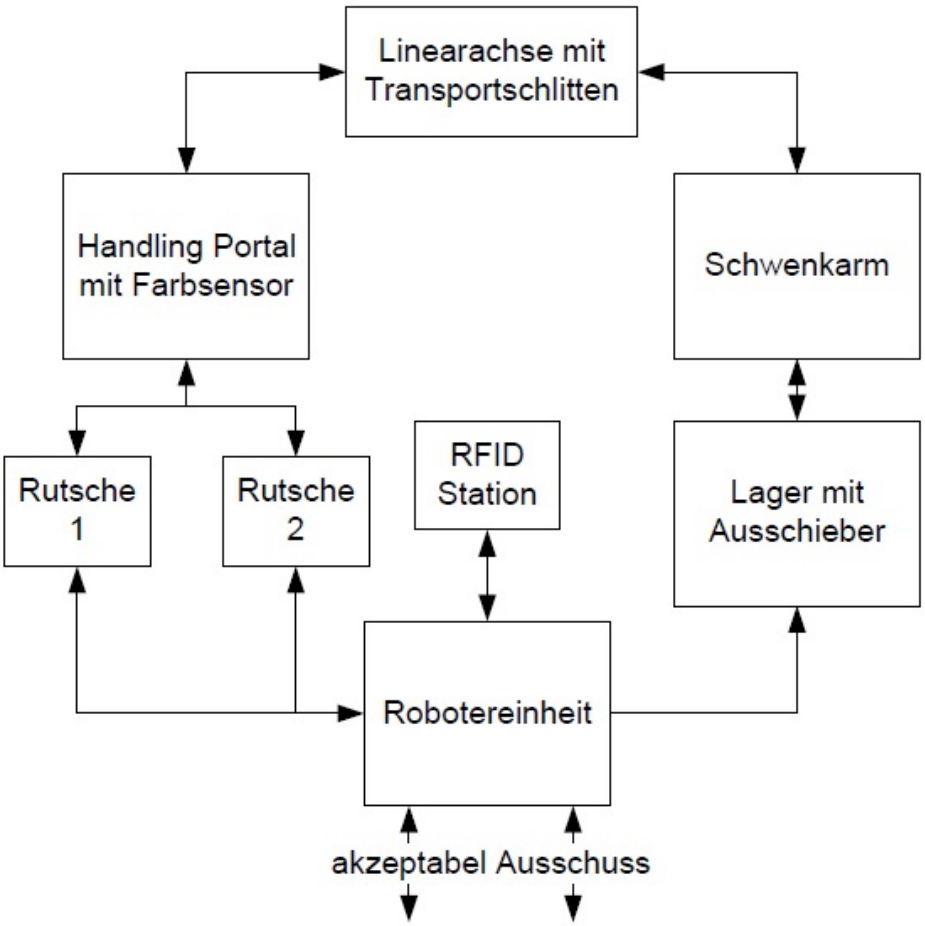


Abbildung 4.5: schematische Darstellung der Anlagenkomponenten

Durch den Ausschieber wird ein Werkstück in die Warteposition für den Schwenkarm gebracht, der das Werkstück von dort aus auf die Transporteinrichtung ablegen kann. Der Transport der Werkstücke ist auch entgegengesetzt möglich. Die Transporteinrichtung kann zwischen dem Handling Portal und der Lagerposition in beide Richtungen bewegt werden. Das Handling Portal kann Werkstücke der Transporteinheit entnehmen und auch hinzufügen. Es kann auch die Werkstücke, die es dem Schlitten entnommen hat, auf beiden vorhandenen Rutschen platzieren. Der Roboter kann die Werkstücke in die beiden Endlager befördern, die RFID Station anfahren oder das Werkstück wieder in den Produktionsprozess einfügen, indem der Roboter das Werkstück erneut dem Rohlager zuführt.

Tabelle 4.1: Ereignis Nomenklatur

Hardwareklasse	Beschreibung
A	Ausgang einer SPS (nicht steuerbar)
E	Eingang einer SPS (nicht steuerbar)
STR	Steuerbares Ereignis
WS	Ereignisse die von der RFID Station generiert werden

Tabelle 4.2: Komponenten Nomenklatur

Komponente	Beschreibung
LA	(Roh)Lagereinheit
LA_AS	Ausschieber der Lagereinheit
SA	Schwenkarm
LI	Linearachse (Transporteinheit)
HU	Handling Unit
R	Vertikal-Roboter
LS	Lichtschranke

4.2 gegebene Modelle

Der vorangegangenen Arbeit von [Url12] sind die Spezifikationen der einzelnen Komponenten sowie das Streckenmodell entnommen, die die Fertigungszelle beschreiben. Um die Generatorenmodelle der Steuerungen und der Strecke verständlich zu machen, sind in 4.1 und in 4.2 Erklärungen für die Zusammensetzung der Namen und Ereignisse zu finden. Diese werden in der Form „Ereignisklasse_Komponente_Aktion" dargestellt.

Die im Anschluss aufgeführten Spezifikationen sind mit dem modularen Ansatz berechnet worden. Die Generatormodelle der Strecke und der Spezifikationen wurden mit DESTool erstellt. Die Steuerbarkeit und das Nichtblockieren sind in der Arbeit von [Url12] nachgewiesen. Zur Modellbildung der Strecke wurde die Anlage in Komponenten unterteilt, wovon zwei der Anlagenteile im Anschluss beschrieben werden.

Lagereinheit inklusive Ausschieber.
Abbildung 4.6 zeigt das Generatormodell für den Ausschieber. Das Modell besteht aus

drei Zuständen. Dem Initialzustand „1”, in dem der Ausschieber eingefahren ist, Zustand „2”, in dem der Ausschieber durch den Ausgang „A_LA_AS_vor” „ausgefahren wird und Zustand „3”, in dem der Ausschieber ausgefahren ist. Wenn im Initialzustand das Ereignis „STR_LA_AS_vor” aktiviert wird, folgt ein Übergang zu Zustand „2”. Ist der Ausschieber komplett ausgefahren wird dies durch den Eingang „E_LA_AS_vor” signalisiert und ein Zustandswechsel zum Zustand „3” bewirkt. Durch den Eingang „E_LA_AS_zu” wird der eingefahrene Ausschieber gekennzeichnet und somit der Zustand „3” verlassen und der Initialzustand „1” aktiviert.

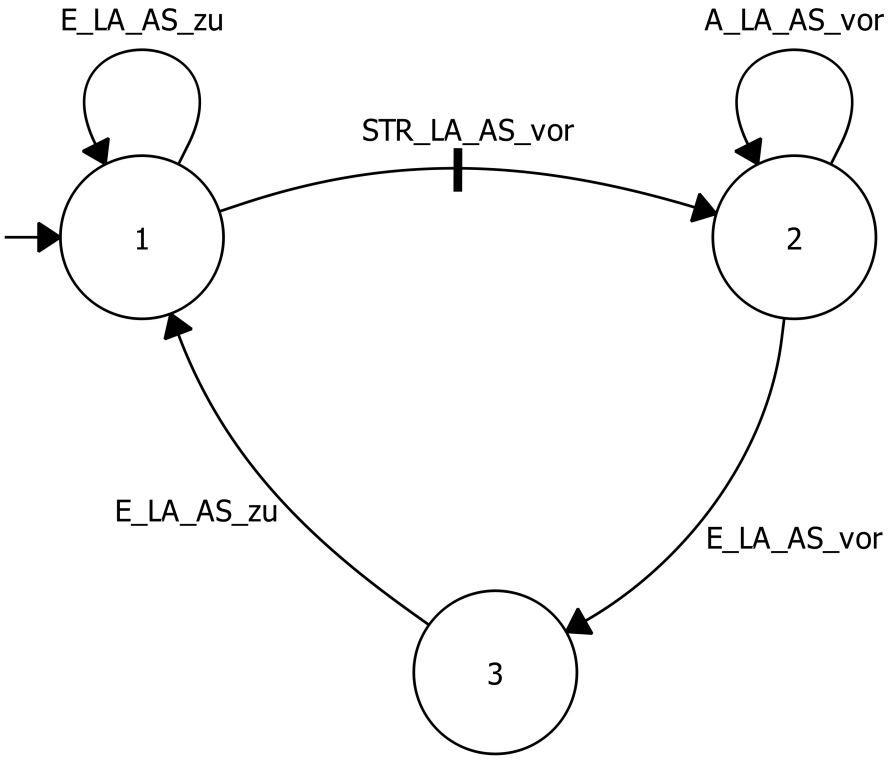


Abbildung 4.6: Generatormodell Ausschieber

Die Spezifikation für den Ausschieber, soll den Ablauf der Strecke bestimmen. Es soll nur dann ein Werkstück aus dem Lager geschoben werden, wenn das Lager nicht leer ist. Das Ausschieben eines weiteren Werkstücks soll erst dann erfolgen, wenn der Schwenkarm das vorige Werkstück dem Lager entnommen hat. Damit ein Werkstück in die Wartepo-sition gebracht wird, muss der Ausgang „A_LA_AS_vor” solange zugelassen werden bis der Ausschieber seine Endposition zurückmeldet. Abbildung 4.7 zeigt die Spezifikation des

Ausschiebers als Generatormodell.

