Monika Heiner, Thomas Mertke, Peter Deussen
mh@informatik.tu-cottbus.de, mertke@aut.tu-cottbus.de,
pd@informatik.tu-cottbus.de, http://www-dssz.informatik.tu-cottbus.de,
http://www.aut.tu-cottbus.de/safety-knight.html

A Safety-oriented Technical Language for the Requirement Specification in Control Engineering

Computer Science Reports
09/01
May 2001

Brandenburg University of Technology at Cottbus

Faculty of Mathematics, Natural Sciences and Computer Science

Institute of Computer Science

This work is supported by the German Research Council under grant ME 1557/1-1,2

Computer Science Reports
Brandenburg University of Technology at Cottbus
Institute of Computer Science

Head of Institute:
Prof. Dr. Bernhard Thalheim thalheim@informatik.tu-cottbus.de
BTU Cottbus
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus

Research Groups:	Headed by:
Computer Engineering	Prof. Dr. H. Th. Vierhaus
Computer Network and Communication Systems	Prof. Dr. H. König
Data Structures and Software Dependability	Prof. Dr. M. Heiner
Database and Information Systems	Prof. Dr. B. Thalheim
Programming Languages and Compiler Construction	Prof. Dr. P. Bachmann
Software and Systems Engineering	Prof. Dr. C. Lewerentz
Theoretical Computer Science	Prof. Dr. B. von Braunmühl

CR Subject Classification (1998): D.2.1, D.2.2, D.2.4, F.3.1

Printing and Binding: BTU Cottbus

ISSN: 1437-7969

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Bei der heutigen kundenorientierten Fertigung von Maschinen und Anlagen stellt die Erzeugung der zugehörigen Steuerungssoftware einen wesentlichen Zeit- und Kostenfaktor dar. Software ist zu einem integralen Bestandteil zeitgemäßer Infrastrukturen geworden und viele technische Systeme sind ohne die Leistungsfähigkeit moderner Computertechnik nicht denkbar. Andererseits kommt es immer wieder zu Ausfällen solcher Systeme. Die Medien berichten über Unfälle und Katastrophen in sicherheitssensiblen Anwendungen (Stichworte Airbus, Ariane-5-Erststart, Mars-Mission). Oftmals lässt sich das Fehlverhalten, nach gründlicher Analyse, auf Fehler in den Steuerungssystemen - und speziell auf Mängel in der verwendeten Software - zurückführen.

Die zunehmende Komplexität moderner Software und das steigende Sicherheitsbedürfnis der Endanwender erfordern neue Methoden und Herangehensweisen, mit denen die Korrektheit und somit die Sicherheit von Steuerungsprogrammen verbessert werden kann. In den letzten Jahren haben sich hierbei zwei Hauptströmungen herausgebildet. Ein Ansatz besteht darin, den Nachweis der geforderten Eigenschaften am bereits erstellten Steuerungsprogramm durch Tests oder besser durch Verifikation zu erbringen. Eine andere Möglichkeit besteht darin, aus einer gegebenen Spezifikation automatisch ein Steuerungsprogramm zu generieren, das exakt den gestellten Anforderungen entspricht.

Beiden Methoden ist die Notwendigkeit einer korrekten, vollständigen und widerspruchsfreien Darstellung der geforderten Programmeigenschaften gemeinsam. Deshalb liegt ein entscheidender Ansatz zur Verbesserung der Softwarequalität in der Anwendung formaler Methoden in der Design-, Spezifikations- und Entwicklungsphase der Software. Die in diesem Beitrag vorgestellte Sicherheitsfachsprache (SFS) wurde innerhalb eines Forschungsprojektes[1] definiert, das sich mit der Entwicklung von Methoden und Werkzeugen zur Verifikation von SPS-Anwenderprogrammen beschäftigt, und dient in diesem Zusammenhang zur Spezifikation der Programmeigenschaften.

1.1 Ursachen für fehlerhafte Software

Die Entwicklung eines Steuerungsprogramms für eine zu automatisierende Anlage kann man als einen Prozess betrachten, bei dem festgelegte und gewünschte Eigenschaften in Software implementiert werden. Diese geforderten Eigenschaften werden üblicherweise in einem Pflichtenheft abgelegt, das durch Kooperation von Auftraggeber und Softwareentwickler erstellt wird. Ein Pflichtenheft (also die Spezifikation) enthält die notwendigen Abläufe des Programms, die erforderlichen Reaktionen auf festgelegte Ausnahmesituationen, aber auch Festlegungen, welche Programmreaktionen unbedingt vermieden werden müssen.

Konventionell (also per Hand) erstellte Programme weisen jedoch häufig nicht nur die gewünschten und erforderlichen Eigenschaften auf. Die oben genannten Beispiele und die persönliche Erfahrung jedes Einzelnen zeigen, dass sich Software mitunter anders verhält als es eigentlich gefordert und erwartet wird. Dieses ‚Fehlverhalten', wie es dann immer genannt wird, beruht jedoch darauf, dass die erzeugte Software auch zusätzliche ‚versteckte' Eigenschaften hat. Diese wurden nicht spezifiziert und vom Programmierer auch nicht direkt implementiert, sie sind dennoch vorhanden. Viele dieser Eigenschaften können harmlos sein,

[1] gefördert unter DFG ME 1557/1-1,2

es gibt mit großer Wahrscheinlichkeit jedoch auch solche, die während des Betriebes zu riskanten und gefährlichen Reaktionen des Programms führen können. Das Fehlverhalten eines Programms kann man wie folgt klassifizieren:

a) das Programm führt eine erwartete Aktion nicht aus, d. h. es reagiert nicht,
b) das Programm reagiert plötzlich anders, als es erwartet bzw. gefordert wird.

Den Fall b) kann man unter Beachtung der Spezifikation noch weiter untergliedern in:

b1) das Programm führt eine Aktion aus, obwohl diese bei der Spezifikation bereits als unerwünscht oder sogar gefährlich eingestuft wurde,
b2) das Programm führt eine völlig unerwartete Aktion aus.

Worin liegen die Ursachen für ein solches Verhalten?

Zur Klärung dieser Frage soll der Spezifikationsprozess von Software folgendermaßen veranschaulicht werden. In dem betrachteten System, in das die Software integriert werden soll, gibt es üblicherweise eine Anzahl von variablen Größen, die während des Betriebes verschiedene Werte (Zustände) einnehmen können. Durch Kombination dieser Variablenzustände entsteht eine (oftmals sehr große, aber dennoch endliche) Menge von Zuständen, die das gesamte System einnehmen kann. Innerhalb dieser Menge von (theoretisch einnehmbaren) Systemzuständen gibt es nun einerseits solche, die erwünscht, d. h. normale Betriebszustände des Systems sind (Abbildung 1). Auf der anderen Seite gibt es auch Systemzustände, die unbedingt vermieden werden müssen, weil sie eine Gefährdung des Systems oder der Umwelt (z. B. des Bedieners) darstellen. Das zu entwickelnde Steuerungsprogramm soll nun den Zustandsraum des Systems so eingrenzen, dass einerseits die gewünschten Systemzustände (und zwar in einer geforderten Reihenfolge) erreicht und andererseits die unerwünschten Systemzustände nicht mehr eingenommen werden.

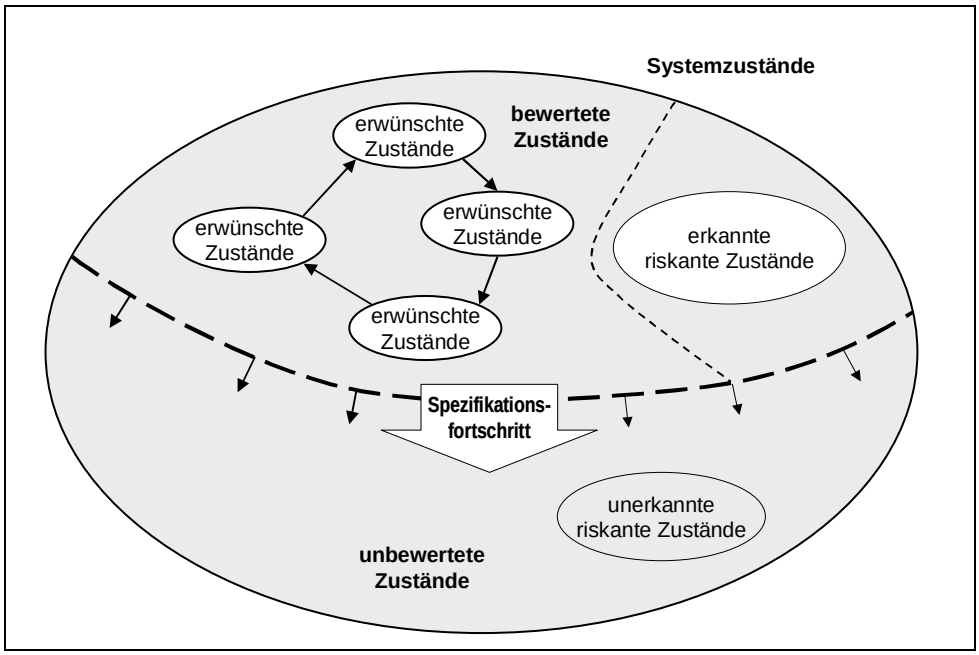


Abbildung 1 - Spezifikation als Bewertung von Systemzuständen

Bei der Spezifikation des Programms sollte jeder einzelne Systemzustand dahingehend bewertet werden, wie sich das Programm verhalten soll, wenn das System diesen Zustand einnimmt, bzw. ob das System diesen Zustand überhaupt einnehmen darf. Durch diese

Bewertung und die Festlegung der Reaktion des Steuerungsprogramms auf diese Systemzustände wird das gewünschte Verhalten des Steuerungsprogramms festgelegt.

Jedoch ist bei Systemen technisch relevanter Größe eine vollständige Spezifikation, also die Betrachtung aller Systemzustände, nicht mehr möglich. Infolgedessen bleibt eine Menge von Systemzustände unbewertet (unspezifiziert). Hierbei entsteht ein Problem, weil es innerhalb der Menge der unbewerteten Systemzustände auch wiederum riskante Zustände geben kann, die zudem noch unerkannt sind.

Wird die festgeschriebene Spezifikation in reale Software überführt, so weist diese oftmals die bereits beschriebenen Fehlfunktionen auf. Die Gründe hierfür sind in zwei Tatsachen zu finden (Abbildung 2):

1. unvollständige Umsetzung der Spezifikation, d. h. die spezifizierten Eigenschaften werden nicht vollständig erfüllt (Fehlerklasse a),
2. Implementierung zusätzlicher Eigenschaften (Fehlerklasse b1 bzw. b2).

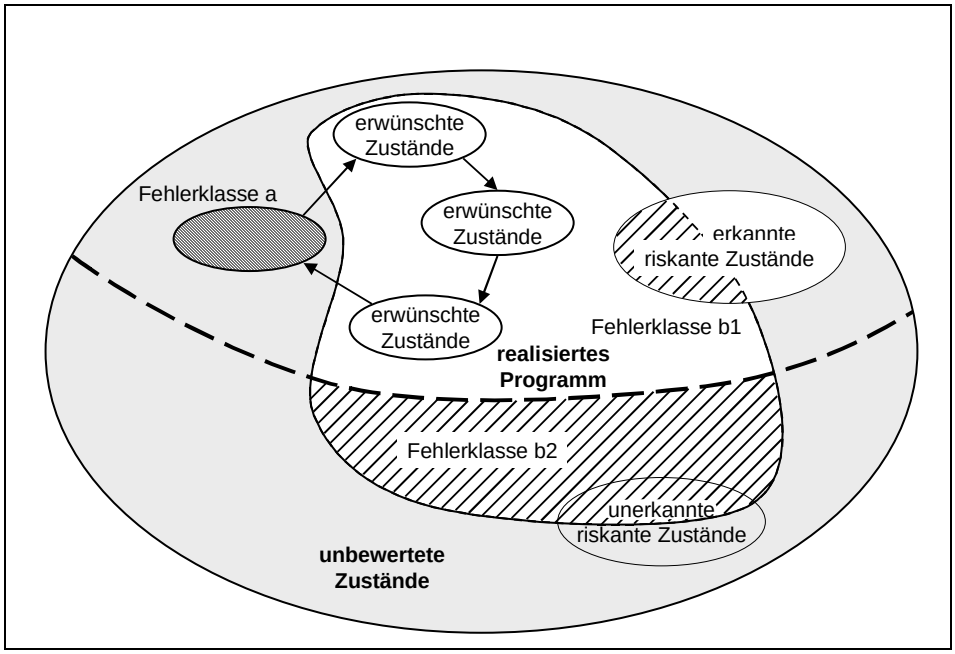


Abbildung 2 - Ursachen für Softwarefehler

Somit entstehen bei der Erstellung von Software kritische Bereiche, die in der Abbildung schraffiert hervorgehoben wurden. Ziel der anschließenden Inbetriebnahmephase muss es dann sein, diese unbeabsichtigt erzeugten Unzulänglichkeiten zu erkennen und zu beseitigen. Die Überprüfung der Eigenschaften wird heute üblicherweise durch verschiedene Tests durchgeführt. Diese Tests beinhalten hauptsächlich die Inspektion der geforderten Solleigenschaften des Programms, die Kontrolle der Programmreaktionen auf zuvor festgelegte Fehlerszenarien sowie - wenn auch nur stichprobenartig - die Vermeidung gefährlicher Situationen.

1.2 Verifikation durch Model-Checking

Die erwähnten Tests hinsichtlich der Abwesenheit von unerwünschten Programmreaktionen müssen aufgrund der Komplexität von Programmen mit einer industriell relevanten Größe jedoch immer als unvollständig betrachtet werden.

Um einen vollständigen und damit exakten Nachweis zu erhalten, ob ein untersuchtes System die festgelegten Anforderungen erfüllt, wird deshalb in der Informatik seit längerer Zeit das Verfahren des Model-Checking eingesetzt. Man versteht darunter ein Verfahren, das überprüft, ob ein Modell des zu untersuchenden Systems eine gegebene Spezifikation erfüllt oder ob es Verstöße gegen diese Vorgaben gibt. Das Modell kann dabei auf verschiedene Arten beschrieben werden, häufig wird eine Darstellung als endlicher Automat verwendet. Zur Beschreibung der Spezifikation werden Formeln der Temporalen Logik genutzt. Beispiele für gebräuchliche Model-Checking-Werkzeuge sind SMV [McMillan1992], HyTech [Alur1996], INA [Starke1992], Prod [VHHP1995] oder PEP [Best1995]. Das Model-Checking-Verfahren liefert als Ergebnis der Überprüfung entweder eine Bestätigung (d. h. einen formalen Beweis) für die vollständige Einhaltung der Spezifikation in allen modellierten Zuständen des Systems oder mögliche Gegenbeispiele, in denen eine oder mehrere Anforderungen nicht erfüllt werden.

Im hier vorgestellten Forschungsprojekt (Abbildung 3) besteht das zu untersuchende System aus einer Speicherprogrammierbaren Steuerung (SPS, engl. PLC), dem Steuerungsprogramm und der zu steuernden Anlage [HMM2000, Mertke2000].

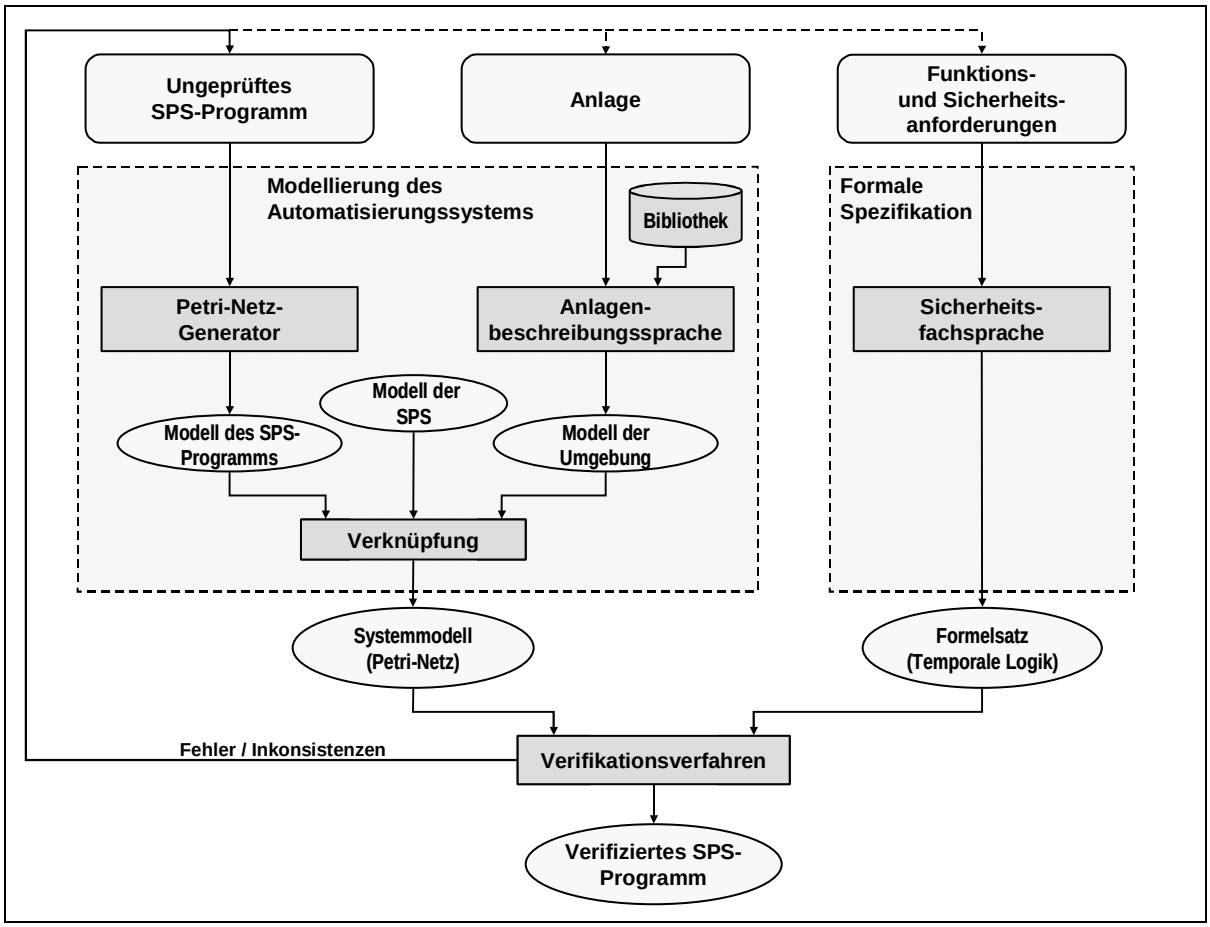


Abbildung 3 - Übersicht über das Verfahren

Das eigentliche Untersuchungsobjekt ist jedoch nur das Steuerungsprogramm (auch SPS-Programm, Anwenderprogramm), da die Anlage und die verwendete Steuerung als feste, unveränderliche Komponenten angenommen werden müssen.

Das SPS-Programm wurde durch den Automatisierungstechniker entworfen und programmiert und wird dann auf der SPS ausgeführt. Die zu steuernde Anlage ist entweder direkt oder über ein Bussystem mit der SPS verbunden. Hierfür gibt es einen Beschaltungsplan, der eine Auflistung der verwendeten Aktoren und Sensoren, sowie spezifische Details der Signalparameter enthält. Dieser Beschaltungsplan ist die Grundlage für die Verbindung der Aktoren und Sensoren mit den Ausgangs- und Eingangsbaugruppen der SPS. Gleichzeitig dient er auch als inhaltliche Schnittstelle zwischen dem SPS-Programm und der Anlage, da er auch eine Auflistung der Signale darstellt, die im SPS-Programm wieder zu benutzen sind. Die einzelnen Schritte zur Erstellung des Systemmodells werden im Kapitel 2 ausführlich behandelt.

Die am Modell zu verifizierende Spezifikation ist durch die Anforderungen des Pflichtenheftes gegeben. Diese Festlegungen liegen jedoch in automatisierungstechnisch spezifischen Darstellungsformen bzw. verbal vor. Da für das Model-Checking jedoch Formeln der Temporalen Logik benötigt werden, müssen die Anforderungen zunächst formalisiert und in die benötigten Formeln überführt werden. Die hierbei verwendete Sicherheitsfachsprache und die unterlagerte Logik werden in den Kapiteln 3 und 4 dargestellt.

Die Überprüfung der Anforderungen erfolgt im Verifizierungsschritt durch den Model-Checker. Sollten alle Anforderungen erfüllt und als richtig verifiziert worden sein, kann das Steuerungsprogramm als korrekt bezüglich den gegebenen Bedingungen bezeichnet werden. Sollten Fehler auftreten, ist eine Modifikation der Eingangsinformationen, also des SPS-Programms, des Anlagenmodells oder der Spezifikation, notwendig.

Im Kapitel 5 wird die Anwendung der Sicherheitsfachsprache und die Erzeugung der temporallogischen Formeln beispielhaft erläutert.

2. Modellierung des Automatisierungssystems

Das Modell des Automatisierungssystems, das als Basis für den Verifizierungsvorgang dienen soll, umfasst drei Komponenten:

1. die zu steuernde Anlage mit ihren Sensoren und Aktoren,
2. die verwendete Steuerung mit den integrierten Betriebssystemfunktionen,
3. das vom Automatisierungstechniker erstellte Steuerungsprogramm.

