Modellierung und Analyse von SPS-Anwenderprogrammen mit Petri-Netzen

Monika Heiner, Thomas Menzel
Institut für Informatik
Brandenburgische Technische Universität Cottbus
Postfach 101344
D-03013 Cottbus
Tel.: +49 (0)355 69-3885
Fax.: +49 (0)355 69-3830
E-Mail: {mh,thm}@informatik.tu-cottbus.de

Abstrakt: In diesem Bericht beschreiben wir eine Möglichkeit zur Modellierung und Analyse von SPS-Systemen. Dazu stellen wir zunächst kurz eine Petri-Netz-Semantik für die SPS-Anwenderprogrammiersprache Anweisungsliste (AWL) vor, welche in der IEC 1131-3 [DIN EN 61131-3] definiert wurde. Anschließend diskutieren wir wesentliche Punkte einer Methodik zur Erstellung eines Systemmodelles, wobei sowohl der Entwurf des notwendigen Umgebungsmodelles als auch drei verschiedene Möglichkeiten für das Systemprogramm angesprochen werden. Zum Abschluß wird die Generierung des Petri-Netz-Modelles aus einem gegebenen SPS-System anhand eines Beispiels praktisch illustriert.

Stichworte: Petri-Netz, Speicherprogrammierbare Steuerung, Anweisungsliste, Verifikation.

1 Einleitung

Speicherprogrammierbare Steuerungen (SPS) werden derzeit in vielen komplexen Anlagen in der Industrie eingesetzt. In diesem Zusammenhang ist es wünschenswert, daß die innerhalb einer SPS verwendeten Anwederprogramme bezüglich einer gegebenen Spezifikation validiert werden. Eine formale Validierung setzt ein formales Modell voraus. Eine Möglichkeit dafür stellen Petri-Netze dar. Grundsätzlich können hier zwei Ansätze unterschieden werden:

- Petri-Netze als Modellierungswerkzeug mit anschließender Synthese [Rausch96]), und
- Petri-Netze als Analysewerkzeug mit nachträglicher Transformation [Hanisch97], [Rausch97]).

Unsere Arbeit folgt dem zweiten Ansatz. Das Ziel besteht darin, ein existierendes Anwenderprogramm (geschrieben als Anweisungsliste) in ein Petri-Netz zu transformieren. Das resultierende Petri-Netz kann dann als formale Basis zur Analyse von funktionalen und sicherheits-orientierten Anforderungen dienen. Die Anforderungen an die SPS können unter Verwendung der temporalen Logik formuliert werden. Die Formeln dieser Logik werden dann mit Hilfe von Modell-Checkern am Petri-Netz-Modell verifiziert bzw. falsifiziert. In Abbildung 1 ist das Prinzip der Validation von SPS-Anwenderprogrammen mit Petri-Netzen schematisch dargestellt. Der dort vorgestellte Zyklus zerfällt in zwei Hauptströme:

- Generierung eines SPS-Systemmodells als Petri-Netz,
- Beschreibung der SPS-Anforderungen in Formeln der temporalen Logik.

Die Generierung eines SPS-Systemmodells als Petri-Netz kann in vier Schritte zerlegt werden:

1. Übersetzung des SPS-Anwenderprogrammes (geschrieben als Anweisungsliste) in ein Petri-

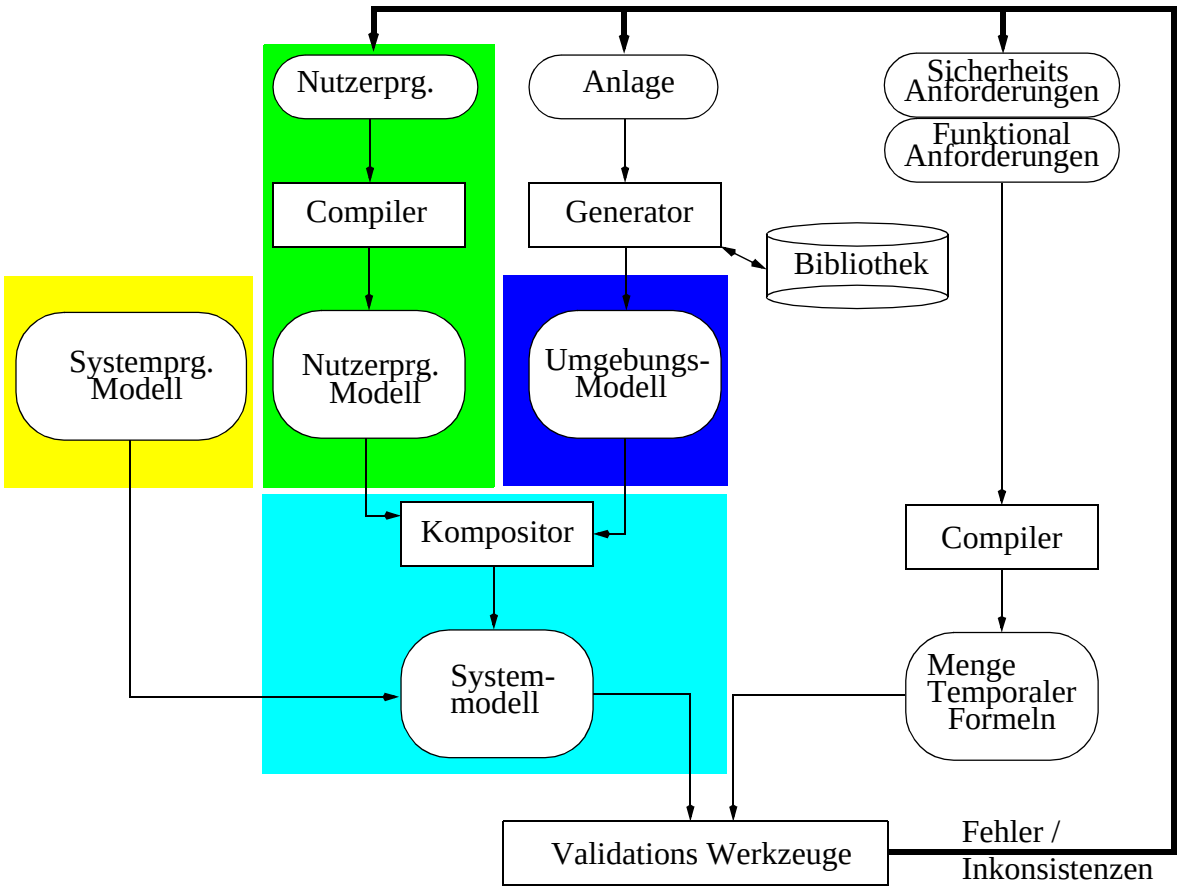


Abbildung 1:Validationszyklus eines SPS-Anwenderprogrammes mit Petri-Netzen

Netz-Modell (vgl. Kapitel 2),
2. Modellierung der ungesteuerten Strecke (Anlage ohne Steuerung) als ein Petri-Netz unter Verwendung einer geeigneten Bibliothek von Petri-Netz-Bausteinen (vgl. Kapitel 3),
3. Modellierung bzw. Wahl des Systemprogrammes (vgl. Kapitel 4),
4. Komposition des gesamten Modells aus den vorangegangenen Teilmodellen.

Schritt 3 ist in der Regel nur einmal für ein zu realisierendes Projekt auszuführen. Der zweite Schritt muß einmal für jede Konfiguration der ungesteuerten Strecke ausgeführt werden. Die Übersetzung im ersten Schritt muß für jedes SPS-Anwenderprogramm so lange ausgeführt werden, bis dies den formulierten Anforderungen genügt.

Nach der in der Abbildung 1 gezeigten Komposition ist es möglich, die gegebene Menge an Anforderungen am Modell der gesamten Anlage zu prüfen. Die Anforderungen an das System können in einer semiverbalen sicherheits-orientierten Spezifikationssprache formuliert werden. Diese Sicherheitsfachsprache (SFS) wurde in [Meier98] definiert und hat als formale Semantik die temporale Logik. Durch diese Semantik ist es möglich, Sätze der SFS in temporal-logische Formeln zu überführen, welche dann mit Hilfe von geeigneten Modell-Checkern am Systemmodell verifiziert werden können. Wir benutzen die temporalen Logiken CTL und LTL als formale Basis und die Werkzeuge PROD, PEP und eine auf BDD's basierende Eigenentwicklung [Spranger98] als Modell-Checker (Details in [Heiner98-1]).

In dieser Arbeit werden wir zunächst in Kapitel 2 eine mögliche Transformation eines AWL-Programmes in ein Petri-Netz vorstellen. Anschließend diskutieren wir in Kapitel 3 wichtige Punkte zum

Umgebungsmodell und im Kapitel 4 verschiedene Möglichkeiten der Modellierung von Systemprogrammen. In Kapitel 5 wird abschließend ein kleineres, aber realistisches Beispiel diskutiert. Die vorliegende Arbeit basiert auf [Heiner97-1] und [Heiner97-2].