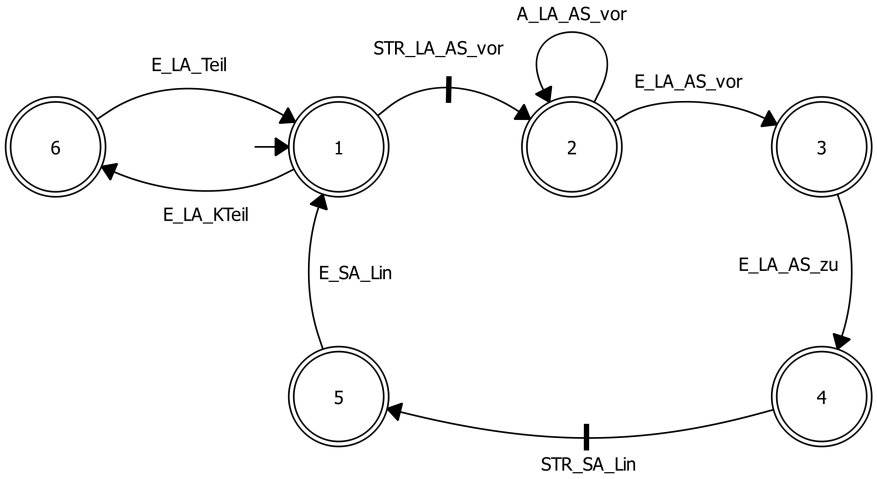


Abbildung 4.7: Spezifikation des Ausschiebers als Automat

Vakuumeinheit.
Dieser Anlagenteil beschreibt den Schwenkarm mit der Vakuumeinheit. Er ist in drei unabhängige Teilkomponenten partitioniert, welche sich einzeln modellieren lassen. Die daraus resultierenden Modelle bestehen aus der Bewegung des Schwenkarms, der Detektierung eines Werkstücks am Greifer und der Ansteuerung der Vakuumeinheit. Nachfolgend wird die Modellierung der Strecke und des Supervisor der Vakuumeinheit beschrieben.

Die Vakuumeinheit dient zum Ansaugen der Werkstücke und kann diese auch mittels Druckluft wieder abstoßen. Beide Funktionen sind bei der Initialisierung inaktiv. Durch das steuerbare Ereignis „STR_SA_VAK_ein” wird das Vakuum eingeschaltet und meldet nach Ablauf einer unterlagerten Schrittkette, die das Ventil wieder schließt, das Ereignis „E_SA_VAK_aus”.
Sobald der Schwenkarm das Lager erreicht hat, wird das Vakuum eingeschaltet. Wird nach Ablauf einer unterlagerten Schrittkette kein Werkstück am Sauger detektiert, wiederholt sich die Schrittkette bis ein Werkstück fest am Schwenkarm angesaugt ist. Wenn der Schwenkarm die Endposition an der Transporteinrichtung erreicht hat („E_SA_Lin” ist aktiv), kann das Werkstück mit Druckluft wieder vom Schwenkarm gelöst werden. In Abbildung 4.9 ist die Spezifikation der Vakuumeinheit als Generator zu sehen.

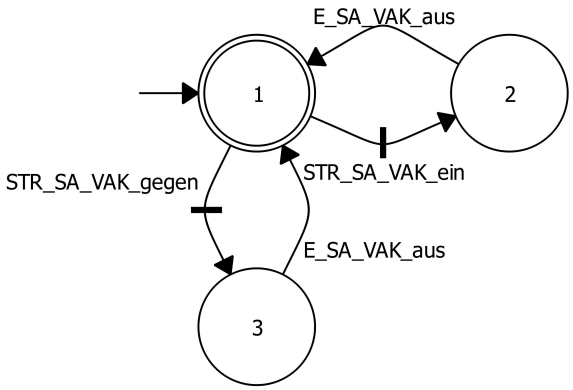


Abbildung 4.8: Generator Vakuumeinheit des Schwenkarms

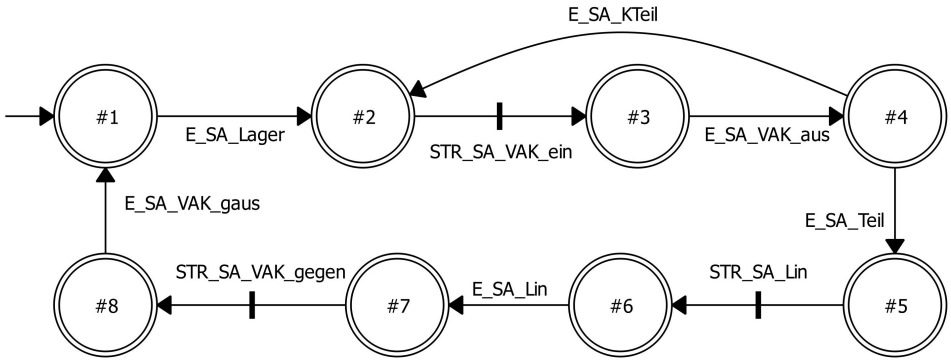


Abbildung 4.9: Spezifikation der Vakuumeinheit

In Tabelle 4.3 ist eine Auflistung aller Streckenkomponenten mit dem jeweils entsprechenden Alphabet Σ und der Zustandsmenge X gegeben. Die einzelnen Spezifikationen der Komponentenmodelle sind in Tabelle 4.4 dargestellt. Eine ausführlichere Beschreibung der Streckenmodelle und der Spezifikationen ist in der Arbeit von Urland [Url12] enthalten.

Tabelle 4.3: Streckenmodelle der Komponenten

Komponente	**Größe** Σ_c	**Größe** Σ_{uc}	**Transitionen**	**Größe der Zustandsmenge** X
Ausschieber Lager	1	3	5	3
Schwenkarm	3	3	6	4
Schwenkarm Sensor	2	0	4	2
Schwenkarm Vakuum	2	2	4	3
Transporteinrichtung	3	3	6	4
On/Off Schalter	2	0	4	2
Sensor Lager	0	2	4	2
RFID Werkstück akzeptabel	0	1	1	1
RFID Werkstück Ausschuss	0	1	1	1
RFID Werkstück reparabel	0	1	1	1
Handling Portal (horizontale Achse)	3	5	19	4
Handling Portal (Greifer)	2	0	2	2
Handling Portal (vertikale Achse)	2	3	5	2
Sensor Entnahme-Position	0	4	4	2
Robotereinheit	6	6	13	7
Lichtschranke Rutsche 1	0	1	1	1
Lichtschranke Rutsche 2	0	1	1	1
Signal „Rutschen leer”	0	1	1	1
Gesamtmodell	24	39	16570368	516096

Tabelle 4.4: Spezifikationen der Streckenkomponenten

Spezifikation	**Größe** Σ_c	**Größe** Σ_{uc}	**Transitionen**	**Größe der Zustandsmenge** X
K1 B1	2	0	2	2
K2 B2	1	1	6	6
K3 Ausschieber	2	6	8	6
K4 Schwenkarm	5	9	12	11
K5 Schwenkarm Vakuum	3	6	9	8
K6 Beladung	4	5	11	9
K7 Handling Unit	7	54	18	19
K8 Transporteineit	3	0	10	10
K9 On/Off	24	0	25	2
K10 Roboter & RFID	5	11	30	21
K11 Puffer 1	2	0	12	7
K12 Puffer 2	2	0	12	7

Kapitel 5

Codegenerierung für die Fertigungszelle

5.1 Adaption von DES2IEC

Für die Adaption des Codegenerators „DES2IEC” wird das bestehende Programm durch zwei Unterfunktionen erweitert. Eine, um den Code für die neuen Ansätze von Liu et. al, und eine andere um den von Hasdemir et. al zu generieren. Außerdem wird die Funktion „dataInput()” um zwei Optionen für die neuen Ansätze ergänzt. Um den neuen AWL-Code nach [HKG07] und nach [JD02] zu generieren, wird die Funktion „ScanFile()” der Version 2.01 des Programmes DES2IEC genutzt. Diese Funktion durchsucht die mit DESTool 0.59 erstellte Projektdatei und filtert die Attribute des geladenen Supervisors heraus. Eine detaillierte Beschreibung zum Programm DES2IEC in der Version 2.01 ist in [Url12] zu finden. Für die DESTool Projekte die mit „DES2IEC_Multi” eingelesen und bearbeitet werden, ist zu beachten, dass die Nomenklatur aus Tabelle 4.1 eingehalten wird.

5.1.1 Ansatz von Liu und Darabi

In dem Ansatz nach Liu et. al wird die Ereignismenge Σ in die Teilmengen Σ_s, Σ_f und Σ_{sm}unterteilt. Außerdem werden die „Activity-ressource” und „State-ressource” Vektoren erstellt. Der „State-ressource” Vektor gibt an, welche Aktion in dem aktuellen Zustand angesteuert wird. Die aus den Gleichungen 3.7, 3.8 und 3.9 erhaltenen Daten werden für die Konvertierung in AWL-Code genutzt.
Die Unterfunktion „createLiu()” übernimmt die Codegenerierung nach dem Ansatz von Liu et. al. Das in der Funktion „ScanFile()” erzeugte Stringarray „Transitionen” wird mit dem „strtok”-Befehl in die Ausgangszustände, die Transitionen und die Endzustände unterteilt und in die Arrays „Qv”, „Tr” und „Qz” geschrieben. In den Spezifikationen werden die Ausgänge mit Hilfe von Signalen, die als Selfloops an den Zuständen hängen, verarbeitet, damit eine Selbsthaltung garantiert wird. Diese Signale repräsentieren die „Aktivitäten”, die in dem

aktiven Zustand ausgeführt werden. Abbildung 5.1 zeigt den groben Programmablaufplan der Funktion „createLiu()".