Diese drei Komponenten werden einzeln modelliert und in einem anschließenden Kompositionsschritt zu einem Systemmodell vereinigt. Grundlage für die Zusammenführung der Einzelmodelle ist der Beschaltungsplan.

2.1 Betrachtungen zur Arbeitsweise einer SPS

An dieser Stelle erfolgt eine kurze Einführung zur Arbeitsweise einer Speicherprogrammierbaren Steuerung, da sich hieraus grundlegende Schlussfolgerungen bezüglich der Erstellung des Systemmodells und der Formulierung der Anforderungen an das Steuerungsprogramm ergeben.

Die Abbildung 4 zeigt den allgemeinen Aufbau einer automatisierten Anlage.

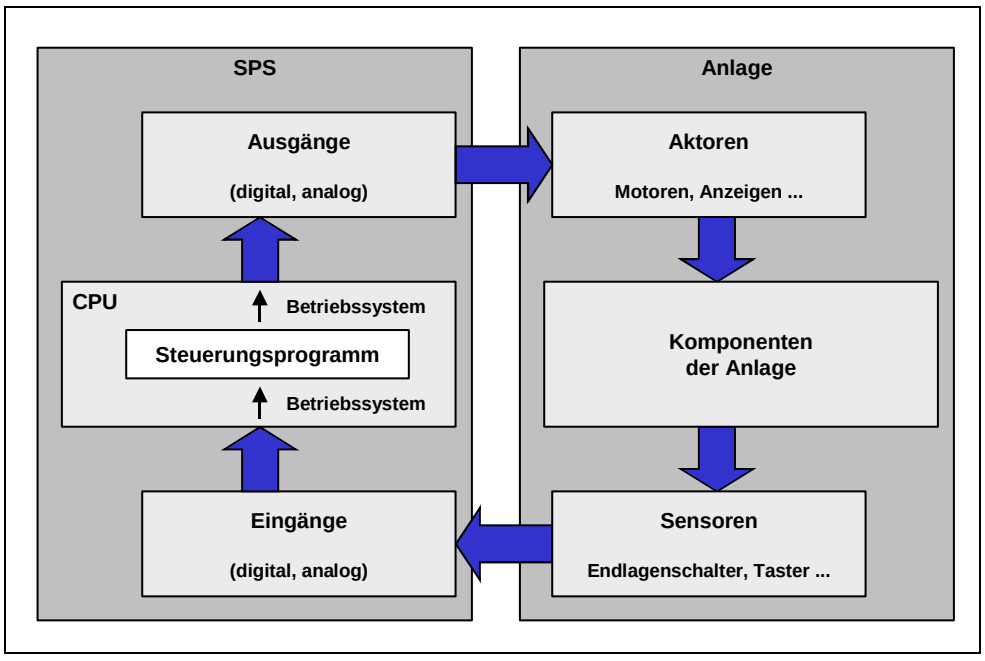


Abbildung 4 - Informationsfluss in einem automatisierten System

Eine SPS arbeitet immer mit einer Maschine oder Anlage zusammen, sie empfängt Informationen aus der Umgebung, verarbeitet diese entsprechend einem vorgegebenem Algorithmus und gibt wiederum Informationen an die Umgebung ab. Die Informationen der Umgebung werden durch Sensoren aufgenommen und in elektrische Signale umgewandelt. Diese werden über digitale oder analoge Eingangsbaugruppen in das SPS-System eingebracht und durch das Betriebssystem eingelesen. Diese Eingangssignale werden entsprechend dem Steuerungsprogramm verarbeitet. Die berechneten Ergebnisse werden wiederum durch das Betriebssystem an die Ausgangsbaugruppen geliefert. Diese übermitteln die neuen

Anweisungen als elektrische Signale an die Aktoren der Maschine, wo die gewünschten Aktionen ausgeführt werden.

Der entscheidende Punkt bei einer SPS ist, das dieses Arbeitsverhalten zyklisch ist. Der zuvor beschriebene Prozess erfolgt in einem festgelegten und konstanten Rhythmus, der auch als *SPS-Zyklus* bezeichnet wird. Dabei werden zu Beginn des Zyklus die Signale der Umgebung eingelesen und zwischengespeichert, mit diesen Werten und internen Variablen des Steuerungsprogramms werden danach die Berechnungen durchgeführt. Nach Beendigung des Zyklus werden die Ergebnisse der Berechnungen an die Umgebung ausgegeben. Zusätzlich gibt es noch eine gewisse Wartezeit, bis der SPS-Zyklus wieder von vorn gestartet wird. Der Neustart des SPS-Zyklus erfolgt in einem konstanten Zeitraster, der *Zyklus*- bzw. *Abtastzeit*.

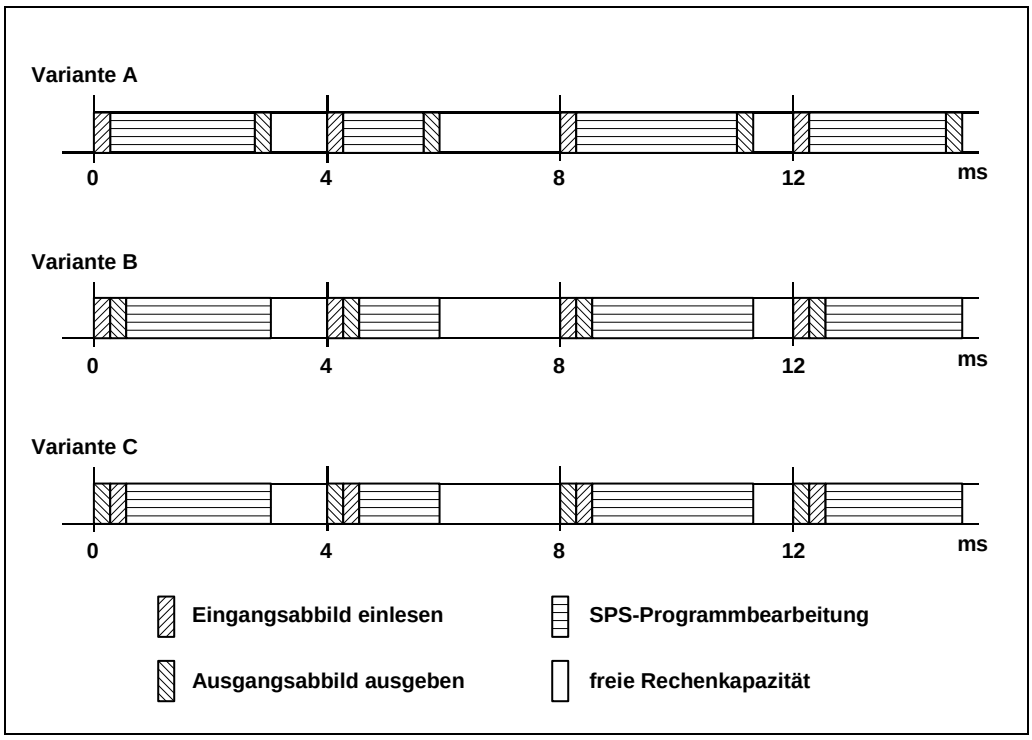


Abbildung 5 - Zyklische Arbeitsweise einer SPS

Die Reihenfolge der dargestellten Teilprozesse wird in der Literatur recht unterschiedlich beschrieben (Varianten A, B, C). Allen Varianten ist gemeinsam, dass zwischen dem Eintreten bestimmter Ereignisse in der Umgebung und der entsprechenden Reaktion durch die SPS immer eine gewisse Zeit vergeht. Im ungünstigsten Fall tritt ein Ereignis genau dann ein, wenn die SPS gerade begonnen hat, das Programm zu bearbeiten. Das Eingangssignal kann also erst mit dem darauffolgenden Zyklus eingelesen und verarbeitet werden. Die worst-case-Reaktionszeit einer SPS beträgt demnach das Doppelte der Zykluszeit, im best-case beträgt sie immerhin auch noch die einfache Zykluszeit. Diesen SPS-typischen Umständen muss bei der späteren Formulierung der Anforderungen unbedingt Rechnung getragen werden.

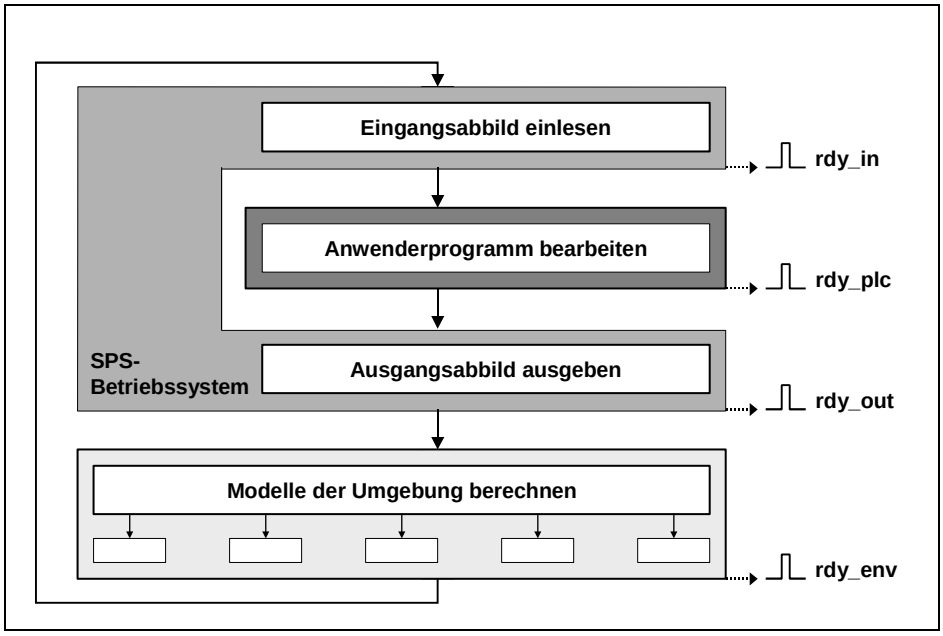


Abbildung 6 - Beobachtungsvariablen zur Identifikation des Systemzustandes

Um die zyklische Ausführung der einzelnen Abschnitte des SPS-Zyklus (Eingänge lesen, Programm bearbeiten, Ausgänge schreiben) beobachten zu können, werden Beobachtungsvariablen definiert (vgl. Abbildung 6). Diese stehen fast immer auf FALSE. Lediglich am Abschluss eines Abschnittes des SPS-Zyklus wird die jeweilige Beobachtungsvariable kurzzeitig auf TRUE gesetzt, um diesen Moment lokalisieren zu können.

Beobachtungsvariable	Bedeutung
rdy_in	das Einlesen der Eingangsvariablen ist abgeschlossen,
rdy_plc	die Bearbeitung des SPS-Programms ist abgeschlossen,
rdy_out	das Ausgeben der Ausgangsvariablen ist abgeschlossen,
rdy_env	die Berechnung des Umgebungsmodells ist abgeschlossen.

Tabelle 1 - Definition von Beobachtungsvariablen

Von diesen Beobachtungsvariablen werden ‚rdy_in' und ‚rdy_plc' bei der Erstellung der CTL-Formeln (Abschnitt 0)verwendet. Die beiden anderen Variablen werden momentan nicht verwendet, wurden dennoch aus Gründen der Vollständigkeit definiert.

2.2 Definition der Registernetze

Wir wollen an dieser Stelle nicht auf Petri-Netze im Allgemeinen eingehen; der interessierte Leser sei z. B. auf [Reisig1991] oder [Peterson1981] verwiesen, während die Standardquelle zur Analyse von Petri-Netzen [Starke1990] ist. Zu Highlevel-Netzen (insbesondere zu gefärbten Netzen) sei auf [Jensen1997] verwiesen.

Zur Beschreibung des Verhaltens von SPS-Programmen wurde in [Deussen2001] eine Highlevel-Petri-Netzklasse, die sog. *Registernetze* definiert. Registernetze stellen den Steuerfluss eines zu modellierenden Systems mit Hilfe eines Petri-Netzes dar, den Datenfluss hingegen unter Verwendung von *Registern*, d. h. Variablen, die als Werte natürliche Zahlen bzw. ein spezielles Symbol $\perp$ annehmen können ($\perp$ wird „undefiniert" gelesen).

Im Folgenden gehen wir nun genauer auf die Semantik der Registernetze ein, da im Kapitel 4.4 auf diese zur Erklärung der exakten Semantik der eingesetzten Temporalen Logik Bezug genommen werden wird.

Die Abbildung 7 stellt anhand einer Transition *t* die Elemente eines Registernetzes dar. Die Vor- und Nachplätze von *t* sind als unausgefüllte Kreise dargestellt; die Plätze und Transitionen eines Registernetzes bilden zusammen mit den Kanten zwischen ihnen ein sicheres Platz/Transitions-Netz (d. h. ein Petri-Netz, auf dessen Plätzen bei jeder erreichbaren Markierung höchstens eine Marke zu finden ist). Daneben gibt es für jede Transition eine Menge $\{ir_1, ir_2, ..., ir_n\}$ von Eingabe- und eine Menge $\{or_1, or_2, ..., or_m\}$ von Ausgaberegistern, die in Abbildung 7 als grau unterlegte Kreise dargestellt sind. Eine Kante von einem Eingaberegister ir_i zu der Transition *t* wird ungerichtet und unbeschriftet gezeichnet, während eine Kante von *t* zu einem der Ausgaberegister or_j als Pfeil dargestellt wird, der mit einem arithmetischen Ausdruck $\tau_j(ir_1, ir_2, ..., ir_n)$ in den Eingaberegistern der Transition beschriftet ist. Daneben existiert für jede Transition *t* ein Prädikat $\pi(ir_1, ir_2, ..., ir_n)$ in den Eingaberegistern der Transition.

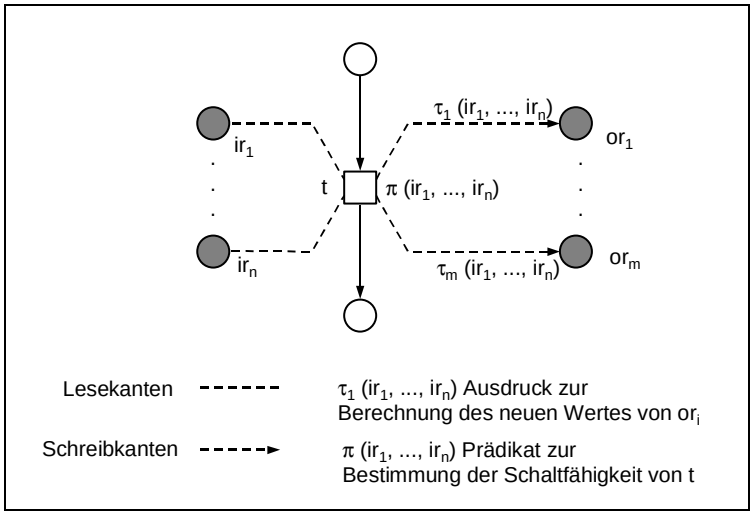


Abbildung 7 - Elemente eines Registernetzes

Um Kantenüberschneidungen zu vermeiden, nehmen wir uns die Freiheit, dasselbe Register mehr als einmal zu zeichnen. Insbesondere ordnen wir die Eingaberegister einer Transition meist links von ihr, ihre Ausgaberegister hingegen rechts von ihr an: Natürlich kann ein Register sowohl Ein- als auch Ausgaberegister einer Transition sein.

Die Schaltregel für Registernetze ist nun wie folgt definiert:

1. Ein Zustand eines Registernetzes ist ein Paar $z = (Q, \text{val})$, wobei *Q* eine Menge von Plätzen des unterliegenden Petri-Netzes und val: $R \rightarrow \mathbf{Nat}_\perp$ eine Abbildung ist. Dabei bezeichnet *R* die Menge der Register eines Registernetzes, **Nat** ist die Menge der natürliche Zahlen und $\mathbf{Nat}_\perp$ steht für $\mathbf{Nat} \cup \{\perp\}$.
2. Ist *t* eine Transition eines Registernetzes, so bezeichnen wir mit $\bullet t$ die Menge der Vorplätze und mit $t\bullet$ die Menge der Nachplätze von *t* (hierbei sind nur Plätze des unterliegenden Petri-Netzes gemeint, nicht die Register des Registernetzes).

3. Eine Transition *t* heißt *schaltfähig* einem Zustand $z = (Q, \text{val})$, wenn

   (a) $\bullet t \subseteq Q \;\&\; Q \cap t\bullet = \varnothing$ sowie

   (b.1) $\pi(\text{val}(\text{ir}_1), \text{val}(\text{ir}_2), ..., \text{val}(\text{ir}_n)) \neq \text{FALSE}$

   gilt; wir schreiben dann $z\,[t\rangle$.

4. Eine Transition t *schaltet* von einem Zustand $z_1 = (Q_1, \text{val}_1)$ in einen Zustand $z_2 = (Q_2, \text{val}_2)$, wenn $z_1\,[t\rangle$

   (a) $Q_2 = (Q_1 \setminus \bullet t) \cup t\bullet$ und

   (b.2) $\text{val}_2(r) = \tau_j(\text{val}_1(\text{ir}_1), \text{val}_1(\text{ir}_2), ..., \text{val}_1(\text{ir}_n))$ für $r = \text{or}_j$,

   andernfalls $\text{val}_2(\text{r}) = \text{val}_1(\text{r})$

   gilt; wir schreiben dann $z_1\,[t\rangle\, z_2$.

Die Bedingungen (a) in der Definition der Schaltfähigkeit und der Schaltrelation einer Transition entspricht hierbei den jeweiligen Bedingungen für sog. *Elementare Netzsysteme*; dabei handelt es sich um Petri-Netze, die bereits durch ihre besondere Schaltregel sicher sind.

Die Bedingung (b.1) erklärt, wie die Schaltfähigkeit einer Transition von den Registern des Registernetzes abhängt: Eine Transition ist danach schaltfähig bei einem gegebenen Zustand, wenn (a) gilt und außerdem das ihr zugeordnete Prädikat mit den Werten der Eingaberegister der Transition bei diesem Zustand entweder zu TRUE ausgewertet werden kann oder aber undefiniert ist (wir diskutieren diesen Punkt später). Bedingung (b.2) beschreibt die Wirkung des Schaltens einer Transition auf ihre Ausgaberegister; dessen neuer Wert ergibt sich durch Auswertung der arithmetischen Ausdrücke, die den entsprechenden Schreibkanten zugeordnet sind.

Abbildung 8 gibt ein Beispiel für eine solche Transition. *t* ist schaltfähig in einem Zustand $z_1 = (Q_1, \text{val}_1)$, wenn der Vorplatz von *t* in der Menge Q_1 enthalten ist und weiterhin $\text{val}_1(b) \leq \text{val}_1(a)$ ist. Wenn *t* schaltet, wird die Marke auf dem Vorplatz von *t* vernichtet, während eine andere Marke auf dem Nachplatz von *t* erzeugt wird. Dem Register c wird der Wert $\text{val}_1(a) - \text{val}_1(b)$ zugewiesen.

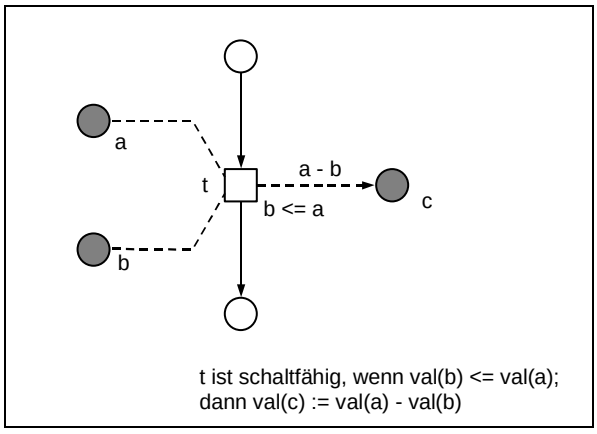


Abbildung 8 - Beispiel für eine Transition eines Registernetzes

Es sind einige Bemerkungen zur Definition der Schaltregel für Registernetze zu machen:

1. Datentypen, die durch die Norm [IEC 1131-3] für die Programmiersprache Anweisungsliste (AWL) zugelassen sind, umfassen skalare Datentypen wie int, uint16, byte, float oder bool. Eine Netzklasse zur Modellierung von AWL-Programmen muss die Darstellung all dieser Datentypen erlauben. Vom theoretischen Standpunkt ist es aber ratsam, die Anzahl der verwendeten Elemente in der Formulierung einer Theorie möglichst klein zu halten, um die Anzahl der Fälle, die bei Definitionen und Beweisen zu beachten sind, zu reduzieren. An dieser Stelle bietet sich eine ausgezeichnete Rationalisierungsmöglichkeit. Offenbar ist es nämlich möglich, diese Datentypen als Bitfolgen darzustellen, d. h. als natürliche Zahlen zu interpretieren. In welcher Form ein solches Bitmuster etwa als Fließkommazahl oder als logischer Wert zu interpretieren ist, soll den Prädikaten und Operationen obliegen, die als Transitions- und Kantenanschriften in einem Registernetz auftreten.

2. $\perp$ stellt den speziellen Wert „undefiniert“ dar. Etwa liefert der Ausdruck a / b bei einem Zustand $z = (Q, \text{val})$ mit val(b) = 0 den Wert $\perp$. Eine andere Anwendungsmöglichkeit für dieses Konzept ergibt sich, wenn Über- oder Unterläufe explizit modelliert werden sollen. Wenn als Datentyp für uint (unsigned int) 16-Bit-Werte zur Verfügung stehen, stellt sich bei der Ausführung der Operation $(2^{16} - 1) + 1$ ein Überlauf ein: Das Ergebnis der Operation ist 0. Obwohl dies den meisten Programmierern bekannt ist, werden solche Überläufe nur in den seltensten Fällen abgefragt und behandelt. Tatsächlich stellen viele Programmiersprachen nicht einmal entsprechende Konstanten zur Verfügung (die Sprachen der IEC 1131-3 zählen zu diesen Programmiersprachen!). Bei der Verifikation von Steuerungsprogrammen haben derartige Laufzeitfehler deshalb einen hohen Stellenwert.