2 Überführung eines Anwenderprogrammes in ein Petri-Netz

Grundlage unserer Verifikationsmethode ist die Überführung eines vorliegenden Anwenderprogrammes einer SPS in ein äquivalentes Petri-Netz-Modell. Das Anwenderprogramm sei dabei in der Sprache Anweisungsliste (AWL) geschrieben. Diese SPS-Programmiersprache ist weit verbreitet und auf dem Gebiet der SPS-Programmierung hinlänglich bekannt. Durch die Entwicklung internationaler Standards ([DIN EN 61131-3], entspricht der IEC-1131-3) ist auch eine Standardisierung dieser Sprache vorgenommen worden. AWL ist eine assembler-ähnliche Programmiersprache, in die alle anderen SPS-Anwenderprogrammiersprachen des Standards überführt werden können. Umgekehrt kann jedoch nicht jedes AWL-Programm in jede andere Programmiersprache überführt werden. Aus diesen Gründen stellt AWL einen geeigneten allgemeinen programmiersprachlichen Ausgangspunkt für die Transformation dar.

In diesem Kapitel wird zunächst eine Teilmenge von AWL (AWL_0) eingeführt. Diese Teilmenge wird formal durch eine abstrakte Syntax, eine statische Semantik und eine operationale Semantik definiert. Durch dieses Vorgehen erhalten wir eine wohl-definierte Sprache, welche als Grundlage für eine formale Transformation geeignet ist. Die Grundidee bei dieser Transformation ist der Austausch der konventionellen operationalen Semantik gegen eine Petri-Netz-Semantik. Durch diesen Wechsel wird ein Compiler definiert, der Programme der Sprache AWL_0 in Petri-Netz-Modelle überführt. Eine schematische Darstellung dieses Vorgehens wird in Abbildung 2 gezeigt.

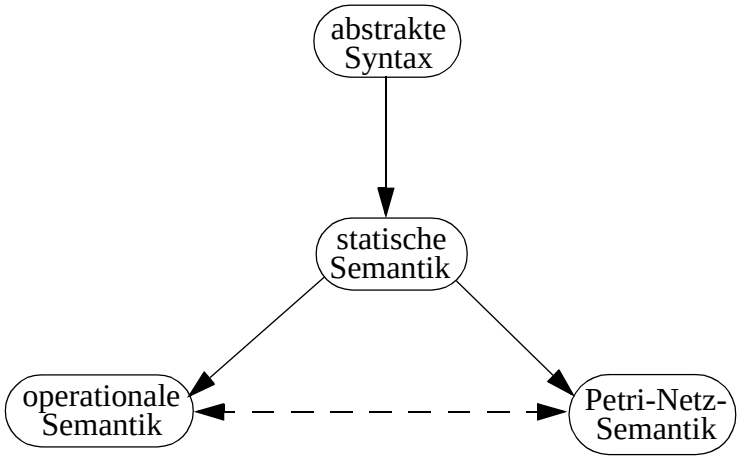


Abbildung 2:Schema der Übersetzung

2.1 AWL_0 - eine Untermenge von AWL

In diesem Abschnitt beschreiben wir eine Untermenge der SPS-Anwenderprogrammiersprache AWL, genannt AWL_0. Der Grundstock der Sprache AWL ist relativ einfach und durch wenige Befehle bestimmt, jedoch wurde die Sprache durch verschiedene zusätzliche Bibliotheken erweitert und damit recht komplex. Diese Funktionsbibliotheken stellen verschiedenste Funktionen und Funktionsbausteine zur Programmierung einer AWL zur Verfügung. Um eine übersichtliche, leicht nachvollziehbare Problemlösung zu erhalten, beschränken wir zunächst den Funktionsumfang von AWL auf einen repräsentativen (aber nicht restriktiven) Teil.

Die Programmiersprache AWL_0 weist gegenüber AWL folgende Einschränkungen auf:

- ausschließliche Verwendung von Standardbefehlen (keine zusätzlichen Kommandos aus Bibliotheken)
- Begrenzung der Datenstrukturen auf eine Länge von 8 Bit (Bool, 8-bit Worte, vorzeichenlose und vorzeichenbehaftete Short-Integer).
- Keine Befehle und Datenstrukturen für Zeit und Datum.
- Es ist nur eine Single-Prozessor- und Single-Task-Konfiguration möglich.

Im folgenden Abschnitt werden wir überblicksmäßig den Weg zur formalen Definition von AWL_0 skizzieren. Zuerst wird eine abstrakte Syntax definiert, woran sich die Definition einer statischen Semantik im Stil einer denotationellen Semantik anschließt. Durch die Semantik wird der erlaubte semantische Kontext der Sprache definiert. Abschließend geben wir als Referenzsemantik eine konventionelle operationale Semantik an.

2.1.1 Abstrakte Syntax von AWL_0

Mit Hilfe der abstrakten Syntax beschreiben wir den syntaktischen Körper von AWL_0. Eine solche abstrakte Syntax ähnelt einer konkreten, wobei nur der Grad der Details unterschiedlich ist. In einer abstrakten Syntax genügt ein Nonterminal wie <PRGNAME>, wobei dies nicht weiter spezifiziert wird, da in dem gegebenen Zusammenhang solche Details nicht interessieren (wie z. Bsp. <PRGNAME> ist eine Zeichenkette). Mit der abstrakten Syntax werden mögliche Ableitungsbäume beschrieben, deren Wurzel ein Nonterminal sind. Die von uns angegebene abstrakte Syntax wurde in EBNF (Extended Backus Nauer Form) verfaßt. Sie besteht aus 48 Regeln. Um einen konkreten Eindruck zu vermitteln, geben wir hier die Startregel der Syntax an:

$$\langle \mathrm{AWL}_0 \rangle \rightarrow \langle \mathrm{PRG} \rangle \ \{ \langle \mathrm{FB} \rangle \parallel \langle \mathrm{FKT} \rangle \}$$

Diese Regel beschreibt den Aufbau eines AWL_0-Programmes auf dem obersten Konstruktionsniveau. Ein syntaktisch korrektes ALW_0-Programm besteht aus dem sogenannten Programmblock ($\langle$ PRG $\rangle$), gefolgt von einer beliebigen Anzahl von Funktionsblöcken ($\langle$ FB $\rangle$) und Funktionen ($\langle$ FKT $\rangle$) in beliebiger Folge.

2.1.2 Statische Semantik

Aufbauend auf die im vorangegangenen Abschnitt vorgestellte abstrakte Syntax definieren wir nun die Kontextbedingungen in Form einer denotationellen Semantik. Wie schon ausgeführt, beschreibt jede Regel der Syntax einen Ableitungsbaum mit einem Nonterminal als Wurzel. Für jeden dieser Ableitungsbäume geben wir eine Funktion an, die den Ableitungsbaum als Parameter erhält. Als Rückgabewert liefert jede Funktion ein Tupel *Env* (siehe unten). Mit der statischen Semantik beschreiben wir semantische Anforderungen an die Programmiersprache AWL_0, welche:

- in Normen definiert wurden
  (z.B. muß der Akkumulatortyp und der Ergebnistyp einer Operation kompatibel sein),
- oder sinnvoll vom Standpunkt eines Informatikers sind, jedoch nicht in die Normen aufgenommen wurden
  (z.B. Sprünge aus Klammerebenen heraus sind verboten).

Die Startfunktion (Hauptfunktion) ist:

$$\begin{aligned} Parse: \ & \mathrm{AWL}_0 \rightarrow Env \\ & Parse\ (p) = NeueMarkierung\ (EinEbnen\ (\ AWL_0\ (p))) \end{aligned}$$

wobei p ein syntaktisch korrektes AWL_0-Programm ist. Die Funktion AWL_0 korrespondiert mit

der Regel, die im vorangegangenen Abschnitt erwähnt wurde.

Das Resultat der Funktion $Parse$ ist ein Tupel der Form:

$$Env = Name \times Domain \times Env_0 \times Label \times Code$$

mit:

- $Name$: der Name des gegebenen Programmes
- $Domain = \{\mathbf{prg}, \mathbf{fb}, \mathbf{fkt}\} \times \{\mathbf{n}, \mathbf{k}\}$, wobei
  prg - Programm, **fb** - Funktionsblock, **fkt** - Funktion, **n** - nicht geklammert, **k** - geklammert
- Env_0: Tupel für Variablen
- $Label$: Menge der benutzten Zeilenbeschriftungen (Sprungmarken)
- $Code = [s_0, s_1, \ldots, s_n]$ ist eine geordnete Liste des Programmcodes, wobei s_i die Programmanweisung in der Zeile i ist.