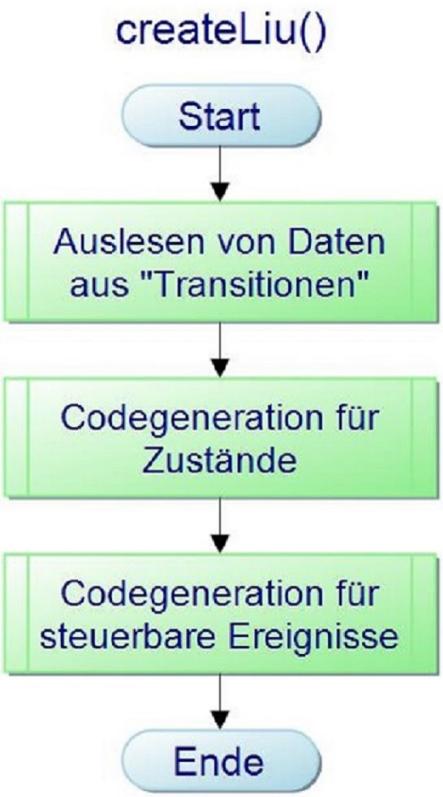


Abbildung 5.1: Programmablaufplan der Funktion „createLiu()"

Die Funktion „createLiu()" kann in Unterfunktionen aufgeteilt werden:

- das Auslesen und Separieren der Daten aus der DESTool Projektdatei
- die Codegenerierung für die einzelnen Zustände und
- die Codegenerierung für die steuerbaren Ereignisse

Abbildung 5.2 beschreibt das Auslesen der Daten, welches nicht nur in der Funktion „createLiu()" genutzt wird, sondern auch in der Funtkion „createHasdemir()", die im nächsten Abschnitt erläutert wird.

Abbildung 5.2: Auslesen der Daten aus „Transitionen”

Abbildung 5.3 zeigt den Ablauf der Codegenerierung für die einzelnen Zustände nach dem Ansatz von [JD02]. Zuerst werden alle steuerbaren Ereignisse gefiltert und in einem seperaten Array gespeichert. Anschließend wird in einer for-Schleife das Array „Transitionen” durchlaufen. Wenn es sich um einen Selfloop handelt, wird nur ein Zähler inkrementiert und die nächste Transition wird untersucht. Wenn es sich bei der Transition nicht um einen Selfloop handelt, wird für den Ausgangszustand der Transition der AWL-Code mit Hilfe der separierten Daten in die Textdatei „Step7AWLCodeLiu” geschrieben. Es wird die Transition für den Zustandswechsel, das Blockbit (in der Nomenklatur von [JD02] wird das Blockbit es_σ genannt) die zu setzenden Ausgänge und die rückzusetzenden Ausgänge berücksichtigt.

Zum Schluss müssen in den Zuständen von denen ein steuerbares Ereignis ausgeht, diese steuerbaren Ereignisse angesteuert werden. Die Abbildung 5.4 zeigt den Ablauf dieser Unterfunktion.

Mittels einer for-Schleife wird das Alphabet untersucht. Wenn ein Ereignis steuerbar ist, werden die Zustände, in denen das Ereignis aktiviert werden muss, mit „oder"-Gattern verbunden und am Ende das steuerbare Ereignis angesteuert.

5.1.2 Ansatz von Hasdemir, Kurtulan und Gören

Die Methode, die von Hasdemir et. al anwandt wird, kreiert zuerst die Mengen I_{Sq}, $I_{S\sigma}$, $I_{T\sigma}$ und $I_{R\sigma}$. Alle Informationen, die hierfür notwendig sind, befinden sich in dem Stringarray „Transitionen". Ein Algorithmus, dessen Ablauf in Abbildung 5.2 gezeigt ist, teilt die einzelnen Strings aus „Transitionen" in mehrere Arrays ein. Anschließend wird in einer for-Schleife der AWL-Code für jeden Zustand generiert. Hierfür kann die Funktion „createHasdemir()" nach dem Auslesen der Daten in vier weitere Unterfunktionen unterteilt werden. Die Aufteilung der Transitionen, die Generierung des Codes für den Zustandswechsel, das Aktivieren der steuerbaren Ereignisse und Ausgänge sowie das Synchronisieren der Hauptzustandsmerker mit den Hilfszustandsmerkern. In Abbildung 5.5 wird der Ablauf der gesamten Funktion gezeigt.

In Abbildung 5.6 werden mittels einer for-Schleife alle Transitionen durchlaufen. Wenn die Transition kein Selfloop ist und zum Zustand „i" führt, wird diese Transition in das zweidimensionale Array „QZ" geschrieben. Wenn die Transition kein Selfloop ist und vom Zustand „i" wegführt, wird die Transition in das Array „QV" geschrieben. Beim Speichern der Transitionen in den Arrays werden entweder der Zähler „zumZustand" oder „vonZustand" inkrementiert.

Abbildung 5.7 zeigt den Ablauf für die Codegenerierung des restlichen AWL-Codes für den aktuellen Zustand. Als erstes werden alle Merker von eventuellen Folgezuständen mit negiertem „und"-Gatter verbunden und an den AWL-Code gehängt. Anschließend wird geprüft wie viele Transitionen zum Zustand „i" führen. Diese werden dann mit den entsprechenden Vorgaben nach Hasdemir et. al verbunden und an den schon generierten Code gesetzt. Zum Schluss wird der Hauptmerker des Zustands „i" angesteuert.

In Abbildung 5.8 wird der Ablauf für die Behandlung der steuerbaren Ereignisse und der Ausgänge gezeigt. In zwei for-Schleifen wird untersucht, ob das aktuelle Element des Alphabets ein steuerbares ist. Ist dies der Fall, werden alle Zustände, in denen dieses Ereignis aktiviert werden soll, mit „oder"-Gattern verbunden und am Ende das steuerbare Ereignis angesteuert. In einer zweiten Schleife wird das Alphabet nach Ausgängen gefiltert. Ist das

Element ein Ausgang, werden alle Zustände, in denen der Ausgang aktiviert werden muss mit einem logischen „oder” verbunden, und am Ende der Ausgang angesteuert.

Am Ende des AWL-Codes werden die Hauptmerker der Zustände mit ihren Hilfsmerkern synchronisiert, um eine Selbsthaltung zu generieren. Dieser Vorgang ist in Abbildung 5.9 dargestellt.

5.2 Codegenerierung

In diesem Abschnitt werden die beiden in Kapitel 4 detailliert beschriebenen Spezifikationen (K3 für den Ausschieber und K5 für die Vakuumeinheit) dem generierten Code nach [HKG07] und nach [JD02] gegenübergestellt. Abbildung 5.10 zeigt den Automaten der Spezifikation des Ausschiebers und den generierten Code nach [JD02], Abbildung 5.11 stellt den Supervisor K5, der die Vakuumeinheit repräsentiert, dem generierten AWL-Code nach [JD02] gegenüber.

Aus einem Vergleich der Abbildungen 5.10 und 5.11 geht hervor, dass der AWL-Code nach der Methode von Liu et. al generiert ist. Außerdem ist zu erkennen, dass die Steuerungsaufgabe des Supervisors erfüllt wird. Wie in der Theorie beschrieben, werden die Ausgänge in den Netzwerken des entsprechenden Zustands gesetzt bzw. rückgesetzt und die Transitionen werden wie gefordert durchgeführt.

Die Abbildungen 5.12 und 5.13 stellen die Spezifikationen K3 und K5 dem jeweiligen AWL-Code gegenüber, der mit der Methode von [HKG07] generiert wird.

Die Gegenüberstellung aus Abbildung 5.12 zeigt zum einen die Spezifikation K3 des Ausschiebers in Form des Generatormodells, und zum anderen den aus diesem Automaten generierten AWL-Code nach den Vorgaben von Hasdemir et. al. Der Code ist nach den Regeln, die in Kapitel 3.22 detailliert erläutert werden, generiert worden. Der Selfloop an Zustand „2” ist als Ausgang detektiert, und wird am Ende des AWL-Codes angesteuert, sobald der Zustand aktiv ist. Wie zu erkennen ist, wird in diesem Ansatz weder eine „set”- noch eine „reset”-Operation verwendet, was nach Hasdemir et. al eine Speicherbedarfreduzierung zur Folge haben soll. Am Ende des Codes werden wie in der Theorie beschrieben, die Hilfs- und Hauptmerker der Zustände synchronisiert, um eine Benutzung von „set”- und „reset”-Operatoren zu umgehen.

Die Gegenüberstellungen zeigen, dass der von „DES2IEC_Multi" generierte AWL-Code der Methode von Hasdemir et. al entspricht. Für jeden Zustand wurde ein Hilfsmerker „qi" definiert („i" steht für die Nummer des Zustands im Generatormodell), der die Selbsthaltung gewährleistet und am Ende des AWL-Codes mit dem Hauptzustandsmerker „Qi" synchronisiert wird. Daneben werden alle abgehenden Kanten eines Zustands mit einem negierten „und" verbunden. Im weiteren Verlauf des AWL-Codes, für den aktuellen Zustand „i", werden alle Kombinationen aus Zustand und Transition, die zu einem Zustandswechsel zum Zustand „i" führen, mit einem logischen „oder" an den Code gehängt. Komplettiert wird das Netzwerk für Zustand „i", indem der Hauptmerker „Qi" angesteuert wird. Am Code für Zustand „2" in Abbildung 5.13 ist zu erkennen, das auch Zustände richtig verarbeitet werden, die ankommende Kanten von mehr als einem Zustand aufweisen. Die für diesen Supervisor gewünschte Steuerungsaufgabe wird erfüllt.