   Die Schaltregel für Registernetze behandelt Transitionen auch dann als schaltfähig bei einem Zustand *z*, wenn sich bei der Auswertung des zugeordneten Prädikats der Wert $\perp$ ergibt. Das erscheint zunächst erstaunlich, da das Systemprogramm einer SPS etwa Divisionen durch Null dadurch abfängt, dass sich die SPS in einen Fehlerzustand begibt. Allerdings gilt dies nicht zwangsläufig auch für Über- oder Unterläufe. Die Behandlung dieser Ausnahmefälle ist nach IEC 1131-3 dem Hersteller des SPS-Systems überlassen.

   Wesentlicher ist jedoch, dass wir Registernetze nicht nur zur Modellierung von SPS-Programmen, sondern auch zur Repräsentation der von ihnen gesteuerten Anlage verwenden wollen; hier existiert so etwas wie ein Systemprogramm oder ein Betriebssystem nicht.

Ein paar Bemerkungen zur Ausdrucksstärke von Registernetzen sollen gemacht werden: Registernetze sind offenbar Turing-mächtig. Zum Beweis können die sog. Zählermaschinen verwendet werden, die ebenfalls als Turing-mächtig bekannt sind. Eine Zählermaschine umfasst eine endliche Menge von *Zuständen*, wobei es einen ausgezeichneten Startzustand gibt, und eine endliche Menge von *Zählern.* Zählern werden natürliche Zahlen zugewiesen, die *initiale* Zuweisung ist hierbei $c = 0$ für jeden Zähler *c*. Befindet sich die Zählermaschine nun in einem Zustand *z*, so erfolgt der Wechsel in einen anderen Zustand *z*’ in Abhängigkeit des Wert eines der Zähler; eine Abfrage $c = 0$ ist hier erlaubt. Erfolgt nun ein Zustandswechsel, kann der Wert eines Zählers um 1 vermindert oder erhöht werden. Eine weitere Form der Aktion ist die Stopp-Aktion, diese Anweisung hält die Zählermaschine an.

Anders gesagt: Eine Transition im Zustand z einer Zählermaschine lässt sich als Anweisung

z: **if** $c = 0$ **then** $c_1 = c_1 \pm 1$; **goto** z_1 **else** $c_2 = c_2 \pm 1$; **goto** z_2 **fi** bzw.
z: **stop**

darstellen. Eine Zählermaschine kann durch eine endliche Folge solcher Anweisungen beschrieben werden. Es ist nun offensichtlich möglich, jede Zählermaschine in ein Registernetz zu überführen.

Andererseits ist die Welt der SPS-Programmierung endlich; Datentypen sind auf die Repräsentation durch 8, 16 oder 32 Bit beschränkt. Wir verwenden deshalb *beschränkte Registernetze*, das sind solche Registernetze, für die für jedes Register r und für jeden Zustand $z = (Q, \text{val})$, der bei der Ausführung eines Registernetzes erreicht werden kann, eine Konstante $K > 0$ existiert, so dass $\text{val}(r) \neq \perp \Rightarrow \text{val}(r) < K$ gilt. Solche beschränkten Registernetze können nun in verschiedenartiger Weise in sichere Platz/Transitionsnetze überführt und mit Mitteln der klassischen Petri-Netz-Theorie analysiert werden (s. [Deussen2001] für eine Diskussion).

2.3 Überführung von AWL-Programmen in Registernetze

Abbildung 9 zeigt Registernetzfragmente für einige der Sprachelemente von AWL. AWL ist eine assemblerartige Programmiersprache. Anweisungen beziehen sich auf einen Akkumulator (der als ein Register acc modelliert wird) und eine symbolische Adresse (Variable), einen numerischen Wert bzw. eine Sprungmarke. Variablen können Eingabevariablen, Ausgabevariablen oder interne Variablen sein: Eingabevariablen werden durch das Systemprogramm einer SPS Sensorwerte zugewiesen, Ausgabevariablen werden auf Aktuatorwerte abgebildet.

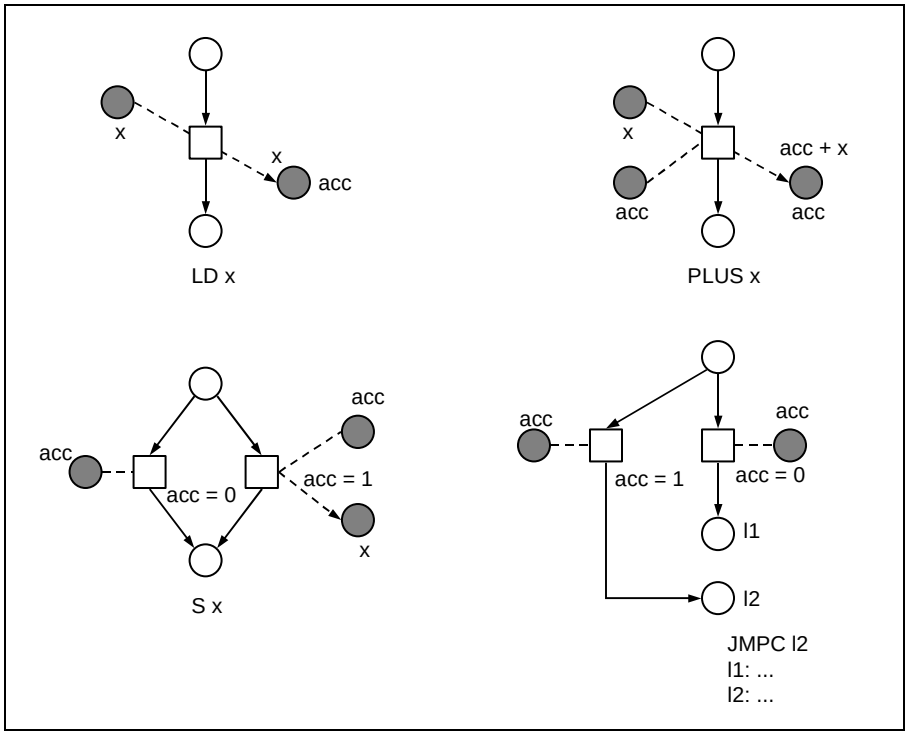


Abbildung 9 - Registernetzstrukturen für einige AWL-Primitive

Die Modellierung der Anweisungen LD (load) und PLUS (Addition) als Registernetz ist offensichtlich, auch die Modellierung der JMPC (conditional jump) Anweisung birgt keine

Überraschungen. Bei der Anweisung S (set) handelt es sich um eine bedingte Anweisung: Der Variablen x wird durch die Anweisung S x nur dann der (logische) Wert 1 zugewiesen, wenn acc ebenfalls den logischen Wert 1 hat. Wir verwenden deshalb zwei Transitionen mit entsprechenden Prädikaten zur Modellierung dieser Anweisung.

Es existieren weitere Sprachelemente in AWL, neben einer Reihe weiterer elementarer Anweisungen insbesondere auch Funktionen und Funktionsblöcke sowie Timer. Funktionen und Funktionsblöcke stellen Unterprogramme dar; der Unterschied ist, dass Funktionsblöcke im Gegensatz zu Funktionen Variablen enthalten können, deren Lebensdauer die Laufzeit dieses Funktionsblocks überschreitet und deren Werte somit bei einem folgenden Aufruf wieder zur Verfügung stehen (C-Programmierer erkennen solche Variablen als „static“). Da der Arbeitsspeicher vieler SPS'en stark eingeschränkt ist und darüber hinaus einheitliche Abtast- bzw. Zykluszeiten gewährleistet sein sollen, ist es durch die IEC 1131-3 untersagt, Funktionen und Funktionsblocke direkt oder indirekt rekursiv zu formulieren. Das besagt aber, dass AWL-Programme mit Funktionen bzw. Funktionsblöcken durch das textuelle Einfügen ihres Programmcodes an ihrer Aufrufstelle in ein AWL-Programm ohne Funktionen und Funktionsblöcke überführt werden kann. Bei unserer Übersetzung von AWL-Programmen in Registernetze beachten wir solche Strukturierungselemente deshalb nicht weiter.

Problematischer sind dagegen Timer. Zwar stehen Petri-Netzklassen zumindest in der Theorie zur Verfügung, die die adäquate Modellierung von Timern erlauben - und die Anreicherung von Registernetzen um ein entsprechendes Zeitkonzept ist eine Übung für Studenten. Für solche zeitbewerteten Petri-Netzklassen stehen jedoch keine in der Praxis ausreichend effizienten Analysetechniken zur Verfügung. Wir verzichten deshalb auf die Möglichkeit, AWL-Programme mit Timern in Petri-Netze zu überführen und damit zu verifizieren.

Wie bereits gesagt, verwenden wir Registernetze nicht nur zur Darstellung von SPS-Anwenderprogrammen, sondern auch zur Modellierung der gesteuerten Anlage; das resultierende Registernetz bezeichnen wir als *Umgebungsmodell*. Das Registernetz für das System SPS-Umgebung entsteht, indem das Umgebungsmodell, das Registernetz für das Anwenderprogramm und das Registernetz für das Systemprogramm der SPS zusammengenommen werden; das entstehende Gesamtmodell bezeichnen wir als *Kontrollmodell*. Um zu illustrieren, wie ein solches Umgebungsmodell erstellt werden kann und wie die Kopplung mit dem Steuerungsmodell zum Kontrollmodell erfolgt, ist in Abbildung 10 ein Fließband mit zwei Sensoren (Lichtschranken) S1 und S2 und einem Motor A abgebildet.

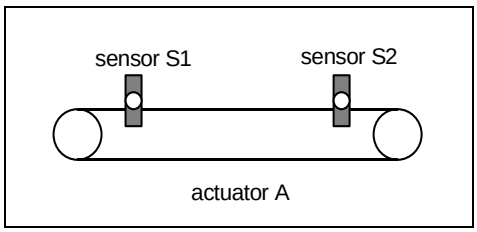


Abbildung 10 - Beispiel: Förderband mit zwei Lichtschranken

Die Abbildung 11 zeigt ein Registernetz, welches das Verhalten dieses Förderbandes modelliert.

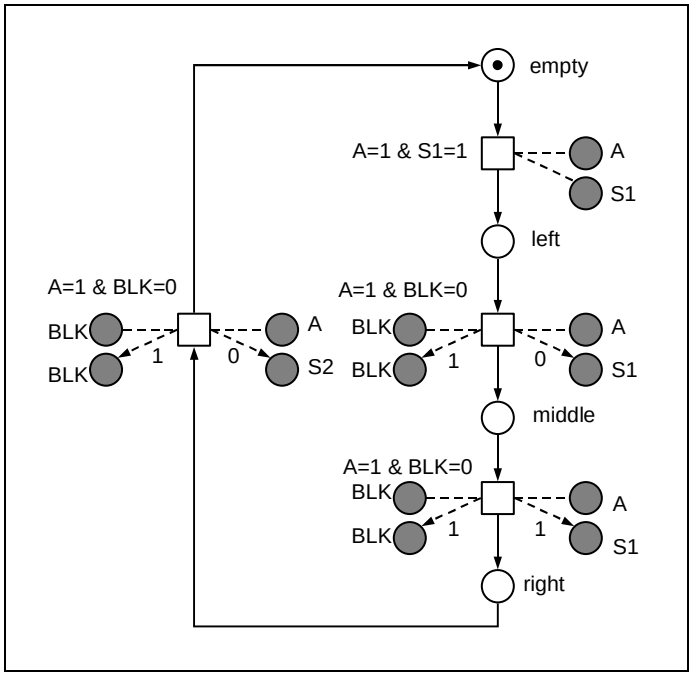


Abbildung 11 - Modell des Förderbandes aus Abbildung 10

Eine Marke auf dem Platz *empty* ist so zu interpretieren, dass sich kein Werkstück auf dem Fließband befindet. Ist der Motor des Fließbands aktiv (A = 1) und liefert der Sensor S1 den Wert 1, so wurde ein Werkstück auf das Fließband im Bereich der Lichtschranke S1 (*left*) abgelegt. Bleibt der Motor aktiv, wird dieses Werkstück in den mittleren Teil des Fließbands aus dem Bereich der Lichtschranke S1 heraus (*middle*) transportiert; dabei wird die Lichtschranke S2 noch nicht angesprochen. Dies geschieht jedoch, wenn der Motor weiterhin aktiv bleibt: Das Werkstück wird in den rechten Teil des Fließbands transportiert. Schließlich (wiederum unter der Voraussetzung A = 1) verlässt das Werkstück das Fließband.

Das Netzmodell des Fließbands enthält noch ein weiteres Register BLK, dessen Aufgabe nun erklärt wird. Bei der Auswahl einer SPS für einen konkreten technischen Prozess und deren anschließender Programmierung ist eine Synchronisationsannahme zu beachten:

> Die Zykluszeit einer SPS ist ausreichend kurz, so dass die SPS jede Änderung von Sensorwerten in der gesteuerten Anlage registriert.

Der Leser, dem diese Annahme unrealistisch erscheint, überlege sich ein SPS-Programm, das unabhängig von dieser Annahme ist. Man beachte hierbei, dass Fehlerzustände für die SPS ebenfalls nur aufgrund von Sensorwerten erkennbar sind.

Um diese Vorgabe in unserer Modellierung zu berücksichtigen, führen wir ein Register BLK für jede SPS in der zu analysierenden Anlage ein. Die Schaltfähigkeit einer jeden Transition *t*, die Register modifiziert, die Sensorwerte repräsentieren, wird von BLK abhängig gemacht: *t* ist nur dann schaltfähig, wenn BLK = 0 gilt; wenn *t* schaltet, wird BLK = 1 gesetzt, d. h. jede Transition, die Sensorregister modifiziert, ist nun blockiert. Diese Blockierung wird von dem Netzmodell des Systemprogramms (Abbildung 12) nach Abarbeitung des Anwenderprogramms und der Abbildung der Ausgabevariablen auf Aktuatorwerte aufgehoben. Natürlich kann die SPS dann erneut in einen Zyklus eintreten, bevor es zu Veränderungen der Sensorwerte im Umgebungsmodell kommen kann. In diesem Fall wird aber das Umgebungsmodell zunächst wiederum blockiert.

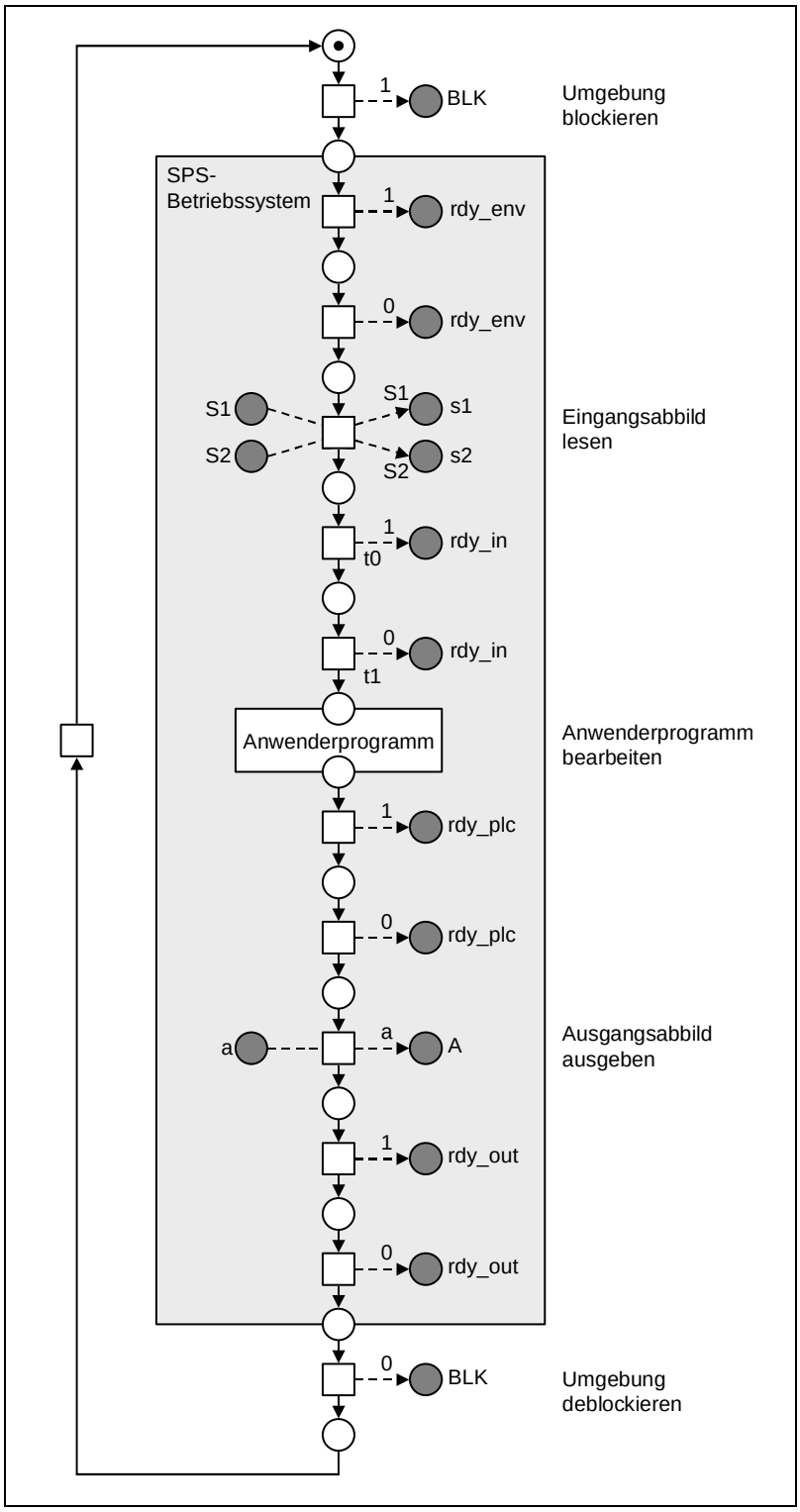


Abbildung 12 - Modell des SPS-Systemprogramms

Die Kopplung der verschiedenen Modelle erfolgt nun, indem die im Netzmodell des Systemprogramms auftretenden Register für Sensoren und Aktuatoren mit den korrespondierenden Registern im Umgebungsmodell verschmolzen werden; ebenso werden die BLK-Register miteinander assoziiert. Das Anwenderprogramm wird an der bezeichneten Stelle im Systemprogrammmodell eingefügt.

3. Formale Spezifikation mit der Sicherheitsfachsprache

In diesem Kapitel wird gezeigt, welche Anforderungen an ein Steuerungsprogramm gestellt werden und wie diese kategorisiert werden können.

Die Spezifikation, also die technologischen und technischen Vorgaben für die gewünschte Funktionsweise einer Maschine oder Anlage, wird üblicherweise in Kooperation zwischen Auftraggeber und Softwareentwickler erstellt und in einem Pflichtenheft schriftlich festgehalten. Die Darstellung erfolgt gewöhnlich als freie verbale Beschreibung der Anforderungen oder mit anderen spezifischen Darstellungsformen.

Das Pflichtenheft zu einem Steuerungsprogramm [VDI/VDE 3694] muss alle Informationen enthalten, die für die vollständige, eindeutige und korrekte Beschreibung aller Funktionen des Programms notwendig sind. Zu den Bestandteilen eines Pflichtenheftes gehört neben einer verbalen Kurzbeschreibung des Programms die Darstellung der verfügbaren Input-Parameter (Sensoren, Bedienereingaben) und die Beschreibung der notwendigen Output-Parameter (Aktoren, Anzeigen, Ausgaben usw.). Die Funktionsbeschreibung soll dabei allgemeinverständlich aber auch übersichtlich sein. Zu diesem Zweck haben sich in der Vergangenheit verschiedene graphische Darstellungsformen, wie Funktionsdiagramme, Funktionspläne und auch Petri-Netze, als praktikabel erwiesen.

Das Pflichtenheft umfasst primär die gewünschten *Funktionsanforderungen* eines Systems, d. h. die Funktionen, die während des normalen, standardmäßigen Betriebs des Systems auftreten (*Betriebsfunktionen*). Darüber hinaus zählen zu den Funktionsanforderungen aber auch die Reaktionen des Programms auf ungewöhnliche, außerplanmäßige Ereignisse, wie Störungen oder Not-Aus (*Störungsfunktionen*). Während der Normalbetrieb eines Systems noch relativ einfach beschrieben werden kann, müssen für die Beschreibung der Programmreaktionen auf anormale Ereignisse weitaus mehr globale Systemzusammenhänge betrachtet werden. Betriebsfunktionen werden während des Normalbetriebs ständig durchlaufen, Störungsfunktionen werden nur im Ausnahmefall aktiviert, für sie ist ein Test oder eine Prüfung in der realen Anlage meist nicht mehr möglich. Eine gemeinsame Eigenschaft der Funktionsanforderungen besteht darin, dass bestimmte Ereignisse und Zusammenhänge eine eindeutige Reaktion innerhalb einer (möglichst minimalen) Reaktionszeit verlangen.