Nach der Ausführung der Funktion $Parse$ für ein gegebenes AWL_0-Programm p enthält $Code$ ein eingeebnetes AWL_0-Programm (ohne Aufrufe von Funktionsblöcken und Funktionen), d. h. alle Aufrufe von Funktionsblöcken und Funktionen werden durch die entsprechenden Anweisungen ersetzt. Desweiteren werden alle Klammerstrukturen aufgelöst, indem pro Klammerebene eine temporäre Variable für das Zwischenergebnis eingeführt wird. Alle symbolischen Sprungmarken werden eliminiert, wobei als Sprungziele die korrespondierenden Zeilennummern eingesetzt werden. Als Ergebnis der Hauptfunktion erhalten wir ggf. ein strikt sequentielles Programm, welches korrekt bezüglich der definierten Syntax und statischen Semantik ist. Dieses Ergebnis ist Voraussetzung für den nächsten Schritt.

2.1.3 Operationale Semantik

Nach der Syntax- und Kontextanalyse erhält man eine einfache Liste von Anweisungen. Jede Zeile dieser Liste kennzeichnet einen Zustand des Programmes, da das Programm, welches durch die Liste repräsentiert wird, streng sequentiell ist. Die Anweisungen des Programmes beschreiben die Übergänge zwischen den Zuständen. Diese Übergänge definieren wir in einer operationalen Semantik. Die Programmzustände (Speicherzustände) werden dabei durch das Tripel (i, m, a) beschrieben, mit

- i: Nummer der Zeile (Index eines Listenelementes aus $Code$),
- $m \in Env_0$: Speicherzustand,
- $a = (v, t)$: Akkumulator, wobei v der Wert und t der Typ sind.

Ausgangspunkt der operationalen Semantik ist eine erfolgreiche Ausführung der Funktion $e = Parse\ (p)$. Darauf aufbauend können wir für jede Anweisung von AWL_0 einen Zustandsübergang unter e definieren. An dieser Stelle einige Beispiele zur Illustration:

(1) $(i, m, a) \xrightarrow[e]{\texttt{AND}\ x} (i + 1, m, (m(x) \wedge a[1], a[2]))$

Das Ergebnis der logischen Operation `AND` zur Verknüpfung des Akkumulatorwertes mit dem Wert des Operanden wird als neuer Wert des Akkumaltors gespeichert. Der Typ des Akkumulators ändert sich nicht.

(2) $(i, m, \langle \mathbf{T}, \mathbf{bool} \rangle) \xrightarrow[e]{\texttt{JMPC}\ x} (x, m, a)$

$(i, m, \langle \mathbf{F}, \mathbf{bool} \rangle) \xrightarrow[e]{\texttt{JMPC } x} (i+1, m, a)$

Wenn der Wert des Akkumulators **TRUE** ist, dann wird mit der Abarbeitung in der Zeile mit der Nummer i fortgefahren, ansonsten gelangt die nachfolgende Zeile zur Ausführung.

(3) $(i, m, a) \xrightarrow[e]{\texttt{ADD } x} (i+1, m, (m(x) + a[1], a[2]))$

Der neue Akkumulatorwert ist das Ergebnis der Addition des bisherigen Akkumulatorwertes und des Operandenwertes. Der Typ des Akkumulators bleibt unverändert.

Zur Definition der operationalen Semantik benötigen wir keine Typüberprüfungen, da diese bereits durch die Ausführung der Funktion *Parse* erfolgt sind. In den folgenden Abschnitten werden wir eine Petri-Netz-Semantik entwickeln, welche wir dann gegen die gerade vorgestellte operationale Semantik austauschen können.

2.2 Petri-Netz Semantik

In den folgenden Abschnitten werden wir eine Semantik von AWL_0 mit Hilfe von Petri-Netzen formulieren. Diese tauschen wir dann gegen die bereits definierte operationale Semantik aus und erhalten somit einen formal definierten Compiler, der AWL_0 in Petri-Netze abbildet.

2.2.1 Syntaktische Konventionen

Die folgenden Abschnitte verwenden bei der Darstellung der Petri-Netze einige Erweiterungen. Die entworfenen Petri-Netze sind alle hierarchisch modelliert, wobei die Hierarchien an dieser Stelle nur syntaktischer Natur sind. Es wird keinerlei semantische Komposition eingesetzt. Vor einer Analyse der Petri-Netze werden sie eingeebnet. Eine Zusammenfassung der syntaktischen Erweiterungen ist in Tabelle 1 zu finden.

	*logische Plätze* -- Während des Einebnen werden alle logische Plätze mit den selben Namen zu einem Platz zusammengefaßt.
	*logische Transitionen* -- äquivalent zu logischen Plätzen, nur für Transitionen.
	*Unternetz* -- repräsentiert einen Teil des Petri-Netzes, der an dieser Stelle beim Einebnen eingefügt wird.
	*Transitions-berandetes Unternetz* -- Unternetz, dessen Randknoten (Knoten zu oder von denen eine Verbindung zum Obernetz besteht) nur Transitionen sind.
	*Platz-berandetes Unternetz* -- Die Randknoten des Unternetzes sind nur Plätze.

Tabelle 1: Syntaktische Petri-Netz-Erweiterungen

Neben den syntaktischen Erweiterungen verwenden wir eine neue Art von Kanten - die Lesekanten. Solche Lesekanten haben anstelle eines Pfeiles einen Kreis als Kopf (siehe Abbildung 3). Solche Kanten haben immer einen Platz als Ursprungsknoten und eine Transition als Zielknoten. Die Schaltregel ist dahingehend modifiziert, daß im Falle des Schaltens der Transition über eine Lesekante keine Marken vom Platz abgezogen werden. Der Platz kann also als reine Nebenbedingung zur Transition verstanden werden, die nicht die Nebenläufigkeit einschränkt. Unter der Interleaving-Semantik kann eine

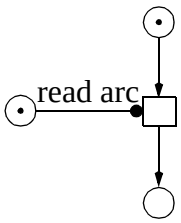


Abbildung 3:Beispiel einer Lesekante

solche Kante durch zwei einfache Hin- und Rückkanten ersetzt werden.

Die Netze, die in den folgenden Abschnitten vorgestellt werden, sind relativ groß und bestehen oft aus vielen Wiederholungen von strukturell identischen Unternetzen. Um diesem Umstand gerecht zu werden, führen wir die folgenden Zeichenkonventionen zur Reduzierung des Platzbedarfes ein. Wenn ein Unternetz n mal wiederholt wird, so stellen wir nur die erste und die letzte Wiederholung dar. Dazwischen machen wir mit drei Punkten die Wiederholung deutlich. Wenn die Unternetze strukturell gleich sind, so werden intuitive Namen gewählt. In der Abbildung 4 ist ein Beispiel dargestellt.

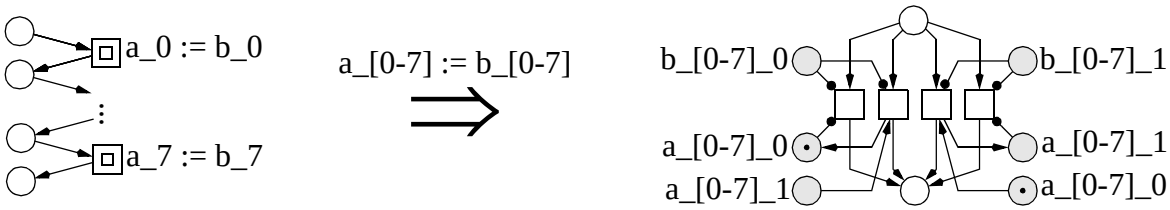


Abbildung 4:Beispiel für die Wiederholung eines Unternetzes

2.2.2 Binäre Repräsentation

Variablen modellieren wir in den Petri-Netzen entsprechend ihrer rechner-internen binären Repräsentation. Zwei Plätze stellen dabei jeweils die Werte 0 und 1 für ein Bit dar (Abbildung 5). Durch diese

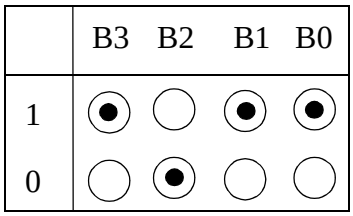


Abbildung 5:Beispiel einer 4-bit Zahl ($11_{10} = 1011_2$)

Darstellung ist es uns möglich, die grundlegenden Algorithmen (mathematische Operationen) für eine binäre Algebra als Petri-Netz zu formulieren.