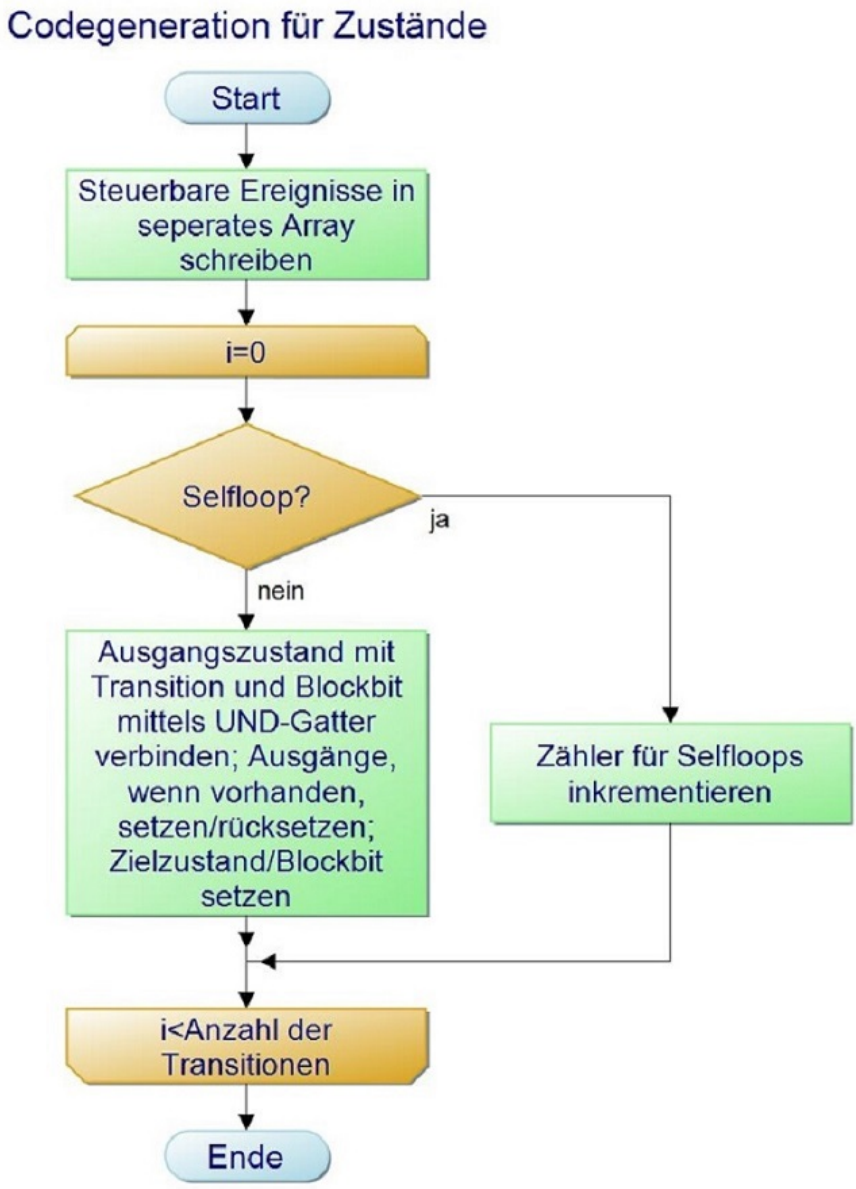


Abbildung 5.3: Codegeneration für die Zustände des Supervisor

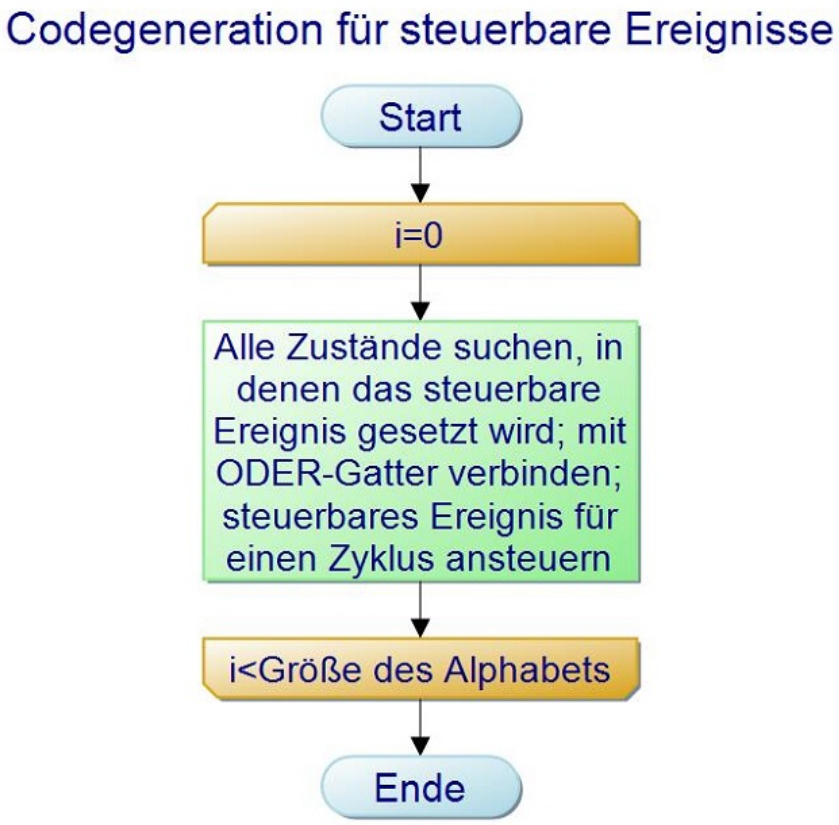


Abbildung 5.4: steuerbare Ereignisse ansteuern

Abbildung 5.5: Programmablaufplan der Funktion createHasdemir()