Das Pflichtenheft beinhaltet neben den notwendigen Abläufen des Programms jedoch auch Festlegungen, welche Programmreaktionen unbedingt vermieden werden müssen. Diese *Sicherheitsanforderungen* lassen sich jedoch meist nicht so einfach wie die bereits erwähnten Funktionseigenschaften beschreiben. Es ist oft problematisch, die kritischen Situationen eines Steuerungssystems vollständig zu identifizieren, bzw. die komplexen Einflussfaktoren und Wechselwirkungen der Systeme untereinander zu erkennen. Das maßgebliche Problem besteht jedoch darin, dass es für die Darstellung auszuschließender Reaktionen eines Steuerungssystems keine ingenieurtechnischen Darstellungsformen gibt.

3.1 Typische steuerungstechnische Anforderungen

Die festgelegten Vorgaben an das Steuerungsprogramm werden im Folgenden allgemein als *Anforderungen* (engl. *requirements*) bezeichnet. In der Einleitung wurde bereits eine grobe Einteilung dieser Anforderungen in Funktions- und Sicherheitsanforderungen dargelegt. Bei der Formulierung dieser Anforderungen werden bestimmte Zustände des Systems betrachtet, es werden bestimmte Aussagen über diese Systemzustände getroffen und es werden gewisse Zusammenhänge zwischen verschiedenen Systemzuständen hergestellt.

Funktionsanforderungen zeichnen sich dadurch aus, dass sie zunächst einen bestimmten Ausgangszustand beschreiben, von dem aus ein anderer Systemzustand (Zielzustand) erreicht werden muss. Der Ausgangs- und der Zielzustand unterscheiden sich dabei mindestens um den Wert einer einzelnen Systemvariablen (z. B. die Aktivität eines Motors). Ein typisches Beispiel für eine solche Funktionsanforderung ist, dass bei Betätigung eines bestimmten Tasters durch den Anlagenbediener unverzüglich ein Motor einzuschalten ist.

Diese genannten Anforderungen werden weiterhin als *Forderungen* (engl. *demands*) bezeichnet. Sie beschreiben die notwendigen und erforderlichen Reaktionen des Steuerungsprogramms auf festgelegte Ereignisse.

Forderungen lassen sich weiterhin hinsichtlich ihres zeitlichen Horizonts betrachten. Man kann auf diese Weise kurzfristige und langfristige Programmanforderungen beschreiben. Unter kurzfristigen Anforderungen kann man solche verstehen, die eine möglichst schnelle Reaktion des Steuerungsprogramms auf das Eintreten bestimmter Ereignisse beschreiben. Ein Beispiel hierfür ist das notwendige Abschalten eines Motors, sobald ein durch ihn bewegtes Maschinenelement einen bestimmten Endlagenschalter erreicht hat und somit ein Signal ausgelöst wurde. Bei langfristigen Anforderungen ist der Zeitpunkt für das Erreichen der beschriebenen Reaktion nicht exakt definiert. Hierbei werden übergeordnete Systemziele definiert, so z. B., dass ein Werkstück, das in eine Fertigungszelle gelangt, diese auch irgendwann wieder verlassen muss.

Bei Sicherheitsanforderungen werden üblicherweise einzelne Zustände des Systems beschrieben, die innerhalb der Betrachtungszeit des Systems niemals erreicht werden dürfen. Beispiele hierfür sind Zustände, die zu einer Gefährdung von Personen, der Umgebung, von Material oder der Maschine selbst führen können. So könnte für eine betrachtete Maschine festgelegt sein, dass es niemals eine Situation geben darf, in der eine Sicherheitsvorrichtung geöffnet ist und gleichzeitig eine Maschinenbewegung ausgeführt wird.

Sicherheitsanforderungen werden im Folgenden als *Verbote* (engl. *prohibitions*) bezeichnet. Sie beschreiben Zustände des Steuerungsprogramms, die nicht eintreten dürfen, bzw. verbotene Reaktionen auf festgelegte Ereignisse.

Es erweist sich als brauchbar und während der Software-Entwicklungsphase als durchaus üblich, eine weitere Kategorie von Anforderungen zu definieren. So gibt es Situationen, in denen eine bestimmte Reaktion des Steuerungsprogramms zwar erlaubt ist, aber noch nicht ausgeführt werden darf, weil eine andere, zusätzliche Bedingung noch nicht erfüllt ist. Eine solche typische Freigabesituation ist gegeben, wenn z. B. eine Maschinenbewegung ausgeführt werden darf, wenn eine Sicherheitsvorrichtung geschlossen ist, die Aktion aber erst dann erfolgen soll, wenn gleichzeitig auch ein bestimmter Taster betätigt wurde.

Diese Anforderungen werden im Folgenden als *Möglichkeiten* (engl. *possibilities*) bezeichnet. Sie beschreiben erlaubte Reaktionen des Steuerungsprogramms auf festgelegte Ereignisse.

3.2 Verwendung spezieller Schlüsselwörter

Die im vorherigen Abschnitt definierten Anforderungskategorien zeichnen sich in ihrer Anwendung bei der verbalen Formulierung der Programmanforderung durch die Verwendung bestimmter sprachlicher Konstruktionen und typischer Schlüsselwörter aus.

Kategorie	Beispiele
Forderungen	*muss, müssen, soll, sollen, ist zu ..., sind zu ...*
Verbote	*darf nicht, soll nicht*
Möglichkeiten	*darf, dürfen, kann, können*

Tabelle 2 - Schlüsselwörter zur Formulierung der Anforderungskategorien

Weiterhin lassen sich bei der Analyse der verbalen Formulierungen Schlüsselwörter identifizieren, die bestimmte zeitliche, logische oder inhaltliche Zusammenhänge zwischen verschiedenen Aussagen herstellen.

Kategorie	Beispiele
Zeitliche Zusammenhänge	*nach, nachdem, danach, vor, bevor, zuerst, sofort, dabei, bis, solange, nie, niemals*
Logische Zusammenhänge	*und, oder, nicht*
Bedingungen, Konditionale	*wenn ... dann ..., falls ...*
Erweiterte Bedingungen, Bikonditionale	*nur dann wenn ..., genau dann wenn ...*

Tabelle 3 - Auflistung weiterer Schlüsselwörter

Das Grundmuster einer Anforderung, die mit der Sicherheitsfachsprache formuliert wird, ist das Konditional. Ein einzelner Satz besteht somit aus zwei Teilen, diese sind durch ein Komma voneinander getrennt. In einem Teilsatz wird eine Bedingung oder ein bestimmtes Ereignis formuliert (erkennbar am Schlüsselwort „*Wenn* ...“). Dieser Satzteil wird im Folgenden als *Bedingung B* bezeichnet. Im anderen Teilsatz wird eine zugehörige Reaktion formuliert (erkennbar am Schlüsselwort „*dann* ...). Dieser Teilsatz wird als *Folgerung F* bezeichnet. Die Reihenfolge von Bedingung und Folgerung wird zunächst nicht festgelegt.

> Eine Anforderung setzt sich aus einer *Bedingung B* und einer *Folgerung F* zusammen.
>
> Zwischen einer Bedingung und einer Folgerung besteht ein implikativer Zusammenhang.

Verbote, die lediglich einzelne Zustände des Systems beschreiben, die nie erreicht werden dürfen, können ebenfalls auf die gezeigte Satzstruktur überführt werden.

3.3 Definition des Gültigkeitsbereichs einer Anforderung

Es wurde bereits festgestellt, dass es drei grundsätzliche Kategorien von Anforderungen an ein Steuerungsprogramm gibt:

a) *Forderungen* (demands - DE) beschreiben notwendige und erforderliche Reaktionen des Steuerungsprogramms auf festgelegte Ereignisse,

b) *Verbote* (prohibitions - PR) beschreiben untersagte Reaktionen des Steuerungsprogramms unter bestimmten Umständen,

c) *Möglichkeiten* (possibilities - PO) beschreiben erlaubte Reaktionen des Steuerungsprogramms unter bestimmten Umständen.

Eine Anforderung setzt sich aus einer Bedingung und einer Folgerung zusammen. Betrachtet man eine Bedingung und eine Folgerung unter Berücksichtigung des erzeugten System- bzw. Kontrollmodells, so lassen sich innerhalb des Zustandraumes des Systems einzelne oder mehrere Zustände identifizieren, bei denen die Werte der betrachteten Variablen mit den in den Bedingungen und Folgerungen definierten Variablenwerten übereinstimmen. Man spricht dann davon, dass diese Zustände die jeweilige Bedingung bzw. Folgerung erfüllen.

Systemzustände, die eine Bedingung erfüllen, werden als *Startzustand S* bezeichnet.

Systemzustände, die eine Folgerung erfüllen, werden als *Zielzustand Z* bezeichnet.

Es wird ein *Beobachtungsintervall BI* definiert, das mit einem *Startzustand* beginnt und eine momentan nicht näher definierte Länge aufweist. Ein Beobachtungsintervall umfasst eine bestimmte Anzahl zeitlich aufeinanderfolgender Systemzustände.

Die Klassifizierung der Anforderungen an ein Steuerungsprogramm erfolgt durch zwei Parameter:

1. der *Modalparameter* gibt die logische Bedeutung der Anforderung an, er stellt einen modalen Zusammenhang zwischen einem Zielzustand und einem Beobachtungsintervall her,
2. der *Zeitparameter* gibt die Länge eines Beobachtungsintervalls, d. h. die zeitliche Reichweite einer Anforderung an.

3.4 Ableitung des Modalparameters

Die Tabelle 4 stellt die modalen Zusammenhänge zwischen dem Zielzustand Z (also der Erfüllung einer Folgerung) und dem Beobachtungsintervall BI dar. Dabei erfolgt eine Betrachtung über die Existenz eines Zielzustandes sowohl innerhalb als auch außerhalb des Beobachtungsintervalls.

		Innerhalb des Beobachtungsintervalls (BI)					
		Forderung (muss)		**Möglichkeit (darf)**		**Verbot (darf nicht)**	
Außerhalb des Beobachtungsintervalls	**Forderung**	**I**	innerhalb des BI: es muss ein Z geben außerhalb des BI: es muss ein Z geben	**II**	innerhalb des BI: es darf ein Z geben außerhalb des BI: es muss ein Z geben	**III**	innerhalb des BI: es darf kein Z geben außerhalb des BI: es muss ein Z geben
	Möglichkeit	**IV**	innerhalb des BI: es muss ein Z geben außerhalb des BI: es darf ein Z geben	**V**	innerhalb des BI: es darf ein Z geben außerhalb des BI: es darf ein Z geben	**VI**	innerhalb des BI: es darf kein Z geben außerhalb des BI: es darf ein Z geben
	Verbot	**VII**	innerhalb des BI: es muss ein Z geben außerhalb des BI: es darf kein Z geben	**VIII**	innerhalb des BI: es darf ein Z geben außerhalb des BI: es darf kein Z geben	**IX**	innerhalb des BI: es darf kein Z geben außerhalb des BI: es darf kein Z geben

Tabelle 4 - Matrix aus Modalkategorie und abstraktem Beobachtungsintervall

Zu erkennen sind neun Kombinationstypen, von denen einige (grau unterlegt) jedoch nicht relevant bzw. untereinander redundant sind.

Bei den Kombinationen I, V und IX ist die Existenz des Zielzustandes in der gewählten Modalkategorie unabhängig vom Beobachtungsintervall. Der Zielzustand ist also unabhängig von einer aufgestellten Bedingung, d. h. die Folgerung würde im gesamten Zustandsraum des betrachteten Systems gültig sein. Die Kombinationen I, V und IX entfallen somit, weil sie nicht dem vorgegebenen Aufbau einer Anforderung (Bedingung und Folgerung) entsprechen. Andererseits können sie im Weiteren auch auf andere Kombinationstypen abgebildet werden (die Bedingung wäre dann generell TRUE).

Die Kombinationen II und IV, III und VII bzw. VI und VIII können durch Negation der Beobachtungsintervalle auf die jeweils andere Kombination abgebildet werden, sie sind also redundant. Aus diesem Grund entfallen die Kombinationen II und III. Die Kombination VI wird jedoch trotzdem beibehalten, da sich hier später auf sprachlicher Ebene unterschiedliche Formulierungen ergeben werden.

Nach Abzug der ausgeschlossenen Kombinationen bleiben lediglich vier Varianten übrig, diese stellen die „automatisierungstechnisch relevanten“ Anforderungen an Steuerungsprogramme dar.

IV - Einfache Forderung – DEs (simple demand)

VI - Einfaches Verbot – PRs (simple prohibition)

VII - Erweiterte Forderung – DEe (extended demand)

VIII - Erweiterte Möglichkeit – POe (extended possibility)

3.5 Ableitung des Zeitparameters

Neben der Kategorisierung der SPS-Anforderungen hinsichtlich ihrer Bedeutung besteht ein weiteres Unterscheidungsmerkmal innerhalb dieser Kategorien in der *Reichweite* der jeweiligen Aussage. Mit ihr wird spezifiziert, über welchen Zeitraum die entsprechende Anforderung gültig sein soll.

Es wurde bereits ein Beobachtungsintervall definiert, in dem je nach Modalparameter ein festgelegter Zielzustand entweder existieren muss, existieren darf oder nicht existieren darf. Das Beobachtungsintervall beginnt immer mit einem Startzustand und besteht aus einer Menge aufeinanderfolgender Zustände. Die Länge eines Beobachtungsintervalls, d. h. die Anzahl der aufeinanderfolgenden relevanten Zustände, wird durch den Zeitparameter festgelegt.

Ein *Beobachtungsintervall BI* beginnt mit einem *Startzustand S* und endet mit einem *Endzustand E*.

Typische automatisierungstechnische Anforderungen an ein Steuerungsprogramm beziehen sich auf zwei Zeitebenen: tritt auf eine Bedingung die zugehörige Reaktion innerhalb einer vorgegebenen (möglichst kurzen) Zeitspanne ein (kurzfristige Anforderungen) bzw. tritt auf eine Bedingung die zugehörige Reaktion überhaupt ein (langfristige Anforderungen)?

Ausgehend von dieser Tatsache lassen sich zwei Hauptgruppen mit insgesamt fünf Varianten von Beobachtungsintervallen differenzieren. Die beiden ersten Varianten beziehen sich auf die typische Arbeitsweise einer Speicherprogrammierbaren Steuerung. Der Zeitpunkt des Endzustandes und somit auch die Länge des Beobachtungsintervalls ist an den SPS-Zyklus gekoppelt.

- *Zustand*: das Beobachtungsintervall ist auf alle Zustände begrenzt, die im selben SPS-Zyklus liegen,
- *Direkt*: das Beobachtungsintervall ist auf alle Zustände beschränkt, die im selben und im direkt nachfolgenden SPS-Zyklus liegen,

Die Länge der folgenden drei Varianten der Beobachtungsintervalle ist im Gegensatz zu den beiden vorherigen nicht starr festgelegt. Der Zeitpunkt des Endzustandes wird durch den Bedingungsteil der Anforderung gegeben

- *Selbstbegrenzt*: Der Startzustand ist hierbei wie üblich mit der Erfüllung der Bedingung gegeben ($S := B$). Der Endzustand ist gegeben, sobald die definierte Bedingung erstmals nicht mehr erfüllt ist ($E := \neg B$).
- *Fremdbegrenzt*: Das Beobachtungsintervall wird vom Startzustand eingeleitet ($S := B_1$), der Endzustand ist durch die Erfüllung einer zusätzlichen Bedingung gegeben ($E := B_2$). Die Bedingung B_1 muss dabei innerhalb des Beobachtungsintervalls nicht unbedingt wahr bleiben.
- *Unbegrenzt*: Das Beobachtungsintervall wird ebenfalls vom Startzustand eingeleitet ($S := B$), hat jedoch kein vorgegebenes Ende. Es handelt sich hierbei um ein offenes Intervall.

Die Abbildung 13 soll diese fünf Varianten des Beobachtungsintervalls noch einmal verdeutlichen.

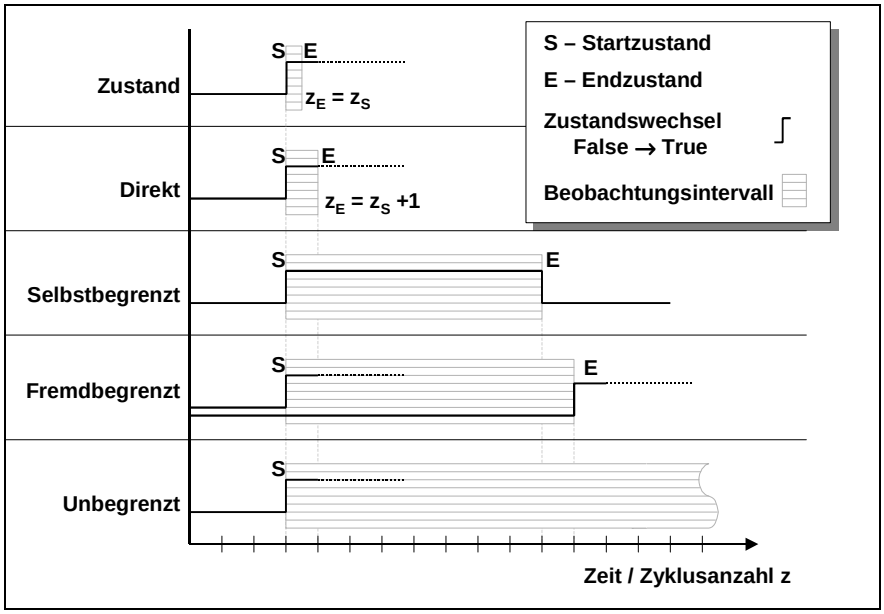


Abbildung 13 - Definition der Beobachtungsintervalle

3.6 Anforderungsmatrix

Für die Zusammenstellung der verschiedenen Arten von Anforderungen an ein Steuerungsprogramm können nun der Modalparameter und der Zeitparameter in einer Matrix vereinigt werden.

Einfache Forderung	**Einfache Möglichkeit**	**Einfaches Verbot**
DEs1 - Zustand		PRs1 - Zustand
DEs2 - Direkt		
DEs3 - Selbstbegrenzt		PRs3 - Selbstbegrenzt
DEs4 - Fremdbegrenzt		PRs4 - Fremdbegrenzt
DEs5 - Unbegrenzt		PRs5 - Unbegrenzt
Erweiterte Forderung	**Erweiterte Möglichkeit**	**Erweitertes Verbot**
DEe1 - Zustand	POe1 - Zustand	
DEe2 - Direkt		
DEe3 - Selbstbegrenzt	POe3 - Selbstbegrenzt	
DEe4 - Fremdbegrenzt	POe4 - Fremdbegrenzt	
DEe5 - Unbegrenzt	POe5 - Unbegrenzt	

Tabelle 5 - Matrix aus automatisierungstechnisch relevanten Kategorien und konkreten Beobachtungsintervallen

Die nicht-relevanten Kategorien wurden wiederum grau hinterlegt. Da die Formulierung von direkten Beobachtungsintervallen nur für Forderungen sinnvoll ist, werden letztendlich durch den Modal- und den Zeitparameter insgesamt 18 Anforderungskategorien definiert.

Diese 18 Anforderungskategorien unterscheiden sich jedoch auch stark hinsichtlich ihrer Relevanz bzw. ihren Einsatzmöglichkeiten bei der Spezifikation eines Steuerungsprogramms. Sie umfassen neben den bereits dargestellten Anforderungen aus der Automatisierungstechnik auch Eigenschaftsklassen, wie sie in der Informatik bekannt und üblich sind. Zu diesen gehören z. B. Fortschritts,- Lebendigkeits- und Sicherheitseigenschaften, die sich alle in der Matrix nach Tabelle 5 wiederfinden lassen.

Thematik	**Darstellung mit SFS**	**Bezeichnung in der Informatik**
Beschreibung des Sollverhaltens der Steuerung - kurzfristig	DEs1, DEs2, DEe1, DEe2	Fortschrittseigenschaften
Beschreibung des Sollverhaltens der Steuerung - langfristig	DEs3 ... DEs5, DEe3 ... DEe5	Lebendigkeitseigenschaften
Beschreibung auszuschließender Reaktionen	PRs1 ... PRs5, (POe1 ... POe5)	Sicherheitseigenschaften

Tabelle 6 - Nutzung der einzelnen SFS-Kategorien

3.7 Bildung von Verbalphrasen durch Interpretation der SPS-Variablen

Bei der Formulierung einer Anforderung mit der Sicherheitsfachsprache wird durch die Auswahl des Modal- und Zeitparameters ein bestimmtes Satzmuster festgelegt (siehe Anhang 0). Die noch bestehenden Freiräume innerhalb des Bedingungs- und des Folgerungsteils dieser Satzmusters müssen anschließend mit applikationsspezifischen *Verbalphrasen* ausgefüllt werden. Bei Betrachtung der Grundstruktur einer beliebigen Anforderungskategorie besteht diese zunächst aus dem konditionalen Grundmuster (Konjunktion aus zwei Teilsätzen) und verschiedenen allgemeinen Schlüsselwörtern (Zeitadverbien, Modalverben). Zur Vervollständigung des Satzes fehlen - gemäß den Grammatikregeln der deutschen Sprache - weitere typische Elemente, wie z. B. *Subjekte* für die Teilsätze und die zugehörigen *Prädikate.*

„Wenn ... ist, dann muss gleichzeitig ... sein.“

Diese fehlenden Satzbausteine müssen durch den Benutzer der Sicherheitsfachsprache durch Interpretation der gegebenen SPS-Variablen und deren relevanten Werten gebildet werden. Der Variablendeklarationsteil des gegebenen SPS-Programms mit den dort hinterlegten Informationen über Aktoren, Sensoren und interne Variablen bildet die Grundlage für die Erstellung der Verbalphrasen.