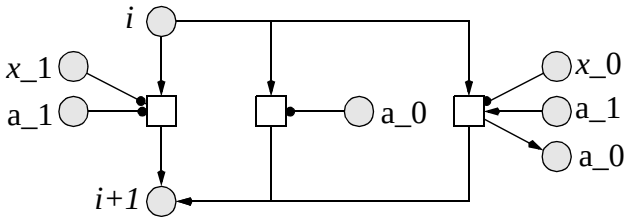
2.2.3 Petri-Netz-Semantik

In diesem Abschnitt geben wir drei Beispiele für die Petri-Netz-Semantik an. Als Voraussetzung benötigen wir (wie auch bei der konventionellen operationalen Semantik) eine erfolgreiche Ausführung der Funktion $Parse$ mit e als Ergebnis. In Analogie zur bereits formulierten operationalen Semantik definieren wir auch in der Petri-Netz-Semantik eine Funktion $Compute_e$.

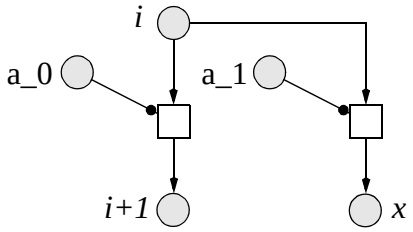
Nach der Ausführung der Funktion $Parse$ müssen wir nur noch die Liste in $Code$ verarbeiten, wobei

jede Anweisung in dieser Liste durch die entsprechende Petri-Netz-Komponente ersetzt wird. Die Beschriftung *i* in den Petri-Netz-Komponenten wird durch die entsprechende Zeilennummer (*i + 1* entsprechend durch den Nachfolger) und *x* durch den Namen des Operanden ersetzt. Die Komposition der Unternetze (für jedes Kommando eines) ergibt ein Modell für das gesamte AWL_0-Programm. Die Kompostion erfolgt dabei rein sequentiell.

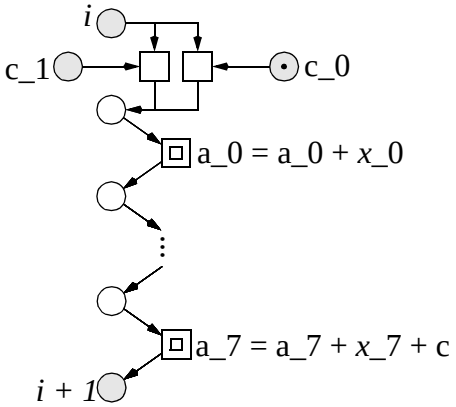
(1) $Compute_e(i, \texttt{AND}\ x)$=



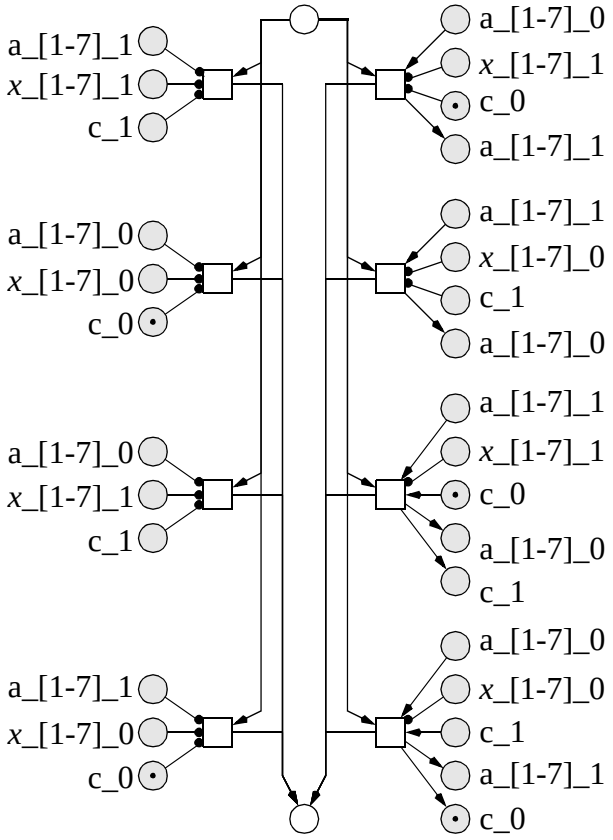
(2) $Compute_e(i, \texttt{JMPC}\ x)$=



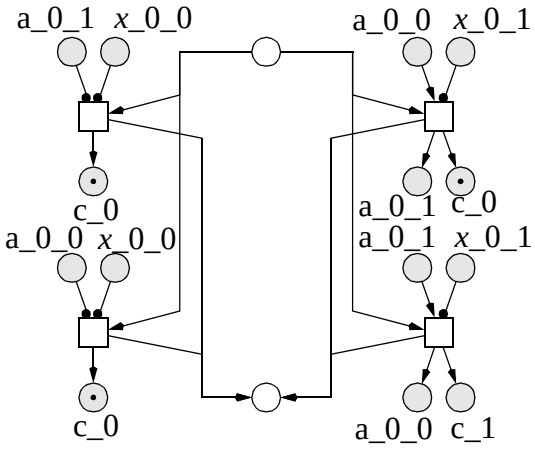
(3) $Compute_e(i, \texttt{ADD}\ x)$=



a_[1-7] = a_[1-7] + x_[1-7] + c



a_0 = a_0 + x_0



3 Von der ungesteuerten Strecke zum Umgebungsmodell

Für die Verifikation einer SPS-Steuerung benötigt man nicht nur ein Modell der Steuerung, sondern auch eine adäquate Beschreibung der ungesteuerten Strecke. Ein Petri-Netz-Modell einer solchen ungesteuerten Strecke nennen wir Umgebungsmodell.

Um ein solches Umgebungsmodell mit dem Modell der Steuerung verbinden zu können, benötigt man eine Beschreibung der Zuordnung der Variablen der Umgebung zu den Variablen des Anwenderprogrammes der SPS. Diese Zuordnungen müssen in einer Datei niedergeschrieben sein, welche beim Kompositionsvorgang zur Verfügung steht. In dieser Datei wird beschrieben, welcher Sensor des Umgebungsmodells mit welcher Eingabevariable des Anwenderprogrammes und welcher Aktor der Umgebung mit welcher Ausgabevariabel des Anwenderprogrammes korrespondiert. In Worten der Automatisierungstechnik wird mit einer solchen Datei ein Beschaltungsplan (Teil der Konfiguration) der SPS mit der entsprechenden Umgebung beschrieben. Im Beispiel im Abschnitt 5 ist eine solche Datei in Auszügen angegeben.

Das Umgebungsmodell ist im allgemeinen ein 1-beschränktes Platz-/Transitionsnetz. Innerhalb dieser Klasse von Petri-Netzen beschreibt die Schaltregel nur die Möglichkeit des Schaltens einer Transition unter Konzession. Ein gesteuerter Schaltzwang für Transitionen ist nicht möglich. Solch ein Mechanismus wird aber bei der Kopplung zwischen Umgebungsmodell und Steuerungsmodell benötigt, da ansonsten ein grundlegendes Theorem der Automatisierungstechnik nicht gewährleistet werden kann. Dieses Theorem besagt, daß während der Ausführung des Anwenderprogrammes eine Zustandsänderung in der Umgebung nicht möglich ist. Im folgenden Abschnitt geben wir eine genauere Beschreibung dieses Theorems. Um das Theorem sicherzustellen, wird ein Mechanismus zur Blockierung von Transition benötigt. Dies wird durch die Einführung eines gesonderten Blockierungsplatzes realisiert.

Der Blockierungsplatz ist ein logischer Platz im Petri-Netz-Modell der ungesteuerten Strecke, der durch Lesekanten mit jeder Transition des Umgebungsmodells verbunden ist. Initial ist dieser Platz markiert. Da der Blockierungsplatz eine Nebenbedingung für jede Transition des Umgebungsmodell ist, kann durch Entnahme der Marke das Umgebungsmodell blockiert werden. Deshalb ist nun eine Synchronisation im Sinne der SPS-Semantik zwischen dem SPS-System und der ungesteuerten Strekke möglich. In der Abbildung 6 wird ein minimales Umgebungsmodell mit einem solchen Blockie-

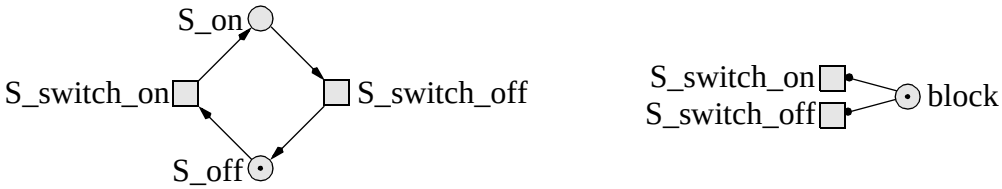


Abbildung 6:Umgebungsmodell mit Blockierungsplatz

rungsplatz gezeigt.