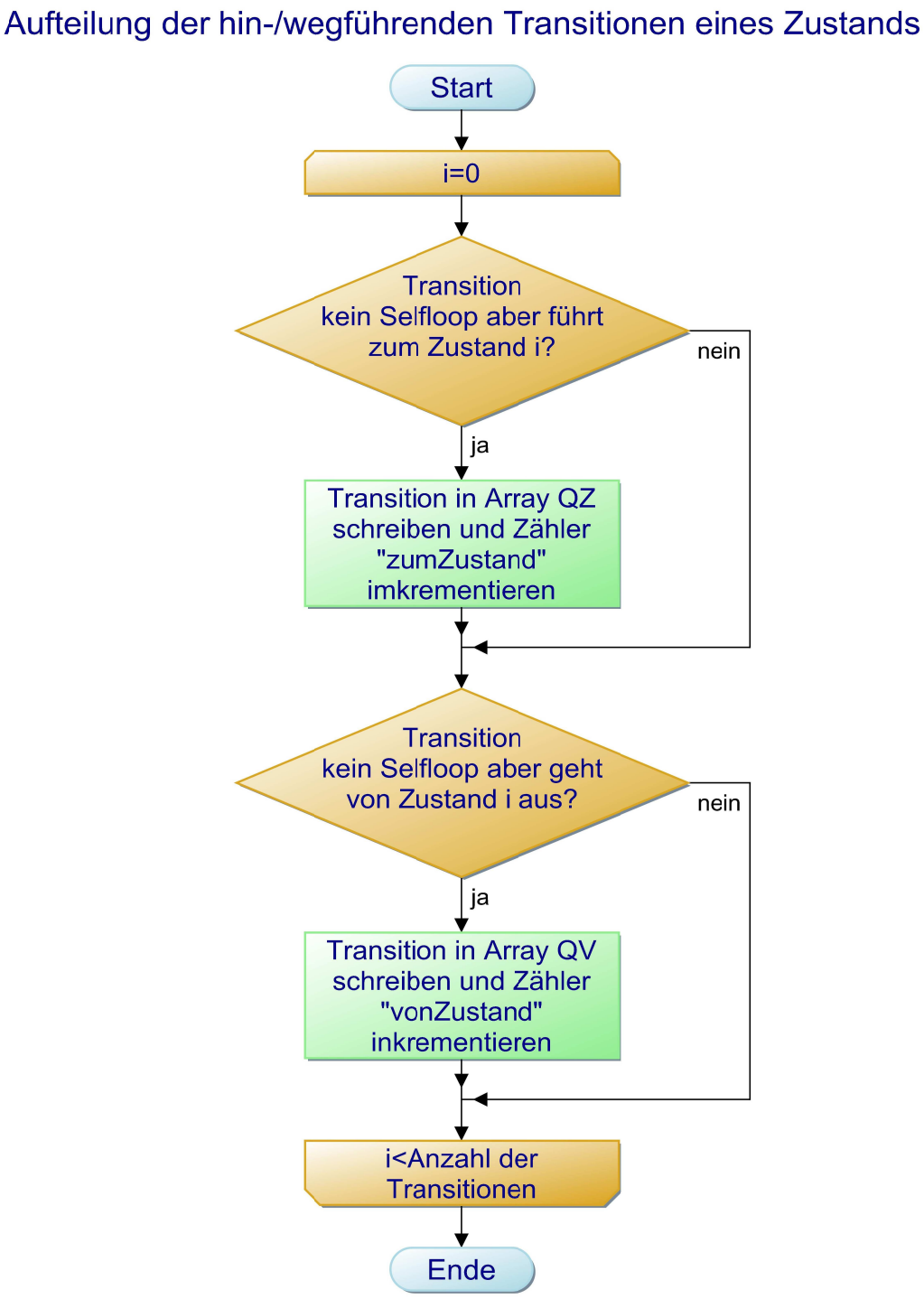


Abbildung 5.6: Aufteilung der Transitionen

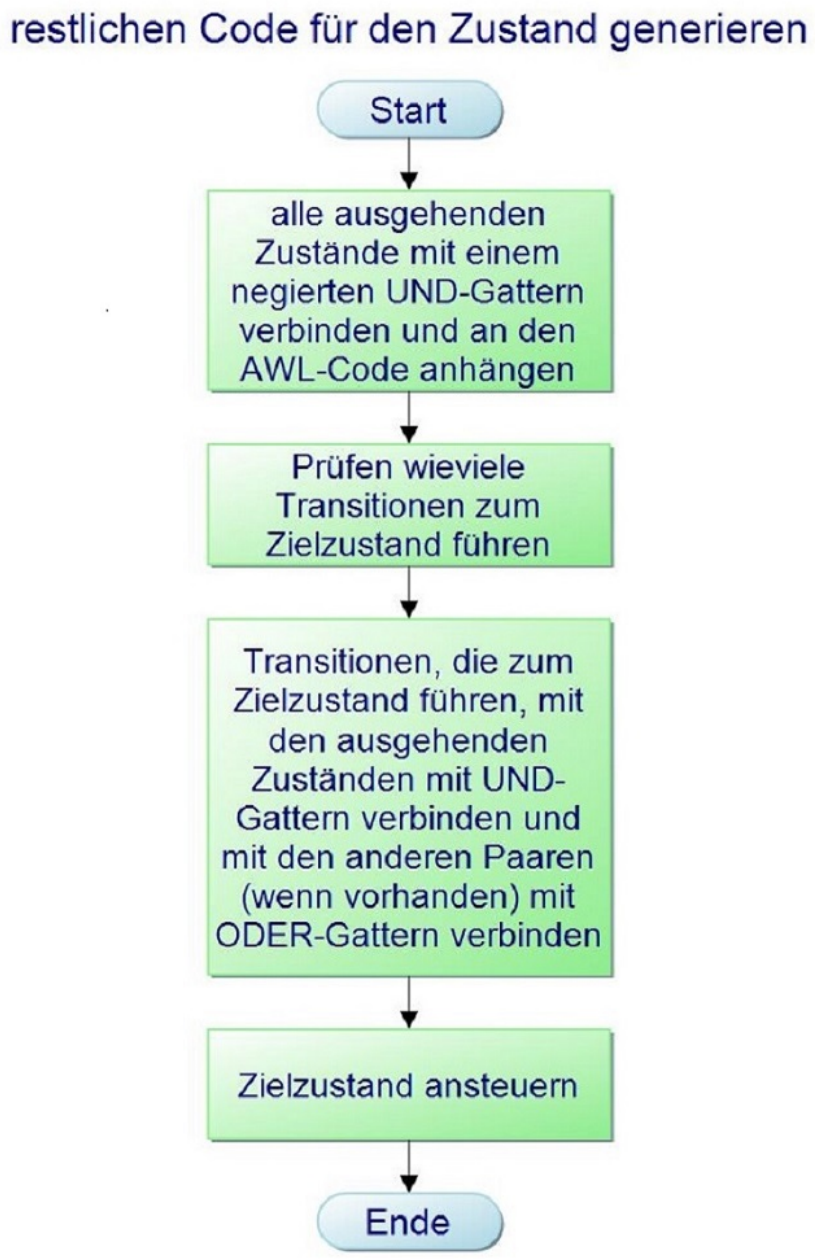


Abbildung 5.7: weiteren Code für den i-ten Zustand generieren

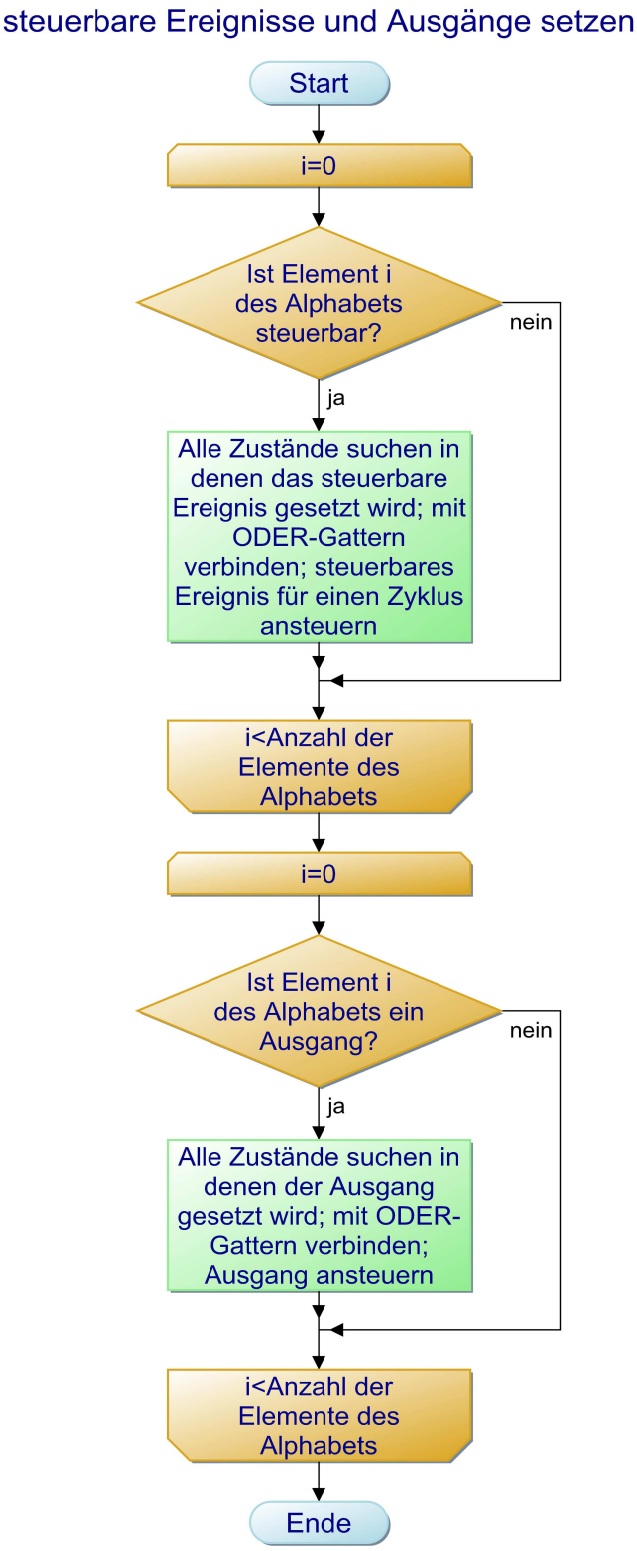
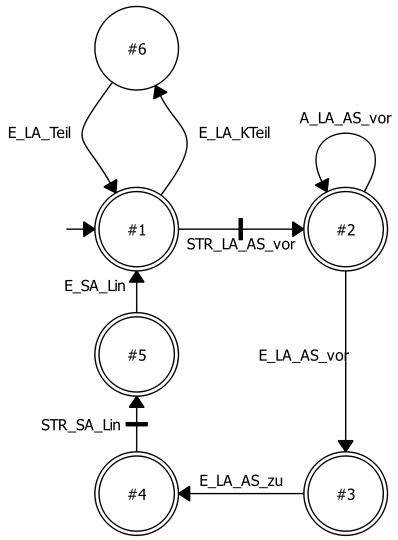


Abbildung 5.8: steuerbare Ereignisse und Ausgänge

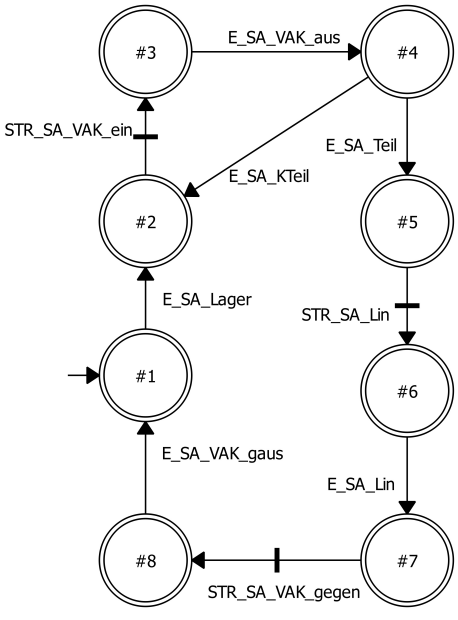
Abbildung 5.9: Synchronisation der Zustandsmerker




```
//Code generated by DESIEC 3.06
//Code fuer Zustand 2:
U K3_Q1
UN block_K3
U STR_LA_AS_vor
R K3_Q1
S K3_A_LA_AS_vor
S K3_Q2
S block_K3

//Code fuer Zustand 6:
U K3_Q1
UN block_K3
U E_LA_KTeil
R K3_Q1
S K3_Q6
S block_K3

//Code fuer Zustand 3:
U K3_Q2
UN block_K3
U E_LA_AS_vor
R K3_Q2
R K3_A_LA_AS_vor
S K3_Q3
S block_K3

//Code fuer Zustand 4:
U K3_Q3
UN block_K3
U E_LA_AS_zu
R K3_Q3
S K3_Q4
S block_K3
```

```
//Code fuer Zustand 5:
U K3_Q4
UN block_K3
U STR_SA_Lin
R K3_Q4
S K3_Q5
S block_K3

//Code fuer Zustand 1:
U K3_Q5
UN block_K3
U E_SA_Lin
R K3_Q5
S K3_Q1
S block_K3

//Code fuer Zustand 1:
U K3_Q6
UN block_K3
U E_LA_Teil
R K3_Q6
S K3_Q1
S block_K3


//Steuerbare Ereignisse setzen
U K3_Q1
= STR_LA_AS_vor_K3

U K3_Q4
= STR_SA_Lin_K3
```