Eine *Verbalphrase* besteht immer aus zwei Teilen: einem *Nomen* und einem *Wert*. Man kann eine Verbalphrase deshalb auch als eine *Nomen-Wert-Kombination* bezeichnen. Das *Nomen* wird dabei durch Interpretation einer SPS-Variablen, der *Wert* durch Interpretation eines bestimmten Wertes, den diese SPS-Variable annehmen kann, gebildet. Da SPS-Variablen typischerweise mehrere Werte annehmen können, gibt es zu einem *Nomen* immer auch mehrere *Werte* und somit auch immer mehrere *Nomen-Wert-Kombinationen* zu einer SPS-Variablen. So kann z. B. zu der booleschen SPS-Variablen ‚x_1’, die einen bestimmten Taster repräsentiert, das *Nomen „der Start-Taster“* erstellt werden, für den ‚TRUE’-Zustand kann der *Wert „gedrückt“* und der ‚FALSE’-Zustand der *Wert „nicht gedrückt“* vereinbart werden. Für diese Variable ergeben sich die beiden *Nomen-Wert-Kombinationen*:

x_1 - *„der Start-Taster“* ist *„gedrückt“*
¬ x_1 - *„der Start-Taster“* ist *„nicht gedrückt“*

Diese Zuordnung ist für alle verwendeten Variablen einmal zu erstellen. Bei der anschließenden Formulierung der Anforderungen werden nur noch diese erzeugten *Nomen* und *Werte* benutzt, ohne dass sich der Anwender ständig über die dahinterliegenden steuerungstechnischen Gegebenheiten im Klaren sein muss. Bei der Vervollständigung der Anforderungssätze werden die *Nomen* als Subjekte innerhalb der Teilsätze verwendet, die *Werte* bilden zusammen mit den bereits vorhandenen Hilfsverben (‚*ist*’, ‚*sein*’, ‚*werden*’) die zugehörigen Prädikate. Als *Wert* und damit auch als Prädikat können dabei nicht nur einzelne Wörter dienen, hier sind - je nach Problemstellung und Initiative des Anwenders - auch größere Zusammensetzungen möglich: Verbindungen von weiteren Verben, Substantiven und Adjektiven, Adverbien, Umstandsbestimmungen, Vergleichsoperatoren usw.

Die Erstellung der *Nomen* und *Werte* ist ein Prozess, der eine gewisse Kreativität und Weitsicht des Anwenders der Sicherheitsfachsprache verlangt. Hierbei sind verschiedene Randbedingungen zu beachten, damit einerseits die bestehenden Restriktionen der SFS eingehalten werden, anderseits eine möglichst freie und natürliche Formulierung möglich ist. In den folgenden Absätzen werden einige Informationen und Anregungen gegeben, wie die erforderlichen *Nomen* und *Werte* zu erstellen sind.

Einsatz der Nomen-Wert-Kombinationen in Bedingungen bzw. Folgerungen

Die erzeugten *Nomen-Wert-Kombinationen* werden sowohl bei der Formulierung des Bedingungs- als auch des Folgerungsteils der Anforderungssätze verwendet. Als Besonderheit ist zu erwähnen, dass SPS-Variablen, die Sensoren repräsentieren, nur innerhalb von *Bedingungen* sinnvoll verwendet werden können. Aktor-Variablen werden dagegen typischerweise für die Darstellung der *Folgerungen* verwendet, können aber auch in *Bedingungen* eingesetzt werden. SPS-interne Variablen werden in *Bedingungen* und in *Folgerungen* gleichermaßen verwendet. Es erweist sich aus diesem Grund als sinnvoll und auch notwendig, für Aktor- und für interne Variablen zusätzliche *Werte* zu definieren, um den veränderten Gegebenheiten bei der Benutzung in einer *Bedingung* oder einer *Folgerung* Rechnung zu tragen. Bei *Bedingungen* wird der Wert einer Variablen abgefragt bzw. mit einem anderen Wert verglichen, bei *Folgerungen* wird einer Variablen ein neuer Wert zugewiesen bzw. es wird die Zuweisung eines Wertes verboten.

y_1	- bei Verwendung in einer Bedingung	„*der Motor*“ ist „*aktiv*“
	- bei Verwendung in einer Folgerung	„*der Motor*“ wird „*gestartet*“
¬ y_1	- bei Verwendung in einer Bedingung	„*der Motor*“ ist „*nicht aktiv*“
	- bei Verwendung in einer Folgerung	„*der Motor*“ wird „*gestoppt*“

Durch die unterschiedliche Verwendung des Hilfsverbs ‚sein’ in den Bedingungen und Folgerungen (einerseits in einem passiven Zusammenhang, andererseits in einem aktiven Zusammenhang) ergibt sich die Notwendigkeit einer unterschiedlichen Definition der *Werte*.

Formulierung der Nomen-Wert-Kombinationen für Bool- bzw. Integer-Variablen

Während Variablen des Datentyps Bool nur zwei Zustände aufweisen und somit die Anzahl der formulierbaren *Werte* begrenzt ist (siehe oben), besteht diese Begrenzung im Allgemeinen nicht für Variablen des Datentyps Integer. Jedoch ist es auf der anderen Seite auch unsinnig, den gesamten Wertebereich einer Integer-Variablen vollständig ‚verbalisieren’ zu wollen. Betrachtet man jedoch die Art und Weise der Verwendung von Integer-Variablen in SPS-Programmen, so fallen einige wenige typische Anwendungsfälle auf: (ohne dass die folgende Liste vollständig ist)

- intern als Zustandsvariable zur Kodierung von Programmzuständen,
- externe Sensor-Werte,
- externe Aktor-Werte,
- ...

Die Anzahl der unterscheidbaren Zustände eines Programms ist begrenzt, so dass hier eine Erstellung der *Werte* für jeden der einnehmbaren Variablenzustände möglich ist.

z_1 = 1	- als Bedingung	„*das Programm*“ ist „*im Zustand 1*“
	- als Folgerung	„*das Programm*“ wird „*in den Zustand 1 überführt*“
z_1 = 2	- als Bedingung	„*das Programm*“ ist „*im Zustand 2*“
	- als Folgerung	„*das Programm*“ wird „*in den Zustand 2 überführt*“
...	...	...

In ereignisorientierten Steuerungsprogrammen, in denen Integer-Variablen zum Einlesen analoger Sensorwerte bzw. zum Ausgeben analoger Aktorwerte verwendet werden, erfolgt in vielen Fällen eine Intervallbildung bzw. Diskretisierung des verfügbaren Wertebereiches. Bei analogen Sensoren werden bestimmte Grenzwerte abgefragt, bei deren Über- oder Unterschreitung Aktionen auszulösen sind. Auf diese Weise lassen sich relevante Werte

dieser Variablen identifizieren, für die verbale Entsprechungen zu formulieren sind. In diesem Zusammenhang ist auch die Benutzung von Vergleichsoperatoren möglich.

x_t = 50 - „*die Temperatur*“ ist „*gleich 50°C*“
x_t > 50 - „*die Temperatur*“ ist „*größer als 50°C*“
x_t < 50 - „*die Temperatur*“ ist „*kleiner als 50°C*“

Sichtenorientierte Formulierung der Nomen-Wert-Kombinationen

Ein weiterer Punkt der kreativen Einflussnahme des SFS-Anwenders auf die Lesbarkeit und Verständlichkeit seiner Anforderungssätze ergibt sich durch einen unterschiedlichen Abstraktionsgrad bei der Erstellung der *Nomen* und *Werte*. Möglich sind hier:

- eine vorwiegend variablenorientierte Beschreibung,
- eine prozessorientierte Beschreibung.

Bei einer variablenorientierten Beschreibung erfolgt lediglich eine direkte, nichtinterpretierte Überführung der SPS-Variablen und ihrer Werte in die *Nomen* und *Werte*.

x_1 - „*die Variable x_1*“ ist „*TRUE*“ („*wahr*“, „*gesetzt*“)
¬ x_1 - „*die Variable x_1*“ ist „*FALSE*“ („*falsch*“, „*nicht gesetzt*“)

Mit dieser Darstellung kann man prinzipiell bereits umgehen, die entscheidenden Möglichkeiten der Sicherheitsfachsprache bestehen jedoch vor allem auch darin, die verwendeten SPS-Termini zu interpretieren und somit eine bessere Verständlichkeit und auch eine Erhöhung der Anwendungssicherheit zu erzielen. Innerhalb einer prozessorientierten Beschreibung kann der Anwender genau die technische Bedeutung der einzelnen Variablenzustände angeben und somit Missverständnisse und Fehlinterpretationen vermeiden.

x_2 - „*die Variable x_2*“ ist „*gesetzt*“ (variablenorientiert)
- „*der Not-Aus-Taster*“ ist „*nicht gedrückt*“ (prozessorientiert)

¬ x_2 - „*die Variable x_2*“ ist „*nicht gesetzt*“ (variablenorientiert)
- „*der Not-Aus-Taster*“ ist „*gedrückt*“ (prozessorientiert)

Der Grad der Interpretation kann natürlich, je nach Notwendigkeit, angepasst werden:

x_3 - „*die Variable x_3*“ ist „*gesetzt*“ (variablenorientiert)
- „*der Endlagenschalter x_3*“ ist „*gedrückt*“ (prozessorientiert)
- „*der Werkzeugschlitten*“ ist „*in der linken Endstellung*“

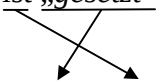
Die Abbildung 14 fasst die verschiedenen Möglichkeiten der Erstellung der Verbalphrasen noch einmal zusammen.

Einsatz der Variablen ...	Datentyp	**Sensoren**	**Aktoren**	**int. Variablen**
... in einer **Bedingung** „*Wenn* ...	**Bool**	**Abfrage eines Sensorzustandes** **x_1 = TRUE** „*der Hydraulikzylinder*“ ist „*oben*“ „*der Kran*“ ist „*in Entnameposition*“ „*die Lichtschranke*“ ist „*blockiert*“	**Abfrage einer laufenden Aktion** **y_1 = TRUE** „*das Öffnen der Presse*“ ist „*aktiv*“ „*der Magnet*“ ist „*aktiv*“	**Abfrage eines besteh. Zustandes** **z_1 = TRUE** „*der Zustand 1 der Presse*“ ist „*aktiv*“
	Integer	**Abfrage eines Sensorzustandes** **x_2 = 50** „*die Temperatur*“ ist „*50° Celsius*“ „*der Roboter*“ ist „*vor der Presse*“	**Abfrage einer laufenden Aktion** **y_2 = 50** „*die Drehzahl des Motors*“ ist „*50 UpM*“	**Abfrage eines besteh. Zustandes** **z_2 = 50** „*der Wert des Merkers*“ ist „*50*“
... in einer **Folgerung** , *dann* ...	**Bool**		**Start oder Ende einer Aktion** **y_1 = TRUE** „*die Presse*“ wird „*geschlossen*“ „*der Magnet*“ wird „*aktiviert*“	**Übergang in einen neuen Zustand** **z_1 = TRUE** „*der Zustand 1 der Presse*“ wird „*gesetzt*“
	Integer		**Veränderung einer Aktion** **y_2 = 50** „*die Drehzahl des Motors*“ wird „*auf 50 UpM gesetzt*“	**Übergang in einen neuen Zustand** **z_2 = 50** „*der Merker*“ wird „*auf 50 gesetzt*“

Abbildung 14 - Möglichkeiten zur Erstellung der Verbalphrasen

Eine letzte Anmerkung soll den grammatikalischen Möglichkeiten der Sicherheitsfachsprache gelten. Bei Anforderungen, deren Bedingungs- bzw. Folgerungsteil mehrere konjunktiv verknüpfte Teilbedingungen bzw. Teilfolgerungen enthält, kann (gemäß den Gegebenheiten der deutschen Umgangssprache) eine Umstellung der Satzglieder erfolgen.

Wenn „*die Variable 1*“ ist „*gesetzt*“ und „*die Variable 2*“ ist „*gesetzt*“, dann ...



Wenn „*die Variable 1*“ „*gesetzt*“ ist und „*die Variable 2*“ ist „*gesetzt*“, dann ...
Wenn „*die Variable 1*“ „*gesetzt*“ ist und „*die Variable 2*“ „*gesetzt*“ ist, dann ...

Im vorliegenden Fall wurde die Reihenfolge des Hilfsverbs ‚ist’ und des *Wertes* „*gesetzt*“ vertauscht, um die geforderte Wortreihenfolge einzuhalten. Die Sicherheitsfachsprache ist auf solche grammatikalischen Besonderheiten ausgelegt, der Anwender kann (mit Ausnahme einiger irregulärer Varianten) die ihm passend erscheinende Wortfolge auswählen. Bei der Überführung der Anforderungssätze in die temporallogischen Formeln werden diese Variationsmöglichkeiten ebenfalls berücksichtigt.

4. Temporale Logik als Basis der Sicherheitsfachsprache

Die zuvor beschriebenen Anforderungskategorien müssen formal dargestellt werden. Für die eingangs beschriebene Verifikation durch das Model-Checking-Verfahren wird eine Darstellung der Programmanforderungen in Form von Formeln der Temporalen Logik benötigt. Die folgenden Abschnitte geben einen Überblick über die logischen Grundlagen und leiten die notwendigen Darstellungsformen ab.

4.1 Aussagenlogik

Das Alphabet der Aussagenlogik besteht aus atomaren Formeln und Symbolen. Zu den Symbolen gehören:

$\neg$	≡	Negation – ‚nicht'
$\rightarrow$	≡	Konditional – ‚wenn ... dann'
(	≡	öffnende Klammer
)	≡	schließende Klammer

Formeln werden durch folgende Regeln gebildet:

1. Jede atomare Formel ist eine Formel.
2. Wenn *p* eine Formel ist, dann ist $\neg p$ auch eine Formel.
3. Wenn *p* und *q* Formeln sind, dann ist $p \rightarrow q$ auch eine Formel.

Mit diesen Grundlagen lassen sich andere Operatoren und Symbole ableiten:

$p \wedge q$	≡	$\neg (p \rightarrow \neg q)$
$p \vee q$	≡	$\neg p \rightarrow q$
$p \leftrightarrow q$	≡	$(p \rightarrow q) \wedge (p \rightarrow q)$
TRUE	≡	$p \vee \neg p$
FALSE	≡	$\neg$ TRUE

4.2 Temporale Logik

Um Zustandssequenzen eines Systems beschreiben zu können, benötigt man eine Logik, die es gestattet, die relativen Zeitbegriffe der Sicherheitsfachsprache („*sofort*“, „*nie*“, „*solange*“) zu verwenden. Die Temporale Logik (TL) ist eine um zeitliche Operatoren erweiterte Aussagenlogik. Sie dient als Werkzeug, um zeitliche Besonderheiten in der logischen Struktur eines Programms beschreiben zu können [Fisher1993, Kröger1987, Manna1992]. Die temporale Logik kann zur Programmverifikation verwendet werden. Man formalisiert hierbei den Begriff des Zeitpunktes und betrachtet nicht mehr feste Interpretationen, sondern vielmehr Interpretationen, die zu verschiedenen Zeitpunkten einen unterschiedlichen Wahrheitsgehalt haben können. So können bestimmte Aussagen in Abhängigkeit vom aktuellen Zeitpunkt wahr oder unwahr sein.

Zeitpunkt	Aktion	a	b	c	A = [(a + b = c ∧ a > 0) → b > 0]
1	...	3	-3	0	FALSE
2	c := b	3	-3	-3	TRUE
3	b := b - a	3	-6	-3	FALSE

Tabelle 7 - Abhängigkeit des Wahrheitsgehaltes einer Aussage vom Zeitpunkt der Betrachtung

In der obenstehenden Tabelle 7 [Kröger1987] ist die Aussage A „Wenn die Summe aus a und b gleich c ist und a ist größer als Null, dann ist b größer als Null“ für die Zeitpunkte 1 und 3 falsch, während sie im Zeitpunkt 2 richtig war.

Für die Darstellung solcher Besonderheiten existieren in der Temporalen Logik spezielle Operatoren (Tabelle 8), die sich auf zeitliche Abhängigkeiten beim Ablauf eines Programms beziehen.

Operator/ Symbol	Bezeichnung	Bedeutung
X / ❍	”at ne**X**t”-Operator	”im nächsten Zeitpunkt gilt”
F / ◊	”eventually / **F**inally”-Operator	”irgendwann gilt schließlich”
G / □	”henceforth / **G**enerally”-Operator	”von nun an gilt immer”
U / ∪	”**U**ntil”-Operator	”x gilt solange, bis y gilt”

Tabelle 8 - Beispiele für Temporaloperatoren

4.3 Computation Tree Logic

Innerhalb der Temporalen Logik sind zwei Sichtweisen der Zeit möglich:

1) Lineare Sichtweise: Hierbei gibt es für jeden Zeitpunkt nur eine mögliche Zukunft, Verzweigungen zu alternativen Varianten sind nicht möglich. (linear TL)
2) Verzweigte Sichtweise: Hierbei gibt es an bestimmten Zeitpunkten die Möglichkeit, dass sich die Zeit durch Nichtdeterminismus verzweigt. Es gibt mehrere Alternativen der Zukunft. (branching-time TL)

Die Wahl fiel auf CTL (computation tree logic), eine Temporale Logik, die sich an der zweiten Sichtweise orientiert.

Neben den bereits erwähnten TL-Operatoren gibt es für branching-time-Logiken die zusätzlichen Pfadquantoren A bzw. E. Damit ergeben sich die folgenden Grundoperationen

EX φ - in mindestens einem Pfad, der dem aktuellen Zustand folgt, wird φ im unmittelbar nachfolgendem Zustand gültig werden

AX φ - φ ist in allen unmittelbar nachfolgenden Zuständen gültig

EF φ - in mindestens einem Pfad, der dem aktuellen Zustand folgt, wird φ irgendwann gültig werden

AF φ - in allen Pfaden, die dem aktuellen Zustand folgen, wird φ irgendwann gültig werden

EG φ - in mindestens einem Pfad, der dem aktuellen Zustand folgt, wird φ immer gültig sein

AG φ - φ ist in allen nachfolgenden Zuständen gültig

E [φ_1 U φ_2] - in mindestens einem Pfad, der dem aktuellen Zustand folgt, gilt φ_1 solange, bis φ_2 gilt

A [φ_1 U φ_2] - in allen Pfaden, die dem aktuellen Zustand folgen, gilt φ_1 solange, bis φ_2 gilt,

Wir beschreiben nun formal die Menge der syntaktisch korrekten CTL-Formeln und geben ihre Bedeutung an. Unsere Darstellung orientiert sich an der bereits eingeführten Terminologie für Registernetze.

Sei *V* ein Registernetz (das etwa das Kontrollmodell eines SPS/Anlagen-Systems ist). Die Menge der CTL-Formeln enthält die folgenden Elemente:

1 $r = c$ und $r < c$ CTL-Formeln, wobei r ein Register von V ist und $c \in \mathbf{Nat}_\perp$ eine Konstante.

2 Sind φ_1 und φ_2 CTL-Formeln, so auch $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, EXφ_1, AXφ_1, EFφ_1, AFφ_1, EGφ_1, AGφ_1, E[φ_1 U φ_2] und A[φ_1 U φ_2].

3 Das sind alle CTL-Formeln.

Die Semantik von CTL-Formeln beruht auf dem Begriff der Zustandsfolge: Ist *V* ein Registernetz, und *z* ein Zustand von *V*, so bezeichnen wir eine endliche oder unendliche Folge von Zuständen $z = z_1z_2...z_n...$, so dass $z = z_1$ und $z_i\ [t\rangle\ z_{i+1}$ für eine Transition *t* gilt, als *Schaltfolge* in *V*. Dabei ist $z = z_1z_2...z_n$ endlich, gdw. $\neg z_n\ [t\rangle$ für alle Transitionen *t* von *V* gilt. In diesem Fall bezeichnen wir die *Länge* von *z* als $|z| = n$. Ist *z* unendlich, so schreiben wir $|z| = \infty$, wobei wir $n < \infty$ für alle natürlichen Zahlen *n* annehmen. Wir betrachten also nur maximale Schaltfolgen, d. h. solche Schaltfolgen, die entweder unendlich oder aber endlich und nicht weiter fortsetzbar sind.

Ist $z = z_1z_2...z_n...$ eine Schaltfolge und ist $i > 0$, so dass z entweder unendlich oder endlich und $|z| \geq i$ ist, so ist $z[i] = z_i$. Ist $z_1z_2...z_n$ eine endliche Schaltfolge, so heißt z_n von z_1 aus *erreichbar*. Ist *z* der Initialzustand von *V* und ist *z*' von *z* aus erreichbar, so heißt *z*' in *V erreichbar*.

Sei *z* ein in *V* erreichbarer Zustand. Wir bezeichnen die Menge aller Schaltfolgen *z* mit $z = z[i]$ mit $P(z)$.