Durch die Verwendung des Blockierungsplatzes kann das angesprochene Theorem in die Praxis umgesetzt werden. Dabei entfernt das Systemprogramm bei Beginn der Abarbeitung des Anwenderprogramms die Marke von dem Platz und legt nach Abarbeitung eine Marke auf den Platz zurück. Durch diese Vorgehensweise werden vor allem uninteressante Zustände des SPS-Systems (aus dem Verständnis der Automatisierungstechnik) eliminiert. Während das Umgebungsmodell unblockiert ist, ist jedoch weiterhin jede Zustandsänderung möglich. So kann es durchaus noch zu Zuständen im Modell

kommen, die nicht in der realen SPS-Anlage auftreten können. Zum Beispiel ist auch keine Zustandänderung möglich, so daß das Programm der SPS immer wieder mit denselben Parametern ausgeführt wird oder daß das Umgebungsmodell mehr als eine Zustandsänderung in Folge vornimmt. Diese Zustände sind aber bei der Analyse irrelevant, da sie ein Systemverhalten beschreiben, was nicht auftreten kann.

Generell wird das Umgebungsmodell zustandsorientiert modelliert. Jeder Zustand, der durch die ungesteuerte Strecke erreichbar ist (z. Bsp. Schalter ist *on* oder *off*), wird durch einen entsprechenden Platz modelliert. Jede Aktion, die eine Zustandsänderung in der ungesteuerten Strecke bewirkt, wird als Transition zwischen den Plätzen modelliert (z. Bsp. *switch_on* und *switch_off*). Ein SPS-System kann nur diskrete Werte verarbeiten, so daß analoge Werte entsprechend diskretisiert werden müssen. Wenn also ein analoger Sensor Bestandteil der ungesteuerten Strecke ist, so modellieren wir jeden signifikanten Zustand bzw. jede signifikante Menge äquivalenter Zustände durch einen Platz im Umgebungsmodell.

Um den Zugriff des Systemprogrammes auf Werte des Umgebungsmodell zu realisieren, müssen alle Plätze, welche Werte repräsentieren, sowie auch der Blockierungsplatz als logische Plätze modelliert werden. Auch Transitionen, die Aktoren in der ungesteuerten Strecke repräsentieren, müssen als logische Transitionen modelliert werden.

4 Das Systemprogramm als Petri-Netz

Jedes SPS-Anwenderprogramm ist in ein SPS-System eingebettet. Ein solches System hat immer ein Systemprogramm, welches die physikalischen Daten der ungesteuerten Strecke in Werte wandelt, die vom Anwenderprogramm verarbeitet werden können, und umgekehrt. Dazu bildet ein Systemprogramm externe Werte auf Werte von internen Variablen ab (Abbildung 7 zeigt die Umsetzung für das

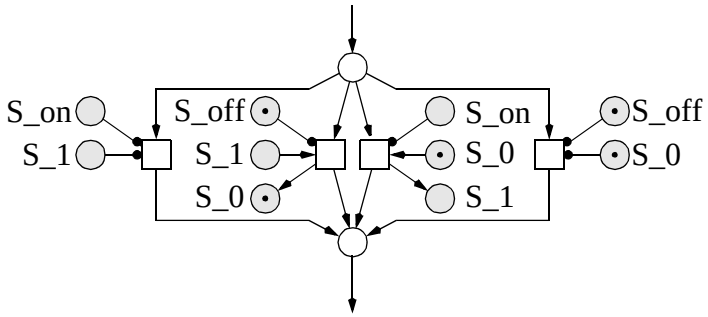


Abbildung 7:Abbildung externer auf interne Werte

kleine Beispiel aus Abbildung 6). In einem SPS-System wird das Anwenderprogramm zyklisch abgearbeitet. Solch ein Zyklus wird als SPS-Zyklus bezeichnet. Der prinzipielle Aufbau eines SPS-Systems aus System- und Anwenderprogramm ist in Abbildung 8 dargestellt.

Im Kapitel 3 wurde bereits auf die grundlegende Annahme zum Verhältnis zwischen Anwenderprogramm und Umgebungsverhalten eingegangen: während der Abarbeitung des Anwenderprogrammes ist keine Zustandsänderung in der Umgebung möglich. Diese Annahme beruht auf dem SPS-Zyklus. In diesem gibt es nur zwei Punkte, an denen eine Interaktion zwischen SPS-Programm und Umgebung auftreten kann. Nur wenn externe Werte in interne abgebildet werden oder entsprechend interne in externe, wird ein Zustand der Umgebung dem SPS-Programm bekannt. Während der Ausführung des Anwenderprogrammes findet kein Kontakt mit der Umgebung statt, so daß ein Zustandswechsel während dieser Ausführung nicht verarbeitet wird. Es ist also wichtig, daß die Ausführung *schnell genug* erfolgt. Wenn die Ausführungsgeschwindigkeit des Anwenderprogrammes im Verhältnis zur Geschwindigkeit, mit der sich die Zustände in der Umgebung ändern, wesentlich schneller ist, so kann

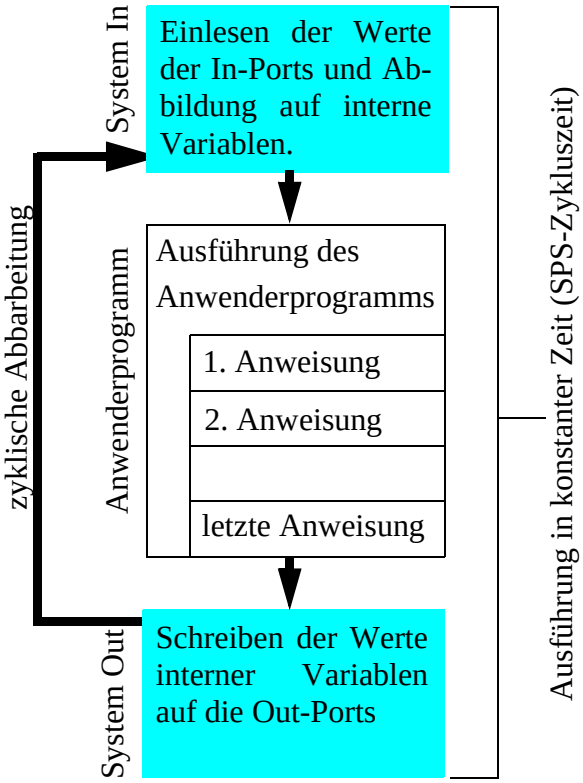


Abbildung 8:Schematischer SPS-Zyklus

man davon ausgehen, daß alle Änderungen in der Umgebung von dem Anwenderprogramm bearbeitet werden können. Somit ist das bereits mehrfach genannte Theorem als Anforderung an die SPS formulierbar.

Damit haben wir eine Relation zwischen der Geschwindigkeit des SPS-Systems (welche die Ausführungszeit bestimmt) und der Geschwindigkeit der Zustandswechsel innerhalb der Umgebung eingeführt. Die Frage ist nun, um welchen Faktor muß das SPS-System schneller sein als die ungesteuerte Strecke. Dieses Verhältnis bestimmt in entscheidendem Maße die Ausführung eines SPS-Systems und kann im Modell des Systemprogramms betrachtet werden. Hierfür stehen drei Alternativen zur Auswahl:

- Nach jedem SPS-Zyklus ist eine Änderung in der Umgebung möglich.
- Nach einer bestimmten Anzahl von Ausführungen des Anwenderprogrammes ist eine Änderung in der Umgebung möglich.
- Eine Zustandsänderung in der Umgebung ist dann möglich, wenn das Anwenderprogramm die Ausgabeparamter für ein Eingabeabbild vollständig berechnet hat.

Diese drei Möglichkeiten, die zu unterschiedlichen Systemprogrammen führen, werden in den folgenden Abschnitten betrachtet.

4.1 Blockierung für die Dauer eines Zykluses

Diese Semantik kann mit dem in Abbildung 9 dargestellten Modell realisiert werden. Der SPS-Zyklus startet und entfernt als erste Aktion die Marke vom Blockierungsplatz. Somit kann eine Zustandsänderung im Umgebungsmodell vollzogen werden. Dieser Zugriff ist auch der Grund dafür, warum der Blockierungsplatz als logischer Knoten modelliert werden sollte. Nachdem durch das Entfernen der Marke vom Blockierungsplatz allen Transitionen im Umgebungsmodell die Konzession entzogen wurde, können die Eingabewerte gelesen und auf Werte von internen Variablen abgebildet werden.

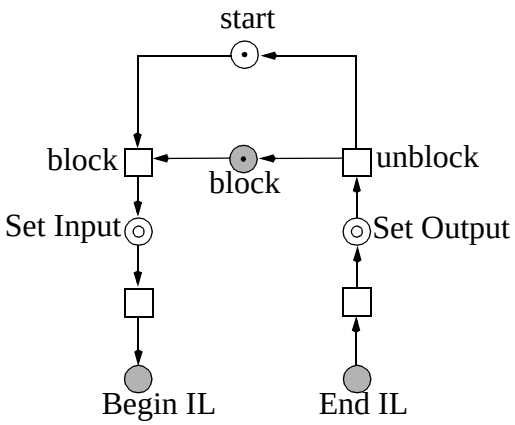


Abbildung 9:Systemprogramm mit Zyklusblockierung

Nach einer Ausführung des Anwenderprogrammes wird die Ausgabe geschrieben und die Umgebung wieder freigegeben, indem eine Marke auf dem Blockierungsplatz abgelegt wird. Die ungesteuerte Strecke wurde immer genau für einen Zyklus gesperrt.