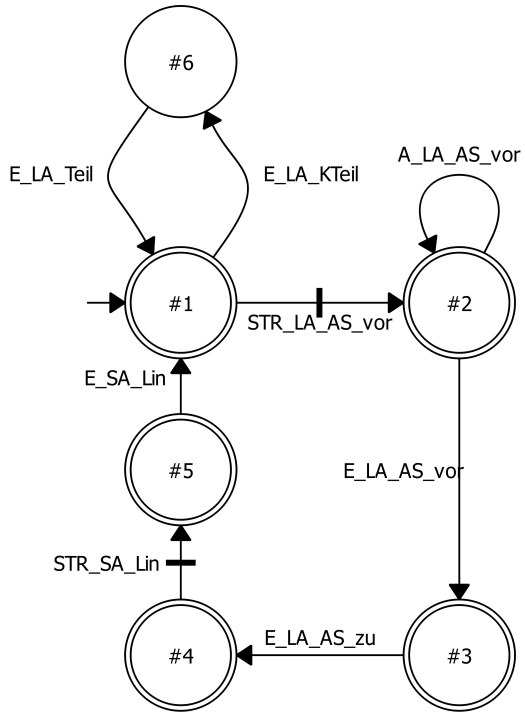
Abbildung 5.10: Spezifikation K3 und der entsprechende nach Liu et. al generierte AWL-Code




```
//Code generated by DESIEC 3.06
//Code fuer Zustand 2:
U K5_Q1
UN block_K5
U E_SA_Lager
R K5_Q1
S K5_Q2
S block_K5

//Code fuer Zustand 3:
U K5_Q2
UN block_K5
U STR_SA_VAK_ein
R K5_Q2
S K5_Q3
S block_K5

//Code fuer Zustand 4:
U K5_Q3
UN block_K5
U E_SA_VAK_aus
R K5_Q3
S K5_Q4
S block_K5

//Code fuer Zustand 5:
U K5_Q4
UN block_K5
U E_SA_Teil
R K5_Q4
S K5_Q5
S block_K5

//Code fuer Zustand 2:
U K5_Q4
UN block_K5
U E_SA_KTeil
R K5_Q4
S K5_Q2
S block_K5
```

```
//Code fuer Zustand 6:
U K5_Q5
UN block_K5
U STR_SA_Lin
R K5_Q5
S K5_Q6
S block_K5

//Code fuer Zustand 7:
U K5_Q6
UN block_K5
U E_SA_Lin
R K5_Q6
S K5_Q7
S block_K5

//Code fuer Zustand 8:
U K5_Q7
UN block_K5
U STR_SA_VAK_gegen
R K5_Q7
S K5_Q8
S block_K5

//Code fuer Zustand 1:
U K5_Q8
UN block_K5
U E_SA_VAK_gaus
R K5_Q8
S K5_Q1
S block_K5


//Steuerbare Ereignisse setzen
U K5_Q5
= STR_SA_Lin_K5

U K5_Q2
= STR_SA_VAK_ein_K5

U K5_Q7
= STR_SA_VAK_gegen_K5
```

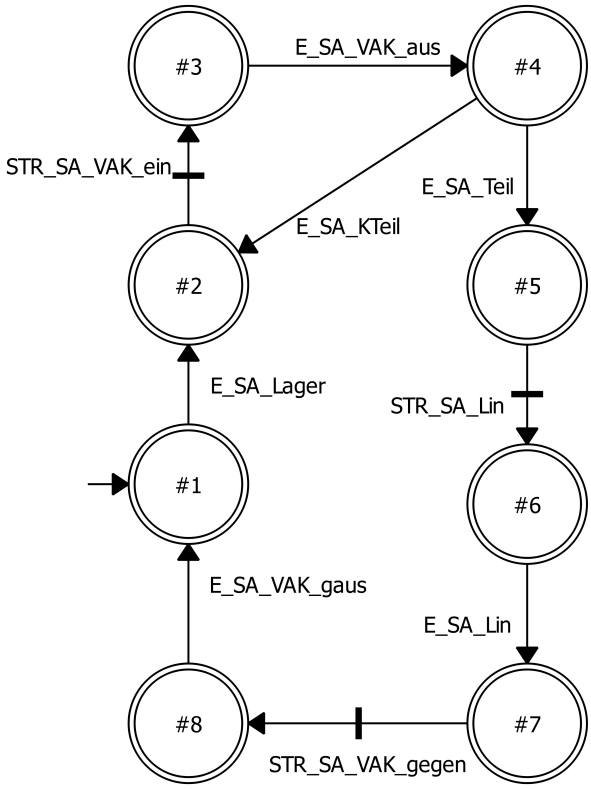
Abbildung 5.11: Spezifikation K5 und der entsprechende nach Liu et. al generierte AWL-Code




```
//Code generated by DESIEC 3.04
//Code fuer Zustand 1:
U K3_q1
UN block_K3
UN STR_LA_AS_vor
UN E_LA_KTeil
O
U K3_q5
U E_SA_Lin
O
U K3_q6
U E_LA_Teil
= K3_Q1

//Code fuer Zustand 2:
U K3_q2
UN block_K3
UN E_LA_AS_vor
O
U K3_q1
U STR_LA_AS_vor
= K3_Q2

//Code fuer Zustand 3:
U K3_q3
UN block_K3
UN E_LA_AS_zu
O
U K3_q2
U E_LA_AS_vor
= K3_Q3

//Code fuer Zustand 4:
U K3_q4
UN block_K3
UN STR_SA_Lin
O
U K3_q3
U E_LA_AS_zu
= K3_Q4
```

```
//Code fuer Zustand 5:
U K3_q5
UN block_K3
UN E_SA_Lin
O
U K3_q4
U STR_SA_Lin
= K3_Q5

//Code fuer Zustand 6:
U K3_q6
UN block_K3
UN E_LA_Teil
O
U K3_q1
U E_LA_KTeil
= K3_Q6


//Steuerbare Ereignisse setzen
U K3_Q1
= STR_LA_AS_vor_K3

U K3_Q4
= STR_SA_Lin_K3


//Ausgaenge der Zustaende setzen...
U K3_Q2
= A_LA_AS_vor

//Selbsthaltung der Zustaende aktivieren
U K3_Q1
= K3_q1
U K3_Q2
= K3_q2
U K3_Q3
= K3_q3
U K3_Q4
= K3_q4
U K3_Q5
= K3_q5
U K3_Q6
= K3_q6
```

Abbildung 5.12: Spezifikation K3 und der entsprechend nach Hasdemir et. al generierte AWL-Code




```
//Code generated by DESIEC 3.04
//Code fuer Zustand 1:
U K5_q1
UN block_K5
UN E_SA_Lager
O
U K5_q8
U E_SA_VAK_gaus
= K5_Q1

//Code fuer Zustand 2:
U K5_q2
UN block_K5
UN STR_SA_VAK_ein
O
U K5_q1
U E_SA_Lager
O
U K5_q4
U E_SA_KTeil
= K5_Q2

//Code fuer Zustand 3:
U K5_q3
UN block_K5
UN E_SA_VAK_aus
O
U K5_q2
U STR_SA_VAK_ein
= K5_Q3

//Code fuer Zustand 4:
U K5_q4
UN block_K5
UN E_SA_Teil
UN E_SA_KTeil
O
U K5_q3
U E_SA_VAK_aus
= K5_Q4

//Code fuer Zustand 5:
U K5_q5
UN block_K5
UN STR_SA_Lin
O
U K5_q4
U E_SA_Teil
= K5_Q5
```

```
//Code fuer Zustand 6:
U K5_q6
UN block_K5
UN E_SA_Lin
O
U K5_q5
U STR_SA_Lin
= K5_Q6

//Code fuer Zustand 7:
U K5_q7
UN block_K5
UN STR_SA_VAK_gegen
O
U K5_q6
U E_SA_Lin
= K5_Q7

//Code fuer Zustand 8:
U K5_q8
UN block_K5
UN E_SA_VAK_gaus
O
U K5_q7
U STR_SA_VAK_gegen
= K5_Q8


//Steuerbare Ereignisse setzen
U K5_Q5
= STR_SA_Lin_K5

U K5_Q2
= STR_SA_VAK_ein_K5

U K5_Q7
= STR_SA_VAK_gegen_K5


//Ausgaenge der Zustaende setzen...
//Selbsthaltung der Zustaende aktivieren
U K5_Q1
= K5_q1
U K5_Q2
= K5_q2
U K5_Q3
= K5_q3
U K5_Q4
= K5_q4
U K5_Q5
= K5_q5
U K5_Q6
= K5_q6
U K5_Q7
= K5_q7
U K5_Q8
= K5_q8
```

Abbildung 5.13: Spezifikation K5 und der entsprechende nach Hasdemir et. al generierte AWL-Code

Kapitel 6

Evaluierung des AWL-Codes

In diesem Kapitel erfolgt die Bewertung der generierten AWL-Codes der verschiedenen Ansätze. In Abschnitt 6.1 werden Kriterien festgelegt, anhand derer eine Evaluierung des generierten AWL-Codes durchgeführt wird. Abschnitt 6.2 beinhaltet eine Gegenüberstellung, in der der AWL-Code nach Methode von [HKG07] bzw. von [JD02] mit dem Code nach dem Ansatz von [UGD09] verglichen wird. Der letzte Abschnitt bewertet die Gegenüberstellungen und gibt ein Fazit.

6.1 Festlegen der Kriterien

Der Aufbau und die Struktur eines Programmes hängt in der Regel sehr stark vom Programmierer ab, da die persönlichen Ideen und Denkweisen in die Programmierung einfließen. Außerdem sind AWL-Programme, inbesondere bei komplexeren Anlagen und Projekten, sehr unübersichtlich und schlecht zu warten ist. Durch die automatische Generierung eines AWL-Codes werden viele negative Aspekte eines AWL-Programmes , die während der individuellen Programmierung auftreten können, eingeschränkt.
Ein AWL-Programm sollte demnach eine übersichtliche Struktur aufweisen und die Größe bzw. die Anzahl der Netzwerke auf ein Minimum reduzieren. Dies kann auch eine Speicherreduzierung zur Folge haben und dadurch kann eventuell eine kleinere CPU verwendet werden. Ein wichtiges Kriterium ist auch die Zykluszeit. Je kürzer die Zykluszeit desto schneller wird auf Änderungen an den Eingangssignalen reagiert. Eine weitere gute Eigenschaft ist eine einheitliche Nomenklatur der Signale, welche jedoch weiterhin vom Programmierer, der die Genertormodelle gestaltet, abhängt . In den Tabellen 6.1 und 6.2 sind Vergleichsmatrizen der Spezifikaionen K3 und K5 zu sehen, in der die Kriterien gewichtet werden. Für alle drei Ansätze ([HKG07, JD02, UGD09]) wird eine Bewertung durchgeführt.