Zur formalen Definition der Semantik von CTL definieren wir wie gewöhnlich eine Relation *V*, *z* **sat** φ für Registernetz *V*, einen in *V* erreichbaren Zustand *z* und einer CTL-Formel φ. *V*, *z* **sat** φ *z* sei hierbei ein in *V* erreichbarer Zustand. Die Definition erfolgt induktiv über dem Aufbau von CTL-Formeln:

1.	*V*, *z* **sat** $r = c$	gdw.	$z = (Q, \text{val})$ und $\text{val}(r) = c$ gilt.
2.	*V*, *z* **sat** $r < c$	gdw.	$z = (Q, \text{val})$ und $\perp \neq \text{val}(r) < c$ gilt.
3.	*V*, *z* **sat** $\neg\varphi$	gdw.	nicht *V*, *z* **sat** φ gilt.
4.	*V*, *z* **sat** $\varphi_1 \vee \varphi_2$	gdw.	*V*, *z* **sat** φ_1 oder *V*, *z* **sat** φ_2 wahr ist.
5.	*V*, *z* **sat** EXφ	gdw.	es ein $z \in P(z)$ mit $\|z\| > 1$ und *V*, $z[2]$ **sat** φ gibt.
6.	*V*, *z* **sat** AXφ	gdw.	für alle $z \in P(z)$ gilt: *V*, $\|z\| > 1$ impliziert $z[2]$ **sat** φ.
7.	*V*, *z* **sat** EFφ	gdw.	es ein $z \in P(z)$ und ein $i > 0$ mit $\|z\| \geq i$ und *V*, $z[i]$ **sat** φ gibt.
8.	*V*, *z* **sat** AFφ	gdw.	es für alle $z \in P(z)$ ein $i > 0$ mit $\|z\| \geq i$ und *V*, $z[i]$ **sat** φ gibt.

9.	V, z **sat** EGφ	gdw.	es ein $z \in P(z)$ gibt, so dass $V, z[i]$ **sat** φ für alle i mit $\|z\| \geq i > 0$ gilt.
10.	V, z **sat** AGφ	gdw.	für alle $z \in P(z)$ und für alle i mit $\|z\| \geq i > 0$ $V, z[i]$ **sat** φ gilt.
11.	V, z **sat** $E[\varphi_1 U \varphi_2]$	gdw.	es ein $z \in P(z)$ und ein i mit $\|z\| \geq i > 0$ gibt, so dass $V, z[i]$ **sat** φ_2 gilt, während für alle j mit $0 < j < i$ die Aussage $V, z[j]$ **sat** φ_j wahr ist.
12.	V, z **sat** $A[\varphi_1 U \varphi_2]$	gdw.	für alle $z \in P(z)$ ein i mit $\|z\| \geq i > 0$ existiert, so dass $V, z[i]$ **sat** φ_2 gilt, während für alle j mit $0 < j < i$ die Aussage $V, z[j]$ **sat** φ_j wahr ist.
13.	V **sat** φ	gdw.	V, z ***sat*** φ für den Initialzustand z von V gilt.

Offenbar sind einige CTL-Formeln äquivalent zu anderen. Unter äquivalenten Formeln verstehen wir solche Formeln φ_1 und φ_2, für die V, z **sat** φ_1 gdw. V, z **sat** φ_2 für alle Registernetze V und alle in V erreichbaren Zustände z gilt. Wir schreiben dann $\varphi_1 \equiv \varphi_2$.

1. Für CTL-Formeln gelten alle Äquivalenzen der Aussagenlogik.
2. $EX\varphi \equiv \neg AX\neg\varphi$, $AX\varphi \equiv \neg EX\neg\varphi$,
3. $EF\varphi \equiv \neg AG\neg\varphi$, $AF\varphi \equiv \neg EG\neg\varphi$, $EG\varphi \equiv \neg AF\neg\varphi$, $AG\varphi \equiv \neg EF\neg\varphi$,
4. $EF\varphi \equiv E[TRUE\ U\ \varphi]$, $AF\varphi \equiv A[TRUE\ U\ \varphi]$,
5. $E[\varphi_1\ U\ \varphi_2] \equiv \varphi_2 \vee (\varphi_1 \wedge EX\ E[\varphi_1\ U\ \varphi_2])$, $A[\varphi_1\ U\ \varphi_2] \equiv \varphi_2 \vee (\varphi_1 \wedge AX\ A[\varphi_1\ U\ \varphi_2])$

4.4 Darstellung der Anforderungskategorien in CTL-Formeln

Es folgt nun die ausführliche Darstellung der Umsetzung der einzelnen Anforderungskategorien der Sicherheitsfachsprache, die in Abschnitt 0 abgeleitet wurden.

Für die Umsetzung der Anforderungen in die CTL-Formeln gelten folgende generelle Bedingungen:

- Die aufgestellten Anforderungen gelten immer für den gesamten Erreichbarkeitsraum des Systems. Diesem Umstand muss durch die Benutzung der CTL-Operatoren AG (‚generell gilt in allen nachfolgenden Zuständen') Rechnung getragen werden. Wenn der Gültigkeitsbereich einer Anforderung nur für bestimmte Abschnitte (Sequenzen) gelten soll, so ist dieser Bereich durch die aufgestellten Bedingungen bzw. das Beobachtungsintervall zu beschränken.
- Alle Anforderungen, die sich auf die korrekte Ausführung des SPS-Programms beziehen, müssen mit den eingeführten Beobachtungsvariablen (siehe Abschnitt 2.1) verknüpft werden. Die meisten Analysen sind erst dann sinnvoll durchführbar, wenn die Bearbeitung des SPS-Programms abgeschlossen ist (rdy_plc).
- Die kürzeste Reaktionszeit einer realen SPS ist von ihrer Zykluszeit und der zeitlichen Abfolge ihrer Betriebssystemroutinen abhängig (vergleiche Abbildung 5). Die schnellste Reaktion einer SPS ist somit dann gegeben, wenn ein relevantes Ereignis im

Moment des Einlesens der Sensorsignale eintritt (rdy_in) und dieses Ereignis unmittelbar anschließend im Steuerungsprogramm verarbeitet wird (rdy_plc wird erstmalig gesetzt). Durch diese beiden Beobachtungsvariablen kann diese Tatsache überwacht werden.

Im Abschnitt 3.4 wurden die folgenden Kategorien des Modalparameters abgeleitet:

Einfache Forderung	**Einfaches Verbot**
innerhalb des *BI* **muss** es ein *F* geben außerhalb des *BI* darf es ein *F* geben	innerhalb des *BI* **darf** es **kein** *F* geben außerhalb des *BI* darf es ein *F* geben
Erweiterte Forderung	**Erweiterte Möglichkeit**
innerhalb des *BI* **muss** es ein *F* geben außerhalb des *BI* **darf** es **kein** *F* geben	innerhalb des *BI* darf es ein *F* geben außerhalb des *BI* **darf** es **kein** *F* geben

An dieser Stelle lassen sich auch wieder Verbindungen zwischen den Kategorien des Modalparameters ableiten:

Ableitung 1: Bei der einfachen Forderung muss es innerhalb des Beobachtungsintervalls ein *F* geben.

Ableitung 2: Das einfache Verbot ist eine Umkehrung der einfachen Forderung (innerhalb des Beobachtungsintervalls darf es kein *F* geben).

Ableitung 3: Bei der erweiterten Möglichkeit wird das Beobachtungsintervall im Vergleich zum einfachen Verbot invertiert (es darf außerhalb des Beobachtungsintervalls kein *F* geben).

Ableitung 4: Die erweiterte Forderung kann man als Verknüpfung zwischen der einfachen Forderung (innerhalb des *BI* muss es ein *F* geben) und der erweiterten Möglichkeit (außerhalb des *BI* darf es kein *F* geben) betrachten.

Diese Zusammenhänge spiegeln sich auch nachfolgend in den CTL-Formeln wieder. Die folgende Auflistung der Anforderungskategorien mit ihrer CTL-Darstellung ist durch den Zeitparameter gegliedert:

- Zuerst werden die Kategorien mit dem Zeitparameter ‚Zustand' dargestellt,
- danach wird die Umsetzung der beiden Kategorien mit dem Zeitparameter ‚Direkt' gezeigt,
- im Gegensatz zum Zeitparameter ‚Direkt' wird die Erfüllung der Folgerung bei Kategorien mit dem Zeitparameter ‚Unbegrenzt' nicht im nächsten SPS-Zyklus, sondern lediglich irgendwann danach gefordert,
- bei Kategorien mit dem Zeitparameter ‚Fremdbegrenzt' wird der Zeitraum für die Erfüllung der Folgerung jedoch wieder durch eine zweite, unabhängige Bedingungen eingegrenzt,
- bei Kategorien mit dem Zeitparameter ‚Selbstbegrenzt' ist das Ende des Beobachtungsintervalls nicht durch eine zweite Bedingung, sondern durch die Nichterfüllung der ursprünglichen Bedingung gegeben.

Zeitparameter ‚Zustand'

- **Kategorie DEs1 - einfache Forderung - Zustand**

Analyseinhalt: In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, muss gleichzeitig auch eine Folgerung *F* erfüllt sein.

AG ((rdy_plc & *B*) -> *F*)

- **Kategorie PRs1 - einfaches Verbot - Zustand**

Analyseinhalt: In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, darf eine Folgerung *F* nicht erfüllt sein.

AG ((rdy_plc & *B*) -> ¬*F*)

Diese Formel lässt sich umformen, sie erhält dadurch das typische Aussehen einer Sicherheitseigenschaft.

AG ¬(rdy_plc & *B* & *F*)

- **Kategorie POe1 - erweiterte Möglichkeit - Zustand**

Analyseinhalt: Hier wird verlangt, dass eine bestimmte Folgerung nur dann eintreten darf, wenn gleichzeitig eine bestimmte Bedingung erfüllt wird. In der Umkehrung bedeutet dies, dass in jedem Zustand, in dem zwar rdy_plc aber eine Bedingung *B* nicht erfüllt ist, die Folgerung *F* nicht erfüllt sein darf.

AG ((rdy_plc & *F*) -> *B*) bzw. AG ((rdy_plc & ¬*B*) -> ¬*F*)

Nach Umformung erhält man:

AG ¬(rdy_plc & ¬*B* & *F*)

- **Kategorie DEe1 - erweiterte Forderung - Zustand**

Analyseinhalt: In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, muss auch eine Folgerung *F* erfüllt sein. Zusätzlich darf in jedem Zustand, in dem zwar rdy_plc aber die Bedingung *B* nicht erfüllt ist, die Folgerung *F* nicht erfüllt sein.

AG (((rdy_plc & *B*) -> *F*) & ((rdy_plc & ¬*B*) -> ¬*F*))

Die gezeigte Formel lässt sich umformen und vereinfachen.

AG ((rdy_plc & *B*) <-> (rdy_plc & *F*))

AG (rdy_plc -> (*B* <-> *F*))

Zeitparameter ‚Direkt'

- **Kategorie DEs2 - einfache Forderung - Direkt**

Analyseinhalt: Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss im nächstmöglichen Zustand, in dem rdy_plc erfüllt ist, auch eine Folgerung *F* erfüllt sein. Es besteht die Forderung, dass das Steuerungsprogramm auf ein beim Einlesen der Sensorwerte erkanntes Ereignis unmittelbar reagiert.

Diese Forderung ist erfüllt wenn die Reaktion nach der unmittelbar folgenden Abarbeitung des SPS-Programms eingetreten ist, d. h. sobald die Beobachtungsvariable rdy_plc erstmals von FALSE auf TRUE gewechselt hat.

AG ((rdy_in & *B*) -> A[¬rdy_plc U (rdy_plc & *F*)])

Die CTL-Formel enthält den Until-Operator U, durch die Verbindung mit ¬rdy_plc und rdy_plc wird die steigende Signalflanke der Beobachtungsvariablen erkannt. Die Formel kann folgendermaßen interpretiert werden: Wenn nach dem Einlesen der Sensorwerte eine Bedingung *B* erfüllt ist, muss die darauf folgende Phase der Bearbeitung des SPS-Programms immer die zugehörige Reaktion *F* liefern.

- **Kategorie DEe2 - erweiterte Forderung - Direkt**

Analyseinhalt: Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss im nächsten Zustand, in dem rdy_plc erfüllt ist, auch eine Folgerung *F* erfüllt sein. In Erweiterung zur Kategorie DEs2 gilt innerhalb dieser Anforderungskategorie, dass unmittelbar nach denjenigen Zuständen, in denen zwar rdy_in, jedoch nicht die bezeichnete Bedingung *B* erfüllt ist, die genannte Folgerung auch nicht erfüllt sein darf.

AG (((rdy_in & *B*) -> A[¬rdy_plc U (rdy_plc & *F*)])
& ((rdy_in & ¬*B*) -> A[¬rdy_plc U (rdy_plc & ¬*F*)]))

Zeitparameter ‚Unbegrenzt'

- **Kategorie DEs5 - einfache Forderung - Unbegrenzt**

Analyseinhalt: Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss es irgendwann einen Zustand geben, in dem rdy_plc und *F* erfüllt sind. Es besteht die Forderung, dass das Steuerungsprogramm auf ein beim Einlesen der Sensorwerte erkanntes Ereignis immer irgendwann reagiert. Die auslösende Bedingung muss dann jedoch nicht mehr erfüllt sein. Das Beobachtungsintervall beginnt mit dem Zustand, in dem beim Einlesen der Sensorwerte erstmalig die Bedingung erfüllt ist.

AG ((rdy_in & *B*) -> AF (rdy_plc & *F*))

- **Kategorie PRs5 - einfaches Verbot - Unbegrenzt**

Analyseinhalt: Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, darf es danach keinen Zustand mehr geben, in dem rdy_plc und eine Folgerung *F* erfüllt werden. Diese Kategorie ist wiederum vergleichbar mit der Kategorie DEs5 - einfache Forderung - unbegrenzt, nur dass hierbei nach dem Erkennen des Ereignisses die beschriebene Reaktion niemals wieder eintreten darf.

AG ((rdy_in & *B*) -> ¬EF (rdy_plc & *F*))

bzw. AG ((rdy_in & *B*) -> AG ¬(rdy_plc & *F*))

- **Kategorie POe5 - erweiterte Möglichkeit - Unbegrenzt**

Analyseinhalt: Bei diese Kategorie wird gefordert, dass eine bestimmte Folgerung erst dann ausgeführt werden darf, wenn vorher eine bestimmte Bedingung erfüllt wurde. Das Beobachtungsintervall beginnt also mit der erstmaligen Erfüllung der Bedingung. Kehrt man

diese Anforderung wieder um, so bedeutet dies, dass die Folgerung solange nicht ausgeführt werden darf, bis die Bedingung nicht mindestens einmal erfüllt wurde. D. h. außerhalb des Beobachtungsintervalls darf es keinen Zustand geben, in dem (rdy_plc & *F*) wahr ist.

$$A[\ \neg(\text{rdy\_plc} \ \&\ F)\ U\ (\text{rdy\_in} \ \&\ B)\ ]$$

- **Kategorie DEe5 - erweiterte Forderung - Unbegrenzt**

Analyseinhalt: Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss es irgendwann einen Zustand geben, in dem rdy_plc und F erfüllt sind. Zusätzlich darf es keinen Zustand mit rdy_plc und *F* geben, bevor nicht rdy_in und eine Bedingung *B* erfüllt worden sind. Dieses Verhalten lässt sich aus den beiden Kategorien DEs5 und POe5 zusammensetzen. Einerseits besteht die Forderung, dass das Steuerungsprogramm auf ein beim Einlesen der Sensorwerte erkanntes Ereignis immer irgendwann reagiert, andererseits darf die genannte Reaktion auch nicht vorher eintreten.

$$AG\ (\ (\ (\text{rdy\_in} \ \&\ B)\ \text{->}\ AF\ (\text{rdy\_plc} \ \&\ F\ )\ )$$
$$\&\ A[\ \neg(\text{rdy\_plc} \ \&\ F)\ U\ (\text{rdy\_in} \ \&\ B)\ ]\ )$$

Zeitparameter ‚Fremdbegrenzt'

- **Kategorie DEs4 - einfache Forderung - Fremdbegrenzt**

Analyseinhalt: Sobald es einen Zustand gibt, in dem rdy_in und eine Startbedingung B_1 erfüllt werden, muss es danach immer einen Zustand geben, in dem rdy_plc und eine Folgerung F erfüllt werden, bevor es einen Zustand gibt, in dem rdy_in und eine Endbedingung B_2 erfüllt sind.

$$AG(\text{rdy\_in} \ \&\ B_1\ \text{->}\ AF(\text{rdy\_plc} \ \&\ F\ \&\ AF(\text{rdy\_in} \ \&\ B_2)))$$

- **Kategorie PRs4 - einfaches Verbot - Fremdbegrenzt**

Analyseinhalt: Bei dieser Anforderungskategorie darf es im Beobachtungsintervall keinen Zustand geben, in dem die Folgerung erfüllt ist. Das Beobachtungsintervall beginnt mit einem Zustand, in dem die erste Bedingung B_1 erfüllt wird und endet mit einem Zustand, in dem die zweite Bedingung B_2 erfüllt wird. Dieses Verhalten lässt sich durch Erweiterung der Kategorie PRs5 erzielen, indem zusätzlich eine Endbedingung vereinbart wird.

$$A[\ (\ (\text{rdy\_in} \ \&\ B_1)\ \text{->}\ AG\ \neg(\text{rdy\_plc} \ \&\ F\ )\ )\ U\ (\text{rdy\_in} \ \&\ B_2)\ ]$$

- **Kategorie POe4 - erweiterte Möglichkeit - Fremdbegrenzt**

Analyseinhalt: Diese Kategorie lässt sich durch Verknüpfung der Kategorien POe5 und PRs5 erstellen. Eine bestimmte Folgerung darf nur zwischen zwei verschiedenen Bedingungen ausgeführt werden. Das Beobachtungsintervall wird durch die beiden Bedingungen B_1 und B_2 begrenzt. In der Umkehrung ergibt sich wiederum, dass die Folgerung nicht außerhalb des Beobachtungsintervalls erfüllt werden darf, d. h. weder *vor der ersten Bedingung* B_1 noch *nach der zweiten Bedingung* B_2.

$$AG\ (\ (\ (\text{rdy\_in} \ \&\ B_2)\ \text{->}\ \neg EF\ (\text{rdy\_plc} \ \&\ F\ )\ )$$
$$\&\ A[\ \neg(\text{rdy\_plc} \ \&\ F)\ U\ (\text{rdy\_in} \ \&\ B_1)\ ]\ )$$

- **Kategorie DEe4 - erweiterte Forderung - Fremdbegrenzt**

Analyseinhalt: Sobald es einen Zustand gibt, in dem rdy_in und eine Startbedingung B_1 erfüllt werden, muss es danach einen Zustand geben, in dem rdy_plc und eine Folgerung F erfüllt werden, bevor es einen Zustand gibt, in dem rdy_in und eine Endbedingung B_2 erfüllt sind. Zusätzlich darf es keinen Zustand mit rdy_plc und *F* geben, bevor rdy_in und B_1 erstmals gültig waren und es darf auch keinen Zustand mit rdy_plc und *F* geben, nachdem rdy_in und B_2 erstmals gültig waren.

$$\mathsf{AG}((\text{rdy\_in} \,\&\, B_1 \text{ -> } \mathsf{AF}(\text{rdy\_plc} \,\&\, F \,\&\, \mathsf{AF}(\text{rdy\_in} \,\&\, B_2)))$$
$$\&\; (\mathsf{A}[C\ \mathsf{U}\ B_1] \vee \mathsf{AG}\ C) \,\&\, \mathsf{AG}(B_2 \text{ -> } (\mathsf{A}[C\ \mathsf{U}\ B_1] \vee \mathsf{AG}\ C)))$$

mit C := rdy_plc & $\neg B_1$ & $\neg F$

Zeitparameter ‚Selbstbegrenzt'

- **Kategorie DEs3 - einfache Forderung - Selbstbegrenzt**

Analyseinhalt: Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, muss es danach einen Zustand geben, in dem sowohl rdy_plc, die Bedingung *B* und eine Folgerung *F* erfüllt werden, bevor die auslösende Bedingung *B* wieder ungültig wird. Die Forderung *F* findet also irgendwann innerhalb des Beobachtungsintervalls B_0 = (rdy_in $\wedge$ *B*) und B_1 = (rdy_in $\wedge$ $\neg B$) statt. Forderungen dieser Form können in CTL nicht formuliert werden: *F* muss irgendwann für eine gewisse Zeitspanne nach dem Eintreten von B_0 gelten; ist diese Zeitspanne jedoch abgelaufen und gilt wiederum $\neg F$, gilt noch immer B_0, ohne dass gefordert wäre, dass *F* wiederum irgendwann Gültigkeit erlangt. Diese beiden Fälle sind in CTL jedoch nicht unterscheidbar.

Wir lösen dieses Problem durch eine Modifikation des Modells, d. h. des Registernetzes *V*, welches das System Anlage/Steuerung repräsentiert. Die Idee hierbei ist, das Netz dahingehend zu modifizieren, dass eine selbstbegrenzte Forderung als äquivalente fremdbegrenzte Forderung aufgefasst werden kann. Sei *F* eine selbstbegrenzte Forderung mit einem Beobachtungsintervall, das durch B_0 und B_1 als Anfangs- und Endpunkt gegeben ist. Wir fügen vier Register *r*, r_{int}, r_0 und r_1 zu dem Registernetz *V* hinzu: Die Werte, die diese Register annehmen können, sind 0 und 1. Dabei gilt val(*r*) = 1 in einem Zustand *z* = (*Q*, val) gdw. *V*, *z* **sat** *B* erfüllt ist. Das Register r_{int} erhält den Wert 1 bei denjenigen Zuständen, die sich innerhalb des Beobachtungsintervalls befinden. *r* und r_{int} werden zur korrekten Bestimmung der Werte der Register r_0 und r_1 verwendet: Es gilt val(r_i) = 1 für *i* = 0, 1 gdw. *V*, *z* **sat** rdy_in = 1 gilt und darüber hinaus

1. *V*, *z* **sat** B_i erfüllt ist und
2. jeder Vorgängerzustand *z*' von *z* (d. h. jeder Zustand *z*' mit *z*' $[t\rangle$ *z* für eine Transition *t*) *V*, *z*' **sat** $\neg B_i$ erfüllt,

val(r_0) = 1 gilt also genau dann, wenn das Beobachtungsintervall beginnt, während val(r_1) = 1 das Ende des Beobachtungsintervalls markiert. Selbstbegrenzte Forderungen können dann durch die CTL-Formel

$$\mathsf{AG}(r_0 = 1 \rightarrow \mathsf{A}[r_1 = 0\ \mathsf{U}\ (F \wedge \mathsf{AF}\ r_1 = 1)]$$

mit r_0 := *V*, *z* **sat** B_0 , r_1 := *V*, *z* **sat** B_1

ausgedrückt werden.