Im Verständnis der Automatisierungstechnik spiegelt dieses Modell den Fakt wieder, daß das SPS-System auf jedes Eingabeabbild in nur einem Zyklus reagiert. Das entsprechende Ausgabeabbild wird in nur einem Zyklus vollständig berechnet.

4.2 Blockierung für *n* Zyklen

Dies scheint der realistische Fall zu sein. Ein SPS-System ist in der Regel um einen Faktor *n* schneller

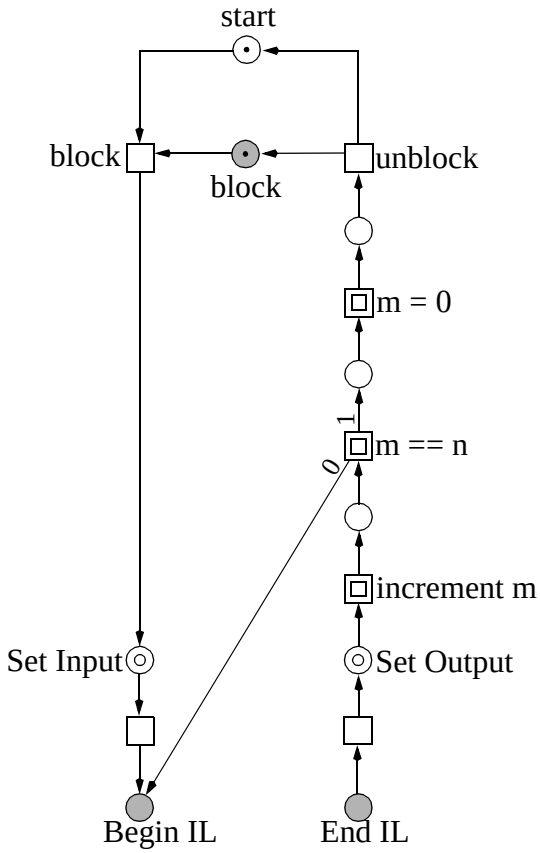


Abbildung 10:Systemprogramm mit Faktor n

als die ungesteuerte Strecke. Um den benötigten Zähler modellieren zu können, verwenden wir die

binäre Repräsentation von Werten aus Abschnitt 2.2.2. Die Umgebung wird für *n* Zyklen nach dem bekannten Verfahren blockiert. Das gesamte Modell ist ein recht großes Petri-Netz, so daß wir nur einen Teil davon in Abbildung 10 zeigen möchten.
Ein bemerkenswerter Aspekt dieser Modellierung ist, daß hiermit die Einführung von Timern möglich wird. In der SPS-Programmierung ist ein Timer kein Realzeit-Timer, sondern er repräsentiert eine Relation zu der Anzahl der Zyklen, welche jeweils in konstanter Zykluszeit ausgeführt werden. Es ist somit möglich, den Timer-Fortschritt mit Hilfe des Zykluszählers zu bestimmen. Um dies zu realisieren, muß die Zykluszeit als Wert modelliert werden. Mit diesem Wert und dem Zähler der SPS-Zyklen kann ein aktueller Timer-Wert zu Beginn jeder Ausführung des Nutzerprogrammes berechnet werden.

4.3 Unbeschränkte Blockierung

In dieser Interpretation wird davon ausgegangen, daß ein SPS-System immer so schnell ist, daß zu einem gegebenen Eingabeabbild immer das zugehörige Ausgabeabbild berechnet werden kann, bevor sich eine Zustandsänderung in der Umgebung vollzieht. Man blockiert also die Umgebung so lange, wie im Anwenderprogramm Änderungen an Variablenwerten vorgenommen werden. Wird kein Wert einer Variablen mehr geändert, liegt kein Rechenprozeß mehr vor und das Ausgabeabbild kann als vollständig berechnet betrachtet werden. In AWL gibt es vier Kommandos, mit denen der Wert einer Variablen verändert werden kann:

- `R` - Reset,
- `S` - Set,
- `ST` - Store,
- `STN` - Store if not.

Jedes dieser Kommandos hat sein eigenes Petri-Netz-Modul, in welchem das Kommando modelliert ist. Jetzt erweitern wir diese Module, um eine Variablenänderung anzuzeigen (siehe Abbildung 11 als Beispiel für die Anweisung `S Step`).

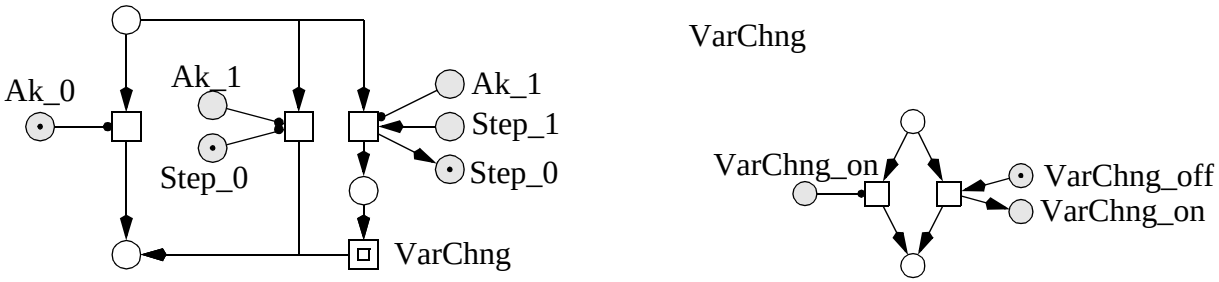


Abbildung 11:Erweiterung einiger Petri-Netz Module

Nach diesen Erweiterungen können wir das Systemprogramm entsprechend modellieren (siehe Abbildung 12). In diesem Modell des Systemprogrammes überprüfen wir, ob eine Änderung einer Variablen vorliegt und geben, solange dies der Fall ist, die Umgebung nicht frei. Erst wenn ein Zyklus erreicht wurde, in dem keine Variable ihren Wert änderte, wird die ungesteuerte Strecke freigegeben.

Mit diesem Systemmodell ist es nun möglich, den Umstand zu beschreiben, daß eine SPS-Steuerung immer wesentlich schneller ist als die zu steuernde Umgebung. Das Ausgangsabbild wird immer vollständig berechnet, bevor sich Eingabegrößen ändern können. Auch dieses Modell kann mit einem Zähler versehen werden, um die Möglichkeit der Verwendung von Timer-Bausteinen zu ermöglichen. Bei diesem Zähler ist jedoch zu beachten, daß eine Wertebereichsabfrage notwendig ist, da die SPS mit einem solchen Systemprogramm unvorhersehbar lange arbeiten kann.

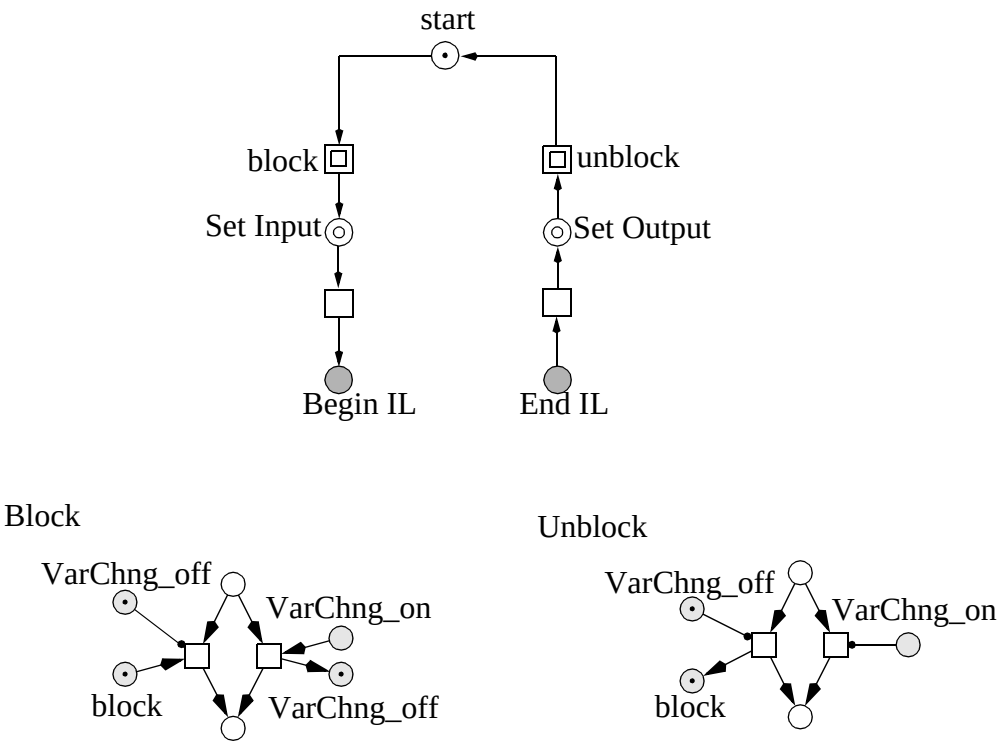


Abbildung 12:Blockierung bis zum vollständigen Ausgebabbild

5 Beispiel: Produktionszelle

In diesem Kapitel wollen wir unsere Methode an einem kleinen, aber realistischen Beispiel, der Produktionszelle ([Lewerentz95]), demonstrieren. Diese Produktionszelle besteht aus sechs physikalischen Komponenten, die gemeinsam als Produktionszelle (siehe Abbildung 13) die Verformung von Metallteilen vornehmen:

- zwei Zu- bzw. Abführbänder,
- ein Hubdrehtisch,
- ein drehbarer Roboter mit zwei Armen,
- eine Presse und
- ein Transportkran.