Für eine realistische Bewertung werden die Kriterien gewichtet. Der Faktor fünf steht für

Tabelle 6.1: Vergleichsmatrix der Ansätze

Spezifikation K3		**Uzam et. al**	**Hasdemir et. al**	**Liu et. al**
Vergleichskriterium	**Faktor**	**Bewertung**	**Bewertung**	**Bewertung**
Struktur	5	3	4	3
Netzwerkgröße im Durchschnitt	2	4 (4,8 Zeilen)	3 (6,4 Zeilen)	4 (4,4 Zeilen)
Netzwerkanzahl	3	3 (10 NW)	3 (10 NW)	4 (9 NW)
Codezeilen	2	4 (48 Zeilen)	2 (64 Zeilen)	4 (48 Zeilen)
Zykluszeit (CPU312)	5	4 (17,9 μs)	2 (49,5μs)	5 (17,8 μs)
Arbeitsspeicherbedarf	3	4 (230 Bytes)	3 (280 Bytes)	4 (230 Bytes)
Gesamt		72	58	80

Tabelle 6.2: Vergleichsmatrix der Ansätze

Spezifikation K5		**Uzam et. al**	**Hasdemir et. al**	**Liu et. al**
Vergleichskriterium	**Faktor**	**Bewertung**	**Bewertung**	**Bewertung**
Struktur	5	3	4	3
Netzwerkgröße im Durchschnitt	2	4 (5 Zeilen)	3 (6,8 Zeilen)	4 (5 Zeilen)
Netzwerkanzahl	3	3 (12 NW)	3 (12 NW)	3 (12 NW)
Codezeilem	2	4 (60 Zeilen)	3 (82 Zeilen)	4 (60 Zeilen)
Zykluszeit (CPU312)	4	5 (22,5 μs)	3 (64,4 μs)	5 (22,5 μs)
Arbeitsspeicherbedarf	3	4 (278 Bytes)	3 (348 Bytes)	4 (278 Bytes)
Gesamt		72	62	72

„hohe Priorität” und reicht bis 1 für „niedrige Priorität”. Die Bewertung erfolgt durch 5 für „sehr gut” bis 1 für „sehr schlecht”. Hat ein Programm eine kurze Zykluszeit, wird die Performance erhöht. Die Anzahl der Codezeilen und deren Inhalt hat Einfluss auf die Zykluszeit. Eine geringe Anzahl von Codezeilen erhöht auch die Übersicht eines SPS-Programms und erleichtert eine Fehlersuche während des Betriebs. In dem Ansatz nach Hasdemir et. al wird eine Reduzierung des Speicherbedarfs angepriesen.
Die Netzwerkanzahl und Netzwerkgröße beeinträchtigen die Übersichtlichkeit eines AWL-Codes. Viele kleine Netzwerke sind zwar einzeln gut zu analysieren, gestalten das gesamte SPS-Programm jedoch sehr unübersichtlich. Es ist zweckmäßig ein Gleichgewicht zwischen der Anzahl der Netzwerke und der Größe der Netzwerke zu finden. Die einzelnen Netzwerke sollten nicht zu groß sein, damit diese noch gut zu analysieren sind. Sie sollten aber auch nicht zu klein sein, um die Gesamtanzahl der Netzwerke zu begrenzen. Speziell bei großen Anlagen sollte dieses Kriterium noch stärker gewichtet sein.
Die Struktur eines SPS-Programmes ist wichtig für die Übersichtlichkeit. Es ist besser An-

lagenteile oder Signale zusammen zu verarbeiten, damit eine nachträgliche Fehlersuche erleichtert wird, damit man einfacher feststellt, wo das gewünschte Signal zu finden ist.

6.2 Gegenüberstellung

Aus den Vergleichsmatrizen in 6.1 und 6.2 ist zu erkennen, dass die Netzwerkanzahl und die Netzwerkgröße für den Ansatz nach Uzam et. al [UGD09], nach Liu et. al [JD02] und nach Hasdemir et. al [HKG07] fast identisch sind. Auch die steuerbaren Ereignisse werden in allen Varianten am Ende in Netzwerken bearbeitet. Allein die Ausgänge werden bei der Methode nach Liu et. al direkt in dem Netzwerk gesetzt, das den dazugehörigen Zustand behandelt.
Bei den Varianten von Uzam et al. und Hasdemir et. al werden die Ausgänge in seperaten Netzwerken angesteuert. Somit kann mit einer Codegenerierung nach Liu et. al eine Reduzierung der Netzwerkanzahl erreicht werden.
Ein AWL-Code, der nach der Variante von Hasdemir et. al [HKG07] erstellt wird, weist zwar die gleiche Menge an Netzwerken auf, jedoch ist deren Größe dabei gestiegen. Dieser Aspekt muss allerdings nicht nur negativ gesehen werden, denn in den Netzwerken sind alle Ereignisse behandelt, die mit dem Zustand in Berührung kommen. Der Anstieg der Zeilen ist auch darauf zurückzuführen, dass in diesem Ansatz keine „set"- und „reset"-Operatoren eingesetzt werden. Außerdem können die Codezeilen für die Selbsthaltung ignoriert werden, da diese nicht zum eigentlichen Programmablauf gehören.
Die Struktur aller Ansätze ist bis auf die Behandlung der Ausgänge gleich. Durch die Berücksichtigung aller Signale, die vom Zustand weg führen bzw. zum Zustand hin führen, ist mit dem Ansatz nach Hasdemir eine Fehlersuche leichter als bei den anderen beiden Varianten. Die Vergleichsmatrizen zeigen aber, dass die versprochene Speicherreduzierung nach dem Ansatz von Hasdemir et. al aber nicht eintritt. Bei dem AWL-Code ohne „set"- und „reset"-Operatoren ist sogar ein höherer Speicherbedarf als bei den anderen Ansätzen zu erkennen. Dies ist auch bei der Zykluszeit der Fall. Sie liegt bei der Variante nach Hasdemir et. al wesentlich höher als bei Liu et. al und Uzam et. al.

6.3 Evaluierung und Fazit

Durch die Gegenüberstellung der drei Ansätze zur Codegenerierung wird deutlich, dass sich die versprochene Speicherreduzierung bei der Variante von Hasdemir et. al nicht einstellt. Es ist sogar eine deutliche Steigerung des Speicherbedarfs und der Zykluszeit zu erkennen. Tabelle 6.2 zeigt, dass die Ansätze von Uzam et. al und Liu et. al gleich viele Punkte bekommen, nur der Ansatz von Hasdemir et. al liegt dicht dahinter. Allein beim Kriterium „Struktur" liegt der AWL-Code von Hasdemir et. al vor den anderen Varianten, da eine leichtere Fehler-

suche möglich ist. Bei allen anderen Kriterien ist der Code von Hasdemir et. al nur maximal gleich gut. Wenn man die Tabelle 6.1 hinzuzieht wird deutlich, dass die Zykluszeit des Ansatzes von Liu et. al kürzer ist, als die von Uzam et. al und kürzer, als die von Hasdemir et. al. Der Speicherbedarf bleibt jedoch im Vergleich zu Uzam et. al derselbe. Die geringere Zykluszeit ist auf die unterschiedliche Bearbeitung der Ausgänge zurückzuführen. Im Gegensatz zum Ansatz nach Uzam et. al bzw. Hasdemir et. al werden die Ausgänge bei Liu et. al in den Netzwerken der jeweiligen Zustände gesetzt bzw. rückgesetzt. Dies spart pro Ausgang ein logisches „und", was einer Reduzierung der Zykluszeit um bis zu $4\mu s$ entsprechen kann. Abschließend ist zu sagen, dass die Ansätze von Uzam et. al und Liu et al. sehr ähnlich sind und sich nur in der Zykluszeit unterscheiden. Dies kann jedoch gerade bei großen Anlagen mit vielen Ausgängen einen erheblichen Unterschied ausmachen. Der Aufbau und die Anzahl der Netzwerke sind fast identisch. Bei Spezifikationen mit Ausgängen reduziert sich die Anzahl der Netzwerke bei der Variante von Liu et. al. Der Ansatz von Hasdemir et. al ist nicht zu empfehlen, da er in fast jeder Kategorie der schlechtere Ansatz ist. Selbst die im Artikel angeführte Speicherreduzierung wird nicht eingehalten. Der Ansatz nach Liu et. al weist so gut wie keinen Unterschied zu dem von Uzam et. al auf. Jedoch hat die Modellierung des „extended" Supervisors den Vorteil, dass zu jedem Zustand sofort ersichtlich ist, welche Ausgänge aktiv sind und welche nicht. Es ist noch zu erwähnen, dass bei Spezifikationen, in denen mehrere Ausgänge in einem Zustand angesteuert werden, die Netzwerkgröße nach Liu et.al stark steigt. Die Behandlung der Ausgänge direkt im Netzwerk des dazugehörigen Zustands ist in diesem Fall ungünstiger.

Kapitel 7

Zusammenfassung und Ausblick

Zusammenfassung

Das Thema der vorliegenden Arbeit war die Bewertung unterschiedlicher Ansätze zur automatischen Codegenerierung bei ereignisdiskreten Steuerungen. Grundlage war die Arbeit von [Url12], in der ein Softwaretool entwickelt wurde, das aus einem Generatormodell einen AWL-Code erzeugt. Der verwendete Ansatz in der Arbeit von Urland war der von Uzam et. al.

Es war Aufgabe zu evaluieren, inwiefern andere Ansätze zur Codegenerierung einen besseren AWL-Code zum Ergebnis haben.

Es wurden die Ansätze nach [HKG07] und [JD02] zur Untersuchung gewählt. Die Variante nach Hasdemir et. al verspricht eine Speicherreduzierung, durch eine Nichtbenutzung von „set"- und „reset"-Operatoren. Der Ansatz von Liu et. al bietet die Erstellung eines Programmes, das modular in seiner Struktur und gut zu modifizieren ist. Die Modelle, die für den Vergleich der AWL-Codes verwendet wurden, sind der Arbeit von Urland entnommen.