In welcher Weise muss die Modifikation des Netzes nun erfolgen?

1. Notieren wir mit reg(*B*) die Menge derjenigen Register, die Variablen zugeordnet sind, die in *B* auftreten. Jede Transition *t*, die ein Register $v \in$ reg(*B*) als Ausgaberegister hat, erhält *r* als neues Ausgaberegister; die Beschriftung der Ausgabekante $t \rightarrow r$ ist

   if *B* **then** 1 **else** 0 **fi**.

   Die Initialbelegung von *r* ist natürlich 1, wenn *B* initial gilt, sonst 0.

2. Initial gilt $val(r_0) = val(r_1) = val(r_{int}) = 0$. Das ist korrekt, da zwar *B* initial bereits erfüllt sein kann, jedoch das der Variablen rdy_in zugeordnete Register r_{rdy_in} den Wert 0 hat: Das Beobachtungsintervall beginnt und endet in einem Zustand, in dem r_{rdy_in} mit dem Wert 1 belegt ist.

3. Sei t_0 diejenige Transition, die den Wert 1 in das Register r_{rdy_in} schreibt, während t_1 diejenige Transition ist, die den Wert von r_{rdy_in} auf 0 zurücksetzt. r_0, r_1 und r_{int} werden Ein- bzw Ausgaberegister von t_0, r_0 und r_1 werden Ausgaberegister von t_1. Dabei wird den entsprechenden Schreibkanten der folgende Code zugewiesen:

   $t_0 \rightarrow r_{int}$: **if** $r = 1 \wedge r_{int} = 0$ **then** 1 **elsif** $r = 0 \wedge r_{int} = 1$ **then** 0 **else** r_{int} **fi**;
   $t_0 \rightarrow r_0$: **if** $r = 1 \wedge r_{int} = 0$ **then** 1 **else** 0 **fi**;
   $t_1 \rightarrow r_1$: **if** $r = 0 \wedge r_{int} = 1$ **then** 1 **else** 0 **fi**;

   $t_1 \rightarrow r_0$: 0; und $t_1 \rightarrow r_1$: 0;

- **Kategorie PRs3 - einfaches Verbot - Selbstbegrenzt**

Analyseinhalt: Es wird gefordert, dass eine bestimmte Folgerung nicht ausgeführt werden darf, solange gleichzeitig eine bestimmte Bedingung erfüllt ist. Das Beobachtungsintervall umfasst also alle Zustände, in denen die Bedingung erfüllt ist. Dieser Umstand entspricht jedoch auch der Kategorie PRs1 - einfaches Verbot - Zustand. Die benutze Formel ist deshalb dieselbe.

AG ((rdy_plc & *B*) -> ¬*F*)

bzw. wiederum:

AG ¬(rdy_plc & *B* & *F*)

- **Kategorie POe3 - erweiterte Möglichkeit - Selbstbegrenzt**

Analyseinhalt: Innerhalb dieser Anforderungskategorie wird gefordert, dass eine bestimmte Folgerung *F* nur dann ausgeführt werden darf, solange gleichzeitig auch eine festgelegte Bedingung erfüllt ist. Das Beobachtungsintervall wird also durch alle Zustände gebildet, in denen die Bedingung erfüllt ist. Bei Invertierung des Beobachtungsintervalls bedeutet dies, dass die bezeichnete Folgerung nicht ausgeführt werden darf, wenn die Bedingung nicht erfüllt ist. Dieses Verhalten kann auf die Kategorie POe1 - erweiterte Möglichkeit - Zustand zurückgeführt werden.

AG ((rdy_plc & *F*) -> *B*) bzw. AG ((rdy_plc & ¬*B*) -> ¬*F*)

bzw. nach Umformung:

AG ¬(rdy_plc & ¬*B* & *F*)

- **Kategorie DEe3 - erweiterte Forderung - Selbstbegrenzt**

Analyseinhalt: Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, muss es danach einen Zustand geben, in dem sowohl rdy_plc, die Bedingung *B* und eine Folgerung *F* erfüllt werden, bevor die auslösende Bedingung *B* wieder ungültig wird. Zusätzlich darf in allen Zuständen, in denen zwar rdy_plc jedoch nicht die Bedingung *B* erfüllt wird, die Folgerung *F* auch nicht erfüllt sein. Diese Kategorie lässt sich wiederum durch Verknüpfung zweier anderer Kategorien, nämlich DEs3 und POe3, erstellen.

$$\mathsf{AG}((r_0 = 1 \rightarrow \mathsf{A}[r_1 = 0 \,\mathsf{U}\, (F \wedge \mathsf{AF}\, r_1 = 1)] \\ \& (\mathsf{A}[C \,\mathsf{U}\, r_0 = 1] \vee \mathsf{AG}\, C) \,\&\, \mathsf{AG}(r_1 = 1 \text{ -> } (\mathsf{A}[C \,\mathsf{U}\, r_0 = 1]) \vee \mathsf{AG}\, C)))$$

mit C := rdy_plc & $\neg B$ & $\neg F$

Im Anhang A.6 sind die einzelnen Kategorien der Sicherheitsfachsprache mit ihren CTL-Formeln noch einmal zusammengefasst dargestellt.

Ergebnis der Übersetzung der mit der Sicherheitsfachsprache formulierten Anforderungen ist eine Menge temporallogischer Formeln. Die atomaren Präpositionen dieser Formeln korrespondieren mit Variablen des Systemmodells, also letztendlich auch wiederum mit den Einträgen im Variablendeklarationsteil des SPS-Programms.

5. Benutzung der Sicherheitsfachsprache

In den vorherigen Abschnitten wurde deutlich, welche typischen steuerungstechnischen Problemstellungen es gibt und wie diese mit der Sicherheitsfachsprache dargestellt werden können. In diesem Abschnitt wird dargelegt, wie diese Anforderungen niedergeschrieben und in temporallogische Formeln umgewandelt werden können.

Die im Zusammenhang mit der SFS erstellten Werkzeuge unterteilen sich in zwei Bereiche:

- ein syntaxgesteuerter Editor zur Erstellung SFS-konformer Anforderungssätze,
- ein Compiler, der aus den Anforderungen CTL-Formeln generiert.

5.1 Erstellung von Programmanforderungen mit dem SFS-Editor

Mit Hilfe des syntaxgesteuerten Editors für die Sicherheitsfachsprache ist es auf einfache Art und Weise möglich, SFS-konforme Anforderungssätze zu erstellen. Natürlich können diese Anforderungssätze prinzipiell vom Anwender auch per hand mit einem Texteditor erzeugt und anschließend kompiliert werden, jedoch ist hier der Aufwand und die Fehleranfälligkeit weitaus größer.

Die Erstellung der Anforderungssätze erfolgt in mehreren Stufen: in einer Vorbereitungsphase müssen zunächst die applikationsspezifischen *Verbalphrasen* erstellt werden (siehe Abschnitt 3.7). In der zweiten Phase werden diese Verbalphrasen zum Ausfüllen der Anforderungskonstrukte verwendet.

In der Abbildung 15 ist die Erstellung der *Verbalphrasen* dargestellt. Die im Projekt verwendeten SPS-Variablen können dem Variablendeklarationsteil des zu analysierenden SPS-Programms entnommen werden. Anschließend ist für jede Variable und für jeden Wert, den diese annehmen kann, eine verbale Interpretation zu erstellen.

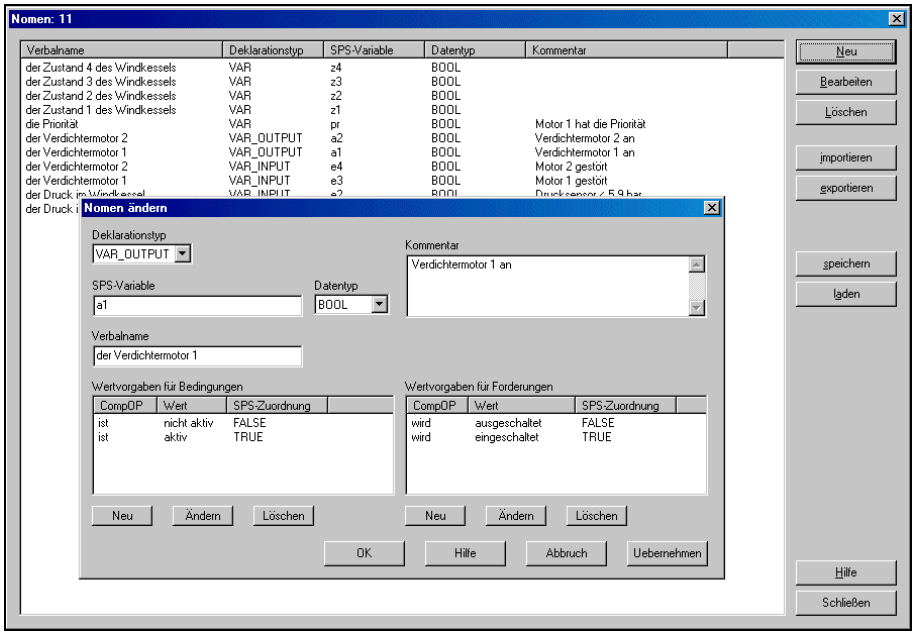


Abbildung 15 - Erstellung der Verbalphrasen

Die Auflistung der Variablen und ihre verbale Interpretation wird in einer Textdatei (Nomendeklarationsdatei) abgelegt und im Kompilierschritt wiederverwendet. Sollten zunächst nicht alle Variablen bekannt sein, kann das Erstellen der *Nomen* und *Werte* auch zu einem späteren Zeitpunkt erfolgen.

Nachdem alle SPS-Variablen interpretiert worden sind, können die eigentlichen Anforderungssätze erstellt werden. Dazu wird im SFS-Editor ein neuer Satz erzeugt und es werden der Modalparameter und der Zeitparameter bestimmt (Abbildung 16).

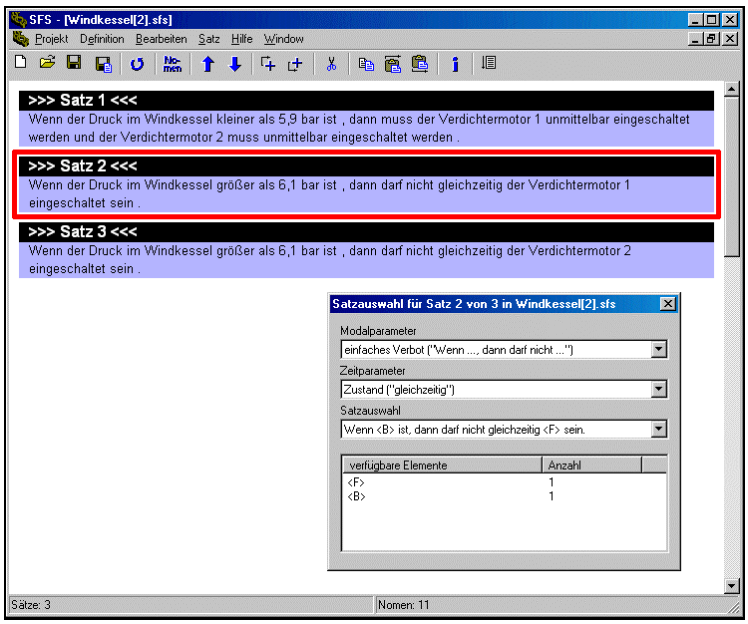


Abbildung 16 - Erstellung der Anforderungssätze

Durch die Bestimmung des Modal- und Zeitparameters ist der Anforderungsinhalt prinzipiell bestimmt, durch eine weitere Auswahl kann die Struktur des Satzes bestimmt werden. An dieser Stelle werden dem Anwender mehrere inhaltlich redundante Satzmuster angeboten, die er nach seinem Geschmack auswählen kann. Diese inhaltliche Redundanz bezieht sich vor allem auf die Stellung der Bedingung und der Folgerung im Satz und auf die Auswahl verschiedener gleichwertiger Schlüsselwörter.

Anschließend ist der Anforderungssatz durch Erstellung der Bedingung/-en und der Folgerung/-en zu vervollständigen (Abbildung 17). Auch hier werden wieder gleichwertige Formulierungen angeboten, die sich jedoch nur um die Stellung der einzelnen Wörter innerhalb der Formulierung unterscheiden.

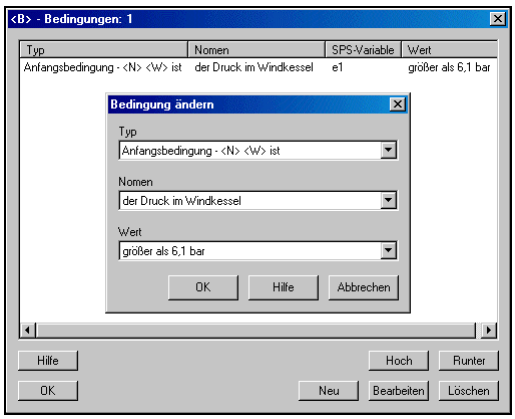


Abbildung 17 - Erstellung der Bedingungen

An dieser Stelle können immer mehrerer Bedingungen und Folgerungen vereinbart werden, diese sind in der Anforderung dann jeweils konjunktiv miteinander verknüpft.

In einem letzten Schritt werden die erstellten Sätze in eine Textdatei ausgegeben und stehen nun für die Kompilierung in eine CTL-Formel zur Verfügung.

5.2 Überführung der Programmanforderungen in CTL-Formeln

Dieser eigentliche Umwandlungsvorgang besteht aus zwei Stufen (Abbildung 18):

1. Im ursprünglichen Text ersetzt ein "PreLexer" die applikationsspezifischen Wortphrasen durch die in der Nomendeklarationsdatei hinterlegten SPS-Variablen und deren Werte. Der dadurch entstandene Zwischentext enthält dann nur noch sprachinterne, applikationsunabhängige Schlüsselwörter sowie formale Ausdrücke des SPS-Programms.

2. Im eigentlichen Umwandlungsvorgang wird der Zwischentext hinsichtlich der Bedeutung analysiert und in CTL-Formeln umgewandelt.

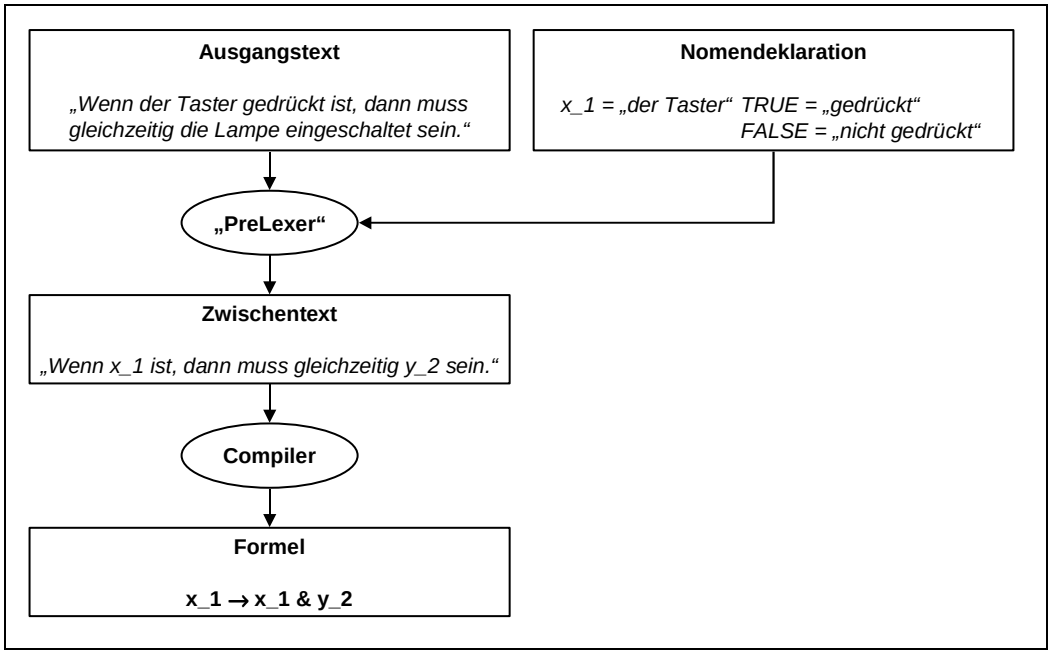


Abbildung 18 - Umsetzung der natürlichsprachlichen Anforderungen

5.3 Der „PreLexer“

Aufgabe des PreLexers ist es, die rein verbalen Anforderungen für den nachfolgenden Kompiliervorgang zu vereinfachen. Hierbei werden die logisch zusammengehörigen *Nomen* und *Werte* wieder zu einem *Nomen-Wert-Paar* zusammengefasst (z. B. *'die Presse'* und *'gestoppt'* zu **¬x_2**). Im Gegensatz zum eigentlichen Compiler "kennt" der PreLexer weder die Bedeutung, noch die Reichweite einer Anforderung. Er orientiert sich lediglich am Satzbau bzw. der Positionierung der in der Nomendeklarationsdatei abgelegten Wortphrasen sowie SFS-typischer Schlüsselwörter.

Die interne Umsetzung erfolgt dabei in zwei Schritten:

1. Umstellen einzelner Wörter im Satz, so dass danach das Nomen und der Wert nebeneinander und im richtigen Verhältnis zu den anderen Schlüsselwörtern stehen.
2. Zusammenfassen des Nomens und des Wertes und Ersetzen durch die SPS-Variable mit dem entsprechenden Wert.

Aus dem Text

/ Satz 1 */*
Wenn „der Zustand 1 der Presse“ „aktiv“ ist und „die Presse“ ist „unten“ , dann muss „das Schliessen der Presse“ unmittelbar „gestoppt“ werden und „das Öffnen der Presse“ muss unmittelbar „gestartet“ werden und „der Zustand 1 der Presse“ muss unmittelbar „zurückgesetzt“ werden und „der Zustand 2 der Presse“ muss unmittelbar „gesetzt“ werden .

/ Satz 2 */*
Wenn „der Zustand 2 der Presse“ „aktiv“ ist und „die Presse“ ist „oben“ , dann muss „das Öffnen der Presse“ unmittelbar „gestoppt“ werden und „der Zustand 2 der Presse“ muss unmittelbar „zurückgesetzt“ werden und „der Zustand 3 der Presse“ muss unmittelbar „gesetzt“ werden .

/ Satz 3 */*
Wenn „die Presse“ „unten“ ist , dann darf „das Ausfahren des Roboterarmes 1“ nicht gleichzeitig „gestartet“ sein .

wird durch den PreLexer der Text

/ Satz 1 */*
WENN press_z1 IST UND press_bottom IST, DANN MUSS UNMITTELBAR !press_down WERDEN UND MUSS UNMITTELBAR press_upward WERDEN UND MUSS UNMITTELBAR !press_z1 WERDEN UND MUSS UNMITTELBAR press_z2 WERDEN .

/ Satz 2 */*
WENN press_z2 IST UND press_top IST, DANN MUSS UNMITTELBAR !press_upward WERDEN UND MUSS UNMITTELBAR !press_z2 WERDEN UND MUSS UNMITTELBAR press_z3 WERDEN .

/ Satz 3 */*
WENN press_bottom IST, DANN DARF NICHT GLEICHZEITIG arm1_forward SEIN .

erzeugt. Dieser Text enthält nur noch SFS-typische Schlüsselwörter sowie SPS-Variablen.

5.4 Der Compiler

Der Compiler generiert automatisch aus dem Zwischentext des PreLexer’s die CTL-Formeln:

```
/* Satz 1 */
AG ( (rdy_in & ( press_z1 & press_bottom ) ) -> A[ !rdy_plc U (rdy_plc
     & ( !press_down & press_upward & !press_z1 & press_z2 ) ) ] )

/* Satz 2 */
AG ( (rdy_in & ( press_z2 & press_top ) ) -> A[ !rdy_plc U (rdy_plc &
     ( !press_upward & !press_z2 & press_z3 ) ) ] )

/* Satz 3 */
AG !( rdy_plc & press_bottom & arm1_forward)
```

Diese CTL-Formeln können anschließend durch einen Model-Checker am Kontrollmodell verifiziert werden.

6. Zusammenfassung und Ausblick

Mit der vorgestellten Sicherheitsfachsprache ist es möglich, die gewünschten Funktions- und Sicherheitseigenschaften eines Steuerungsprogramms verbal zu formulieren. Durch die Nutzung der natürlichen Sprache ist jetzt eine verständliche und auch für Nichtspezialisten lesbare Darstellungsform gegeben. Die SFS geht damit über die Möglichkeiten bisheriger Beschreibungsmittel hinaus, da mit ihr einerseits eine Verständigung zwischen verschiedenen Projektpartnern möglich wird und andererseits Anforderungen darstellbar werden, die mit den bisher verfügbaren Beschreibungsmitteln nicht formulierbar waren.