Der Produktionszyklus dieser Produktionsfolge ist folgendermaßen spezifiziert:

Das Zuführband transportiert die neuen unbehandelten Teile zum Hubdrehtisch. Dieser fährt mit den Teilen in eine Position (Anheben, Rotieren), von der aus der Roboter das Teil mit einem seiner Arme greifen kann. Dieser Arm des Roboters legt das Teil in die Presse. Nach dem Preßvorgang wird das Werkstück durch den anderen Arm des Roboters auf das Abführband transportiert. Damit ein geschlossenes System vorliegt, wurde der Kreislauf durch einen horizontalen Transportkran ergänzt, der die Werkstücke vom Abführband zum Zuführband transportiert.

Ein mögliches Anwenderprogramm in AWL_0 hat ca. 570 Anweisungen. Dieses Programm hat 14 Eingabevariablen, 21 Ausgabevariablen und 22 lokale Variablen. Es wurde ohne Verwendung von Funktionsblöcken und Funktionen geschrieben. In Abbildung 1 ist ein Ausschnitt aus dem gesamten Programm zur Steuerung des Zuführbandes dargestellt.

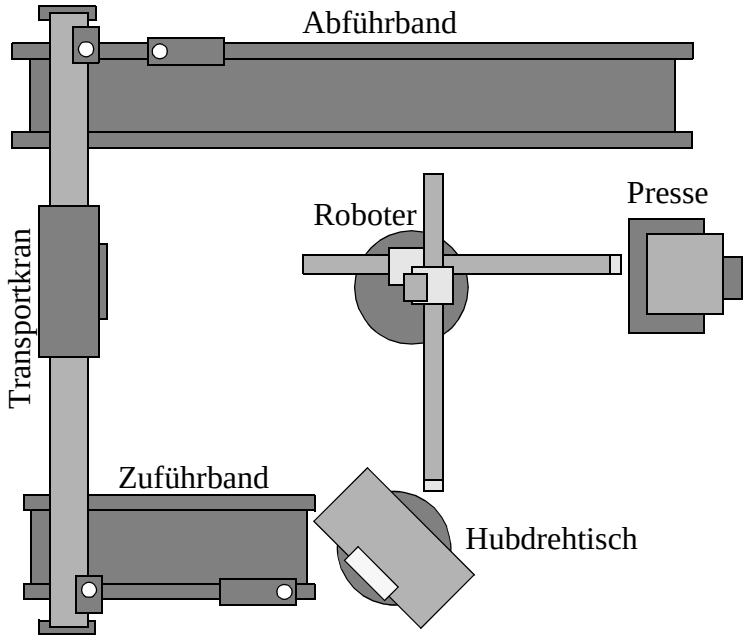


Abbildung 13:Prinzipieller Aufbau der Produktionszelle

5.1 Umgebungsmodell

Zuerst modellieren wir das Umgebungsmodell. Im Beispiel besteht die ungesteuerte Strecke aus den beiden Bändern, dem Hubdrehtisch, dem Roboter, der Presse und dem Kran. Im vorhergehenden Abschnitt zeigten wir den AWL_0-Code für das Zuführungsband, so daß wir an dieser Stelle auch das Zuführband als Beispiel in Abbildung 14 angeben.

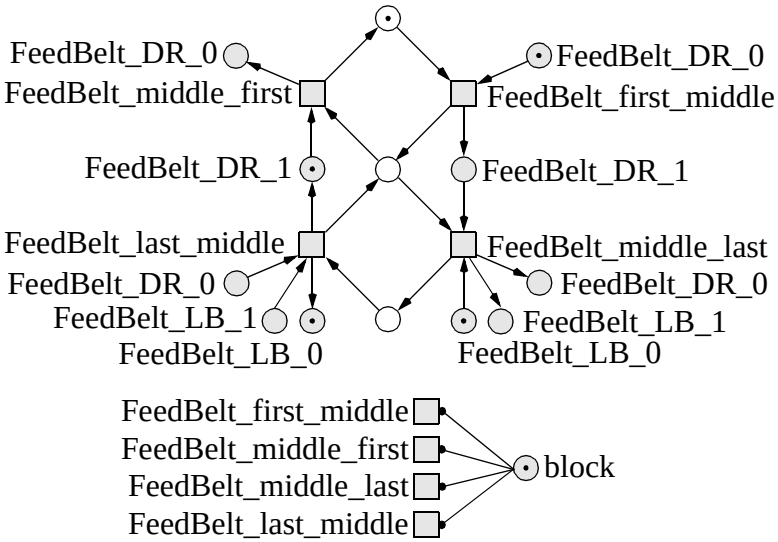


Abbildung 14:Umgebungsmodell für das Zuführband

5.2 Systemprogramm

Wieterhin benötigen wir das Systemprogramm, d.h. man muss sich für eines der in Abschnitt 4 diskutierten Modelle entsprechend der gewünschten Semantik entscheiden. Weiterhin muß dieses Sy-

```
1     (* program for the feed belt *)
2
3     M0: LD    Fbelt_state
4        EQ    0
5        AND   Crane_rd1
6        JMPCN M1
7        S     FBelt_start
8        R     Crane_rd1
9        LD    1
10       ST    Fbelt_state
11    M1: LD    Fbelt_state
12       EQ    1
13       AND   LB_at_Fbelt
14       JMPCN M2
15       R     FBelt_start
16       LD    2
17       ST    Fbelt_state
18    M2: LD    Fbelt_state
19       EQ    2
20       AND   Table_r2t
21       JMPCN M3
22       S     FBelt_start
23       R     Table_r2t
24       LD    3
25       ST    Fbelt_state
26    M3: LD    Fbelt_state
27       EQ    3
28       ANDN  LB_at_Fbelt
29       JMPCN M4
30       R     FBelt_start
31       S     Fbelt_rd
32       S     Fbelt_r2t
33       LD    4
34       ST    Fbelt_state
35    M4: LD    Fbelt_state
36       EQ    4
37       JMPCN M5
38       LD    0
39       ST    Fbelt_state
```

Abbildung 1:Teil des AWL_0-Programmes (Kontrolle des Zuführbandes)

stemprogramm die abzubildenden Werte (externe in interne und umgekehrt) kennen. Dazu wurden bereits im Abschnitt 3 Ausführungen gemacht. In der Abbildung 1 ist ein Ausschnitt aus einer solchen

```
1     feed_belt {
2      file:       feed_belt.ped
3      map_inport:FeedBelt_LB(_1, _0) in LB_at_fbelt(TRUE, FALSE): BOOL
4                 (* light barrier at the end of the feed belt *)
5      map_outport:FeedBelt_DR(_1, _0) in FBelt_start(TRUE, FALSE): BOOL
6                 (* drive of the feed belt *)
7    }
```

Abbildung 1:Teile der Abbildungsdatei

Abbildungsdatei angegeben (Abbildung der Sensoren und Aktoren von bzw. in externe und interne Werte des Zuführbandes) .