Nach Analyse der Ansätze wurde das Programm „DES2IEC" neu gestaltet. Es wurde eine Auswahl installiert, die den User entscheiden lässt, nach welchem Ansatz er einen AWL-Code generieren möchte. Anschließend wurden Unterfunktionen entworfen, die aus einer im Vorfeld mit DESTool modellierten Spezifikation einen AWL-Code nach der gewünschten Methode erstellen.

Für die Bewertung der Ansätze wurden Kriterien festgelegt und jeweils mit einem Faktor zwecks Wertigkeit belegt. Nach einer Gegenüberstellung der generierten AWL-Codes für jeweils zwei ausgewählte Spezifikationen, wurden die Kriterien Arbeitsspeicherbedarf, Zykluszeit, Netzwerkanzahl, Netzwerkgröße, Codezeilen und Struktur in einer Tabelle für beide Spezifikationen bewertet.

Daraus wurde ersichtlich, dass der Ansatz nach Hasdemir seine versprochene Speicherreuzierung nicht einhält. Einzig das Kriterium der Struktur wird bei diesem Ansatz positiver bewertet als bei den anderen. Die Struktur wird jedoch je nach Benutzer sehr individuell bewertet.

Der Ansatz nach Liu et. al weist eine große Übereinstimmung der generierten Codes bezüglich des Ansatzes nach Uzam et. al auf, besitzt aber bei Spezifikationen mit Ausgängen eine kürzere Zykluszeit.

Ausblick

Die vorliegende Arbeit zeigt, dass es durchaus sinnvoll ist, unterschiedliche Ansätze zur Codegenerierung zu evaluieren. Eine Optimierung des Speicherbedarfs bzw. eine weitere Reduktion der Zykluszeit wären ein guter Fortschritt. Neue Ansätze in das bestehende Programm einzugliedern ist nicht all zu schwierig, man muss jedoch für die Modellierung der Spezifikationen die DESTool Version 0.59 verwenden.
Der Aufbau der Projektdatei von DESTool ändert sich bei neueren Versionen. Dies hat ein inkorrektes Einlesen der Daten zur Folge. Es besteht jedoch die Möglichkeit, Projekte aus DESTool als „*.faudes" Dateien zu exportieren. Der Aufbau dieser Dateien ändert sich bei neuen Versionen laut Hersteller nicht. Ein Programm, das diese Dateien als Grundlage für eine automatische Codegenerierung nutzt, könnte Inhalt einer weiteren Arbeit sein.
Es gibt noch eine Vielzahl an Ansätzen, um einen Supervisor (modelliert als Generator oder als Petrinetz) in eine Programmiersprache für eine SPS zu konvertieren. Eine weiterführende Arbeit könnte sich mit der Evaluierung weiterer Themen, vielleicht auf Basis der Petrinetze, konzentrieren.

Anhang A

Verzeichnis über den Anhang auf der DVD

Der Anhang dieser Thesis befindet sich auf der eingeklebten DVD, die bei Herrn Prof. Wenck eingesehen werden kann.

A.1 Programm und Quellcode

Der Ordner „Programm und Quellcode” beinhaltet eine modifizierte Version von „DES2IEC” in Form der ausführbaren Datei „DES2IEC_M_306”, sowie den Ordner „DES2IEC_Multi”, der den Microsoft Visual Studio 2012 Workspace und alle Quellcodes des Programmes enthält.

A.2 DESTool

Der Ordner „DESTool” enthält das DESTool Projekt (Version 0.66) aus der vorangegangenen Arbeit sowie alle zu Testzwecken erstellten DESTool Projekte der einzelnen Spezifikationen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

δ	Zustandsübergangsfunktion
Σ	Ereignisalphabet
Σ^*	Kleene-Hülle
Σ^+	Ereignisalphabet ohne leeres Wort
Σ_{CE}	Gemeinsame Ereignisse
Σ_c	Menge der steuerbaren Ereignisse
Σ_f	Endereignisse
Σ_{PE}	Private Ereignisse
Σ_{sm}	Systemereignisse
Σ_s	Startereignisse
Σ_{uc}	Menge der nicht steuerbaren Ereignisse
ε	leeres Zeichen/Wort
$\varnothing$	leere Menge
A	Adjazenzmatrix
A	Ausgang der SPS
$AC(G)$	Erreichbarer Teil des Generators G
AR	Activity-Ressource Vektor
AWL	Anweisungsliste
$CoAc(G)$	Ko-erreichbarer Teil des Generators G

$DES2IEC$	Software zur Generierung eines SPS Programmes in AWL
DFA	deterministic finite automata
E	Eingang der SPS
G	Generator
G_x	Produktautomat
$HISC$	Hierarchical Interface-Based Supervisory Control
HU	Handling Unit
I	Input (logisches Eingangssignal)
$IDEF$	Integrated Definition
K	formale Spezifikation
$K^{\uparrow c}$	supremale steuerbare Teilsprache
$L_m(G)$	markierendes Verhalten von G
LA	Lagereinheit
LA_AS	Ausschieber der Lagereinheit
LI	Linearachse (Transporteinheit)
LS	Lichtschranke
O^L	latch Output (logischer Ausgang der verriegelt)
O^U	unlatch Output (logischer Ausgang der entriegelt)
P	Natürliche Projektion
P^{-1}	inverse natürliche Projektion
R	Vertikal-Roboter
R_i	reset-Operator (Rücksetzen eines Bits) für den i-ten Zustand
RW	Ramadge und Wonham
S	Supervisor

S/G	Gesteuertes System (S steuert G)
S_i	set-Operator (Setzen eines Bits) für den i-ten Zustand
SA	Schwenkarm
SCT	Supervisory Control Theory
SPC	Strict Product Composition
SPS	Speicherprogrammierbare Steuerung
SR	State-Ressource Vektor
$SYPC$	Synchrones Produkt
$Trim(G)$	Erreichbarer und ko-erreichbarer Teil des Generators G
WS	Werkstück
X	Zustandsmenge
x'	Folgezustand
x_0	Anfangszustand
X_m	Menge der markierten Zustände
DES2IEC	Programm zum Generieren eines AWL-Codes
L(G)	Reguläre Sprache von G
s	(String) Sequenz von Symbolen
STR	Steuerbares Ereignis der SPS

Literaturverzeichnis

[Bra96] Bertil A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Transactions on Robotics and Automation, Vol. 12, No. 1*, 1996.

[CL99] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[HFL01] I.T. Hellgren, M. Fabian, and B. Lennartson. Modular implementation of discrete-event systems as sequential function charts applied to an assembly cell. *In: Proceedings of the 2001 IEEE International Conference on Control Applications*, 2001.

[HKG07] T. Hasdemir, S. Kurtulan, and L. Gören. An implementation methodology for supervisory control theory. *Springer-Verlag London Limited*, 2007.

[JD02] Liu Jing and Houshang Darabi. Ladder logic implementation of ramadge-wonham supervisory controller. *Sixth International Workshop on Discrete Event Systems*, 2002.

[Led05a] R.J. Leduc. Hierarchical interface-based supervisory control - part i : Serial case. *In: Proceedings of the IEEE Transactions on Automatic Control 50*, 2005.

[Led05b] R.J. Leduc. Hierarchical interface-based supervisory control - part ii : Parallel case. *In: Proceedings of the IEEE Transactions on Automatic Control 50*, 2005.

[Led06] R.J. Leduc. Hierarchical interface-based supervisory control of a flexible manufacturing system. *IEE Proceedings Transactions on Control Systems Technology 14*, 2006.

[Lun12] Jan Lunze. *Ereignisdiskrete Systeme*. Oldenbourg Verlag, 2012.

[QC00] M.H. Queiroz and J.R.E. Cury. *Modular Control of Large Scale Discrete- Event Systems*. R. Boel, 2000.

[QC02] M.H. Queiroz and J.R.E. Cury. Synthesis and implementation of local modular supervisory control for a manufacturing cell. *In: Sixth International Workshop on Discrete- Event Systems*, 2002.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization 25*, Januar Nr. 1:206–230, 1987.

[Sch95] U. Schöning. *Theoretische Informatik - kurzgefasst. 2. überarbeitete Auflage Spektrum*. Akad.-Verlag, 1995.

[SIE] SIEMENS. Rfid systeme - simatic rf300. *Systemhandbuch*.

[SIE08] SIEMENS. Operationsliste s7-300. *Dokumentation für 6ES7398-8FA10-8AA0*, 6:31–54, 2008.

[SUP] SUPREMICA. Supremica: Software for formal verification and synthesis for control systems. *Dep. of Signals and Systems, Chalmers University of Technology, Schweden*, www.supremica.org.

[UGD09] Murat Uzam, Gökhan Gelen, and Recep Dalci. A new approach for the ladder logic implementation of ramadge-wonham supervisors. *In: XXII International Symposium on Information, Communication and Automation Technologies*, 2009.

[Url12] Christian Urland. Ereignisdiskrete steuerungssynthese und codegenerierung am beispiel einer fertigungszelle. Master's thesis, Hochschule für angewandte Wissenschaften Hamburg, 2012.

[Wen06] Florian Wenck. *Modellbildung, Analyse und Steuerungsentwurf für gekoppelte ereignisdiskrete Systeme*. Shaker, 2006.

[Won04] W.M. Wonham. Supervisory control of discrete- event systems. Master's thesis, Dep. of Electrical and Computer Engineering University of Toronto, Kanada, 2004.

[WR88] W.M. Wonham and P.J. Ramadge. Modular supervisory control of discrete- event systems. *In: Mathematics of Control, Signals and Systems*, 1988.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5)APSO-TI-BM ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 28.07.2013

Ort, Datum

Unterschrift