5.3 Modell des Anwenderprogrammes

Im Abschnitt 2.2.3 setzten wir eine Ausführung der Funktion *Parse* mit einem bestimmten AWL_0 Programm als Parameter voraus. Im Falle des Beispieles ist natürlich das gegebene AWL_0-Programm für die Produktionszelle dieser Parameter. Nach der erfolgreichen Ausführung der Funktion überfüh-

ren wir das Ergebnis in ein Petri-Netz (siehe Abbildung 15) mit Hilfe der Petri-Netz Semantik aus

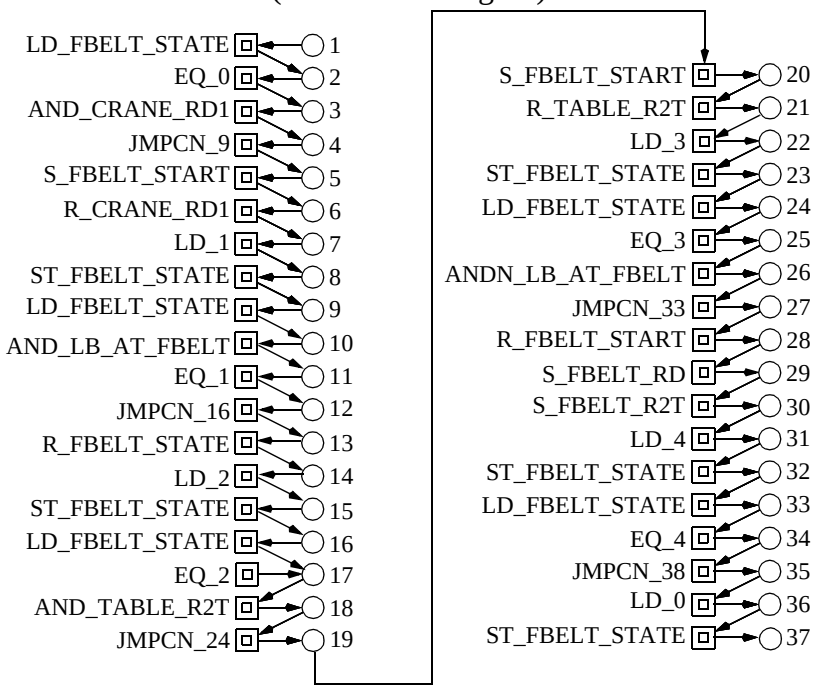


Abbildung 15:Petri-Netz des Anwenderprogrammes

Kapitel 2. Der erste Platz dieses Netzes heißt `Begin IL` und der letzte PLatz trägt die Bezeichnung `End IL`. Über diese beiden Plätze wird später die Kopplung zum Systemprogramm erfolgen. Die Makrotransitionen repräsentieren Unternetze, die Module der Petri-Netz-Semantik enthalten. Die Verbindungsplätze, welche in der Petri-Netz-Semantik noch mit i bezeichnet wurden, werden hier durch konkrete Zeilennummern ersetzt. Diese Verbindungsplätze sind logische Plätze und werden dadurch entsprechend automatisch miteinander verschmolzen (zwei logische Plätze mit gleichem Namen repräsentieren denselben Platz). Als Ergebnis dieser Verschmelzung erhalten wir eine Sequenz von Unternetzen, wobei jedes Unternetz ein Petri-Netz-Repräsentant für die entsprechende Anweisung ist. Die Position innerhalb der Sequenz entspricht der ursprünglichen Position der Anweisung in der Anweisungsliste. Diese Struktur ermöglicht es, recht einfach Ergebnisse der Analyse des Petri-Netzes im Ursprungscode wiederzufinden bzw. zu interpretieren.

Die initiale Markierung des Petri-Netzes ist durch das Tupel e als Ergebnis der Funktion $Parse$ gegeben. Der Teil Env_0 des Tupels beinhaltet alle vom Programm verwendeten Variablen, deren Typen (Anzahl der Plätze pro Variable) und deren initiale Werte (Markierung dieser Plätze).

5.4 Komposition zum Systemprogramm

Nachdem das Systemprogramm, das Umgebungsmodell und das Anwenderprogramm bereits als Petri-Netze modelliert wurden, müssen wir diese Teile noch zum gesamten Systemmodell komponieren. Wie bereits ausgeführt, erfolgt die Komposition zwischen Systemprogramm und Anwenderprogramm über die Plätze `Begin IL` und `End IL` sowie natürlich über die Plätze, die interne Werte von externen Werten repräsentieren. Das Umgebungsmodell wird durch die bereits mehrfach angesprochenen Abbildungen zwischen Werten mit dem Systemmodell gekoppelt. Abbildung 16 zeigt das komponierte Systemmodell, wobei die Unternetze die entsprechenden Teilmodelle repräsentieren.

6 Zusammenfassung und Ausblick

In diesem Bericht wurde eine Methode beschrieben, wie ein SPS-System mit einem in AWL_0 ge-

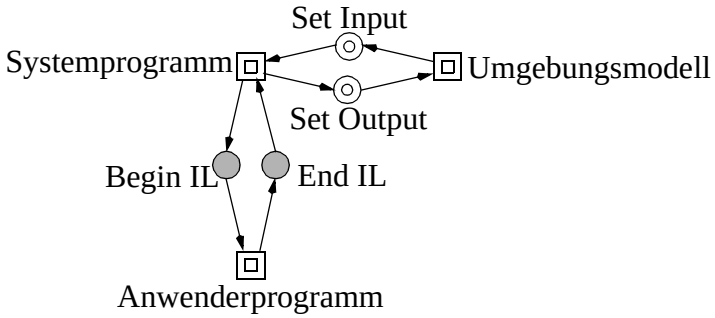


Abbildung 16:Komposition des Systemmodells

schriebenen Anwenderprogramm in ein Petri-Netz überführt werden kann. Das sich ergebende Petri-Netz verwendet keine speziellen Elemente mit einer neuen Semantik; alle verwendeten Elemente haben nur syntaktische Auswirkungen. Das sich aus der Transformation ergebende Petri-Netz ist deshalb mit herkömmlichen Methoden und Verfahren analysierbar, da es der Klasse der sicheren Platz-/Transitionsnetze angehört. Die meisten Analysemethoden arbeiten erfahrungsgemäß recht gut auf Netzen dieser Klasse. Ergänzend kann das Petri-Netz zur Simulation der Ausführung des SPS-Programmes benutzt werden.

In der Zukunft wird die Sprache AWL_0 bis zum Sprachumfang von AWL erweitert. Letztendlich soll die hier vorgestellte Methode für Multiprozessor- und Multitask-Systeme angewendet werden. Bekanntlich erreichen gerade in diesem Anwendungsbereich Petri-Netze die besten Ergebnisse.

Bibliographie

[DIN EN 61131-3]	DIN EN 61131-3 (IEC 1131-3) *Speicherprogrammierbare Steuerungen,* *Teil 3: Programmiersprachen* Deutsche Version der EN 61131-3: 1993
[DIN EN 574]	DIN EN 574 (EN 574) *Sicherheit von Maschinen: Zwei-Hand-Schalter;* Deutsche Version der EN 574: 1991
[Hanisch97]	H.-M. Hanisch, J. Thieme, A. Lüder, O. Wienhold *Modeling of PLC Behavior by Means of Timed Net Condition / Event Systems* IEEE Int. Symposium on Emerging Technologies and Factory Automation (ETFA ‘97), Los Angeles, September 1997, S. 361-396
[Heiner97-1]	M. Heiner, H. Meier, T. Menzel, T. Mertke *Petri-Netz basierte Methoden zur sicherheitsorientierten Zertifizierung von SPS Anwenderprogrammen* BTU-Cottbus Technische Berichte I-19/1997, Cottbus, Dezember 1997
[Heiner97-2]	M. Heiner, T. Menzel *Petri-Netz-Semantik für die SPS-Anwenderprogrammiersprache Anweisungsliste* BTU-Cottbus Technische Berichte I-20/1997, Cottbus, Dezember 1997

[Heiner98-1] M. Heiner
Petri Net Based System Analysis without State Explosion
High Performance Computing ‘98, Boston, April 1998, S. 394 - 403

[Heiner98-2] M. Heiner, T. Menzel
Instruction List Verification Using a Petri Net Semantics
IEEE Int. Conf. on Systems, Man, and Cybernetics, San Diego, Oktober 1998

[Meier98] H. Meier; T. Mertke
Sicherheitsfachsprache zur Formulierung und Verifikation steuerungstechnischer Anforderungen
Manuskript eingereicht zur EKA’99

[Lewerentz95] C. Lewerentz, Lindner
Formal Development of Reactive Systems - Case Study Production Cell
LNCS 891, 1995.

[Rausch96] Mathias P. Rausch
Modulare Modellbildung, Synthese und Codegenerierung erignisdiskreter Steuerungssysteme
Dissertation, Universität Magdeburg, 1997

[Rausch97] M. Rausch, B. Krogh
Transformations Between Different Model Forms in Discrete Event Systems
IEEE Int. Conf. on Systems, Man, and Cybernetics, Orlando, Florida, Oktober 1997, S. 2841-2846

[Spranger98] J. Spranger
Combining Structural Properties and Symbolic Representation for Efficient Analysis of Petri Nets
Workshop on Concurrency, Specification & Programming (CSP ’98), Berlin, 28. - 30. September 1998, ISSN 0863-095, S. 236 - 244.