Da die Formulierung von Programmanforderungen mit Hilfe der natürlichen Sprache aufgrund ihrer Mehrdeutigkeit zunächst sehr problematisch ist, wurde die Sicherheitsfachsprache gemäß den typischen steuerungstechnischen Zusammenhängen, Anforderungen und Formulierstilen definiert. Sie bietet durch die formale Vorgabe einer Syntax die Möglichkeit der Hinterlegung einer formalen Semantik, durch die Vorgabe fester Satzmuster und -strukturen werden klare und leicht lesbare Darstellungen erzeugt. Ziel der SFS ist es weiterhin, den Anwender durch eindeutige und leicht verständliche Formulierungen zu einer klaren Problemdefinition zu zwingen.

Als Grundmuster der Sätze dient das Konditional („wenn ..., dann ...“), weil dem Anwender das Denken in solchen Kausalketten (auf eine erfüllte Bedingung folgt eine Reaktion) entgegenkommt. Die Anforderungskategorien werden durch einen Modal- und einen Zeitparameter definiert. Der Modalparameter bestimmt dabei den Inhalt einer Anforderung, der Zeitparameter bestimmt deren zeitliche Reichweite. Insgesamt sind in der Sicherheitsfachsprache 18 Anforderungskategorien definiert, wobei innerhalb jeder Kategorie weitere sprachlich unterschiedliche, jedoch inhaltlich redundante Satzmuster vorhanden sind. Die Vielfalt und Redundanz wurde jedoch bewusst gewählt, um dem Anwender verschiedene Möglichkeiten der Formulierung seiner Aufgabenstellung zu geben.

Für die Überprüfung der Handhabbarkeit wurden mehrere Fallstudien angefertigt, die sowohl die Umsetzung der SPS-Variablen in die elementaren Verbalphrasen als auch die Erstellung kompletter Anforderungssätze beinhalten.

Durch einen Compiler werden die Anforderungen in Formeln der Temporalen Logik (hier CTL) umgewandelt. Die Anwendung der Sicherheitsfachsprache gibt durch ihre anwenderfreundliche Schnittstelle jedem Nutzer die Möglichkeit, mit den Darstellungs- und Beschreibungsmöglichkeiten der Temporalen Logik umzugehen. Die erzeugten CTL-Formeln können für eine anschließende Verifikation des Steuerungsprogramms durch einen Model-Checker verwendet werden. Dieser liefert als Ergebnis der Überprüfung entweder eine Bestätigung für die Einhaltung der Spezifikation in allen modellierten Zuständen des Systems oder mögliche Gegenbeispiele, in denen eine oder mehrere Anforderungen nicht erfüllt werden.

Die Aussagekraft des Verifikationsergebnisses hängt jedoch immer von der Qualität der gegebenen Eingangsinformationen (Programmmodel, Umgebungsmodell, Anforderungen) ab. Als Ergebnis einer erfolgreichen Verifikation kann man somit von einem korrekten SPS-Programm bezüglich der spezifizierten Anforderungen unter den gegebenen Modellannahmen sprechen. Die in dieser Arbeit vorgestellten SFS verfolgt in diesem Zusammenhang speziell das Ziel, die Qualität der Anforderungen deutlich zu verbessern.

7. Literatur

[Alur1996] Alur, R.; Henzinger, T. A.; Ho, P.-H.: Automatic Symbolic Verification of Embedded Systems. IEEE Transactions on Software Engineering 22:181-201, 1996.

[Best1995] Best, E.; Grahlmann, B.: PEP - Programming Environment Based on Petri Nets, Documentation and User Guide; Univ. Hildesheim, Institut für Informatik, 1995

[Deussen2001] Deussen, P.: Partial Order Verification of Programmable Logic Controllers, Proc. Int. Conf. Application and Theory of Petri Nets, 20 S., 2001, im Druck.

[Emerson1990] Emerson, E. A.: Temporal and Modal Logic; in: J. v. Leeuwen ed.: Handbook of theoretical computer science, vol. B, Elsivier Science Publishers B. V. Amsterdam, pp. 995 - 1072

[Fisher1993] Fisher, M.; Owens, R.: Executable Modal and temporal logics, IJCAI‘93 Workshop, Chambery, France, Springer-Verlag, 1993

[HMM2000] Heiner, M., Mertke, T., Menzel, T.: "Safety-Knight - Methoden und Werkzeuge zur Verifikation von SPS-Anwenderprogrammen“, SPS/IPC/DRIVES, Nürnberg, November 2000, S. 395 - 403

[IEC 1131-3] DIN IEC 61131-3 (IEC 1131-3), „Speicherprogrammierbare Steuerungen, Teil 3: Programmiersprachen“, 1993

[Jensen1997] K. Jensen, "Coloured Petri Nets. Basic Concepts, Analysis, Methods and Practical Use", vol. 1, Basic Concepts, Monographs in Theoretical Computer Science. Springer Verlag, 1997.

[Kröger1987] Kröger, F.: Temporal Logics of Programs; Springer-Verlag, Berlin, 1987

[Manna1992] Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - Specification, Springer-Verlag, New York, 1992.

[Manna1995] Manna, Z., Pnueli, A.: Temporal verification of reactive systems – Safety, Springer-Verlag, New York, 1995.

[McMillan1992] McMillan, K. L.: The SMV System, Techn. Report, Carnegie-Mellon Univ. 1992.

[Mertke2000] Mertke, T., Menzel, T.: "Methods and tools to the verification of safety-related control software", IEEE International Conference of Systems, Man and Cybernetics, SMC 2000, pp.2455 - 2457, Nashville, TN (USA) October 2000

[Peterson1981] Peterson, J. L.: Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Reisig1991] Reisig, W.: Petri-Netze: Eine Einführung, Springer-Verlag, Berlin Heidelberg, 1991

[Starke1990] Starke, P. H.: Analyse von Petri-Netz-Modellen, B.G. Teubner Stuttgart, 1990

[Starke1992] Starke, P. H.: INA - Integrated Net Analyser Manual, Berlin, 1992

[VDI/VDE 3694] VDI/VDE 3694: Lasten/Pflichtenheft für den Einsatz von Automatisierungssystemen, 1991

[VHHP1995] Varpaaniemi, K., Halme, J., Hiekkanen, K., Pyssyslao, T.: PROD Reference Manual, Helsinki Univ. of Technology, Digital Systems Laboratory, Series B: Techn. Report no. 13, Espoo, 1995.

A Anhang

A.1 Formale Definition der Sicherheitsfachsprache

0. Ebene - Definition der Satzliste

```
<satzliste> ::=   <statement>
                | <satzliste> <statement>
```

1. Ebene - Definition des Modalparameters

```
<statement> ::=   <comment>
                | <DEs>              (* einfache Forderung *)
                | <DEe>              (* erweiterte Forderung *)
                | <POe>              (* erweiterte Möglichkeit *)
                | <PRs>              (* einfaches Verbot *)
```

2. Ebene - Definition des Zeitparameters

```
          <DEs> ::=   <DEs1>
                    | <DEs2>
                    | <DEs3>
                    | <DEs4>
                    | <DEs5>

          <DEe> ::=   <DEe1>
                    | <DEe2>
                    | <DEe3>
                    | <DEe4>
                    | <DEe5>

          <POe> ::=   <POe1>
                    | <POe3>
                    | <POe4>
                    | <POe5>

          <PRs> ::=   <PRs1>
                    | <PRs3>
                    | <PRs4>
                    | <PRs5>
```

Ebene 3 - Aufteilung in Satzvarianten, ab hier gleiche CTL-Formeln

```
<DEs1>  ::=   ’Wenn’ <ist_Bedingung_pl> ’, dann’ <DE1_f_pl> ’.’
            | <DEs1_b_pl> ’, wenn’ <ist_Bedingung_pl> ,.’

<DEs2>  ::=   'Wenn' <ist_Bedingung_pl> ', dann' <DE2_f_pl> '.'
            | <DEs2_b_pl> ', wenn' <ist_Bedingung_pl> '.'

<DEs3>  ::=   'Solange' <ist_Bedingung_pl> ',' <DE3_f_pl> '.'
            | <DEs3_b_pl> ', solange' <ist_Bedingung_pl> '.'

<DEs4>  ::=   'Wenn irgendwann' <war_Bedingung_pl> ', dann' <DE3_f_pl> ',
                bevor' <ist_Bedingung_pl> '.'
            | 'Nachdem' <war_Bedingung_pl> ',' <DE3_f_pl> ',
                bevor' <ist_Bedingung_pl> '.'

<DEs5>  ::=   'Wenn irgendwann' <war_Bedingung_pl> ', dann' <DE3_f_pl> '.'
            | 'Nachdem' <war_Bedingung_pl> ',' <DE3_f_pl> '.'
            | <DEs3_b_pl> ', wenn irgendwann' <war_Bedingung_pl> '.'
            | <DEs3_b_pl> ', wenn vorher' <war_Bedingung_pl> '.'


<DEe1>  ::=   'Nur wenn’ <ist_Bedingung_pl> ', dann ' <DE1_f_pl> '.’
            | <DEe1_b_pl> ', wenn gleichzeitig’ <ist_Bedingung_pl> '.’

<DEe2>  ::=   'Nur wenn' <ist_Bedingung_pl> ', dann' <DE2_f_pl> '.'
            | <DEe2_b_pl> ', wenn' <ist_Bedingung_pl> '.'

<DEe3>  ::=   'Nur solange' <ist_Bedingung_pl> ',' <DE3_f_pl> '.'
            | <DEe3_b_pl> ', solange gleichzeitig' <ist_Bedingung_pl> '.'

<DEe4>  ::=   'Nur wenn irgendwann' <war_Bedingung_pl> ', dann' <DE3_f_pl> ',
                bevor' <ist_Bedingung_pl> '.'
            | 'Nur nachdem' <war_Bedingung_pl> ',' <DE3_f_pl> ',
                bevor' <ist_Bedingung_pl> '.'

<DEe5>  ::=   'Nur wenn irgendwann' <war_Bedingung_pl> ', dann' <DE3_f_pl> '.'
            | 'Nur nachdem' <war_Bedingung_pl> ',' <DE3_f_pl> '.'
            | <DEe3_b_pl>  ', wenn irgendwann' <war_Bedingung_pl> '.'
            | <DEe3_b_pl>  ', wenn vorher' <war_Bedingung_pl> '.'


<POe1>  ::=   'Nur wenn' <ist_Bedingung_pl> ', dann' <PO1_f_pl> '.'
            | <POe1_b_pl> ', wenn gleichzeitig' <ist_Bedingung_pl> '.'

<POe3>  ::=   'Nur solange' <ist_Bedingung_pl> ',' <PO3_f_pl> '.'
            | <POe3_b_pl> ', solange gleichzeitig' <ist_Bedingung_pl> '.'

<POe4>  ::=   'Nur wenn irgendwann' <war_Bedingung_pl> ', dann' <PO3_f_pl> ',
                bis' <ist_Bedingung_pl> '.'
            | 'Nur nachdem' <war_Bedingung_pl> ', dann' <PO3_f_pl> ',
                bis' <ist_Bedingung_pl> '.'

<POe5>  ::=   'Nur wenn irgendwann ' <war_Bedingung_pl> ', dann' <PO3_f_pl> '.'
            | 'Nur nachdem ' <war_Bedingung_pl> ',' <PO3_f_pl> '.'
            | <POe3_b_pl> ', wenn irgendwann' <war_Bedingung_pl> '.'
            | <POe3_b_pl> ', wenn vorher' <war_Bedingung_pl> '.'
```

```
<PRs1>  ::=   'Wenn' <ist_Bedingung_pl> ', dann' <PR1_f_pl> '.'
            | <PRs1_b_pl> ', wenn' <ist_Bedingung_pl> '.'

<PRs3>  ::=   'Solange' <ist_Bedingung_pl> ',' <PR3_f_pl> '.'
            | <PRs3_b_pl> ', solange' <ist_Bedingung_pl> '.'

<PRs4>  ::=   'Wenn irgendwann' <war_Bedingung_pl> ', dann' <PR3_f_pl> ',
                bis' <ist_Bedingung_pl> '.'
            | 'Nachdem' <war_Bedingung_pl> ',' <PR3_f_pl> ',
                bis' <ist_Bedingung_pl> '.'

<PRs5>  ::=   'Wenn irgendwann' <war_Bedingung_pl> ', dann' <PR3_f_pl> '.'
            | 'Nachdem' <war_Bedingung_pl> ',' <PR3_f_pl> '.'
            | <PRs3_b_pl> ', wenn irgendwann' <war_Bedingung_pl> '.'
            | <PRs3_b_pl> ', wenn vorher' <war_Bedingung_pl> '.'
```

Ebene 4 - Definition der konjunktiven Verknüpfungen

```
<ist_Bedingung_pl>  ::=   <ist_Bedingung_sg>
                        | <ist_Bedingung_sg> ‘und‘ <ist_Bedingung_pl>

<war_Bedingung_pl>  ::=   <war_Bedingung_sg>
                        | <war_Bedingung_sg> ‘und‘ <war_Bedingung_pl>

        <DE1_f_pl>  ::=   <DE1_f_sg>
                        | <DE1_f_sg> ‘und‘ <DE1_f_pl>

       <DEs1_b_pl>  ::=   <DEs1_b_sg>
                        | <DEs1_b_sg> ‘und‘ <DEs1_b_pl>

       <DEe1_b_pl>  ::=   <DEe1_b_sg>
                        | <DEe1_b__sg> ‘und‘ <DEe1_b_pl>

        <DE2_f_pl>  ::=   <DE2_f_sg>
                        | <DE2_f_sg> ‘und‘ <DE2_f_pl>

       <DEs2_b_pl>  ::=   <DEs2_b_sg>
                        | <DEs2_b_sg> ‘und‘ <DEs2_b_pl>

       <DEe2_b_pl>  ::=   <DEe2_b_sg>
                        | <DEe2_b_sg> ‘und‘ <DEe2_b_pl>

        <DE3_f_pl>  ::=   <DE3_f_sg>
                        | <DE3_f_sg> ‘und‘ <DE3_f_pl>

       <DEs3_b_pl>  ::=   <DEs3_b_sg>
                        | <DEs3_b_sg> ‘und‘ <DEs3_b_pl>

       <DEe3_b_pl>  ::=   <DEe3_b_sg>
                        | <DEe3_b_sg> ‘und‘ <DEe3_b_pl>

        <PO1_f_pl>  ::=   <PO1_f_sg>
                        | <PO1_f_sg> ‘und‘ <PO1_f_pl>

       <POe1_b_pl>  ::=   <POe1_b_sg>
                        | <POe1_b_sg> ‘und‘ <POe1_b_pl>

        <PO3_f_pl>  ::=   <PO3_f_sg>
                        | <PO3_f_sg> ‘und‘ <PO3_f_pl>
```

```
<POe3_b_pl>  ::=   <POe3_b_sg>
                 | <POe3_b_sg> ‘und‘ <POe3_b_pl>

 <PR1_f_pl>  ::=   <PR1_f_sg>
                 | <PR1_f_sg> ‘und‘ <PR1_f_pl>

<PRs1_b_pl>  ::=   <PRs1_b_sg>
                 | <PRs1_b_sg> ‘und‘ <PRs1_b_pl>

 <PR3_f_pl>  ::=   <PR3_f_sg>
                 | <PR3_f_sg> ‘und‘ <PR3_f_pl>

<PRs3_b_pl>  ::=   <PRs3_b_sg>
                 | <PRs3_b_sg> ‘und‘ <PRs3_b_pl>
```

Ebene 5 - ausformulierte Satzteile

```
<ist_Bedingung>  ::= <Nomen_Wert_Paar> 'ist'

<war_Bedingung>  ::= <Nomen_Wert_Paar> 'war'

 <DE1_f_sg>  ::=   'muss gleichzeitig' <Nomen_Wert_Paar> 'sein'
                 | 'muss gleichzeitig' <Nomen_Wert_Paar> 'bleiben'

<DEs1_b_sg>  ::=   <Nomen_Wert_Paar> 'muss gleichzeitig sein'
                 | <Nomen_Wert_Paar> 'muss gleichzeitig bleiben'

<DEe1_b_sg>  ::=   <Nomen_Wert_Paar> 'muss nur sein'
                 | <Nomen_Wert_Paar> 'muss nur bleiben'

 <DE2_f_sg>  ::=   'muss unmittelbar' <Nomen_Wert_Paar> 'werden'
                 | 'muss sofort' <Nomen_Wert_Paar> 'werden'
                 | wird unmittelbar' <Nomen_Wert_Paar>
                 | 'wird sofort' <Nomen_Wert_Paar>

<DEs2_b_sg >  ::=   <Nomen_Wert_Paar> 'muss unmittelbar werden'
                 | <Nomen_Wert_Paar> 'muss sofort werden'
                 | <Nomen_Wert_Paar> 'wird unmittelbar'
                 | <Nomen_Wert_Paar> 'wird sofort'

<DEe2_b_sg>  ::=   <Nomen_Wert_Paar> 'muss nur unmittelbar werden'
                 | <Nomen_Wert_Paar> 'muss nur sofort werden'
                 | <Nomen_Wert_Paar> 'wird nur unmittelbar'
                 | <Nomen_Wert_Paar> 'wird nur sofort'

 <DE3_f_sg>  ::=   'muss irgendwann' <Nomen_Wert_Paar> 'werden'
                 | 'wird irgendwann' <Nomen_Wert_Paar>

<DEs3_b_sg>  ::=   <Nomen_Wert_Paar> 'muss irgendwann werden'
                 | <Nomen_Wert_Paar> 'wird irgendwann'

<DEe3_b_sg>  ::=   <Nomen_Wert_Paar> 'muss nur irgendwann werden'
                 | <Nomen_Wert_Paar> 'wird nur irgendwann'

 <PO1_f_sg>  ::=   'kann gleichzeitig' <Nomen_Wert_Paar> 'sein'
                 | 'darf gleichzeitig' <Nomen_Wert_Paar> 'sein'
```

```
<POe1_b_sg>  ::=   <Nomen_Wert_Paar> 'kann nur sein'
                 | <Nomen_Wert_Paar> 'darf nur sein'

 <PO3_f_sg>  ::=   'kann irgendwann' <Nomen_Wert_Paar> 'werden'
                 | 'darf irgendwann' <Nomen_Wert_Paar> 'werden'

<POe3_b_sg>  ::=   <Nomen_Wert_Paar> 'kann nur irgendwann werden'
                 | <Nomen_Wert_Paar> 'darf nur irgendwann werden'

 <PR1_f_sg>  ::=   'darf nicht gleichzeitig' <Nomen_Wert_Paar> 'sein'
                 | 'darf niemals gleichzeitig' <Nomen_Wert_Paar> 'sein'

<PRs1_b_sg>  ::=   <Nomen_Wert_Paar> 'darf nicht gleichzeitig sein'
                 | <Nomen_Wert_Paar> 'darf niemals gleichzeitig sein'

 <PR3_f_sg>  ::=   'darf nicht irgendwann' <Nomen_Wert_Paar> 'werden'
                 | 'darf niemals' <Nomen_Wert_Paar> 'werden'

<PRs3_b_sg>  ::=   <Nomen_Wert_Paar> 'darf nicht irgendwann werden'
                 | <Nomen_Wert_Paar> 'darf niemals werden'
```

A.2 Erzeugbare Beispielsätze

Ausformulierte Satzteile in Ebene 5, es werden Sätze mit jeweils zwei konjunktiv verknüpften Bedingungen (B1 und B2) bzw. Folgerungen (F1 und F2) dargestellt (selbstverständlich ist die Anzahl der verwendeten Bedingungen / Folgerungen beliebig)

Kategorie DEs1 - einfache Forderung, Zustand

Wenn B1 ist und B2 ist, dann muss gleichzeitig F1 sein und muss gleichzeitig F2 sein.

Wenn B1 ist und B2 ist, dann muss gleichzeitig F1 bleiben und muss gleichzeitig F2 bleiben.

F1 muss gleichzeitig sein und F2 muss gleichzeitig sein, wenn B1 ist und B2 ist.

F1 muss gleichzeitig bleiben und F2 muss gleichzeitig bleiben, wenn B1 ist und B2 ist.

Kategorie DEs2 - einfache Forderung, direkt

Wenn B1 ist und B2 ist, dann muss unmittelbar F1 werden und muss unmittelbar F2 werden.

Wenn B1 ist und B2 ist, dann muss sofort F1 werden und muss sofort F2 werden.

Wenn B1 ist und B2 ist, dann wird unmittelbar F1 und wird unmittelbar F2.

Wenn B1 ist und B2 ist, dann wird sofort F1 und wird sofort F2.

F1 muss unmittelbar werden und F2 muss unmittelbar werden, wenn B1 ist und B2 ist.

F1 muss sofort werden und F2 muss sofort werden, wenn B1 ist und B2 ist.

F1 wird unmittelbar und F2 wird unmittelbar, wenn B1 ist und B2 ist.

F1 wird sofort und F2 wird sofort, wenn B1 ist und B2 ist.

Kategorie DEs3 - einfache Forderung, selbstbegrenzt

Solange B1 ist und B2 ist, muss irgendwann F1 werden und muss irgendwann F2 werden.

Solange B1 ist und B2 ist, wird irgendwann F1 und wird irgendwann F2.

F1 muss irgendwann werden und F2 muss irgendwann werden, solange B1 ist und B2 ist.

F1 wird irgendwann und F2 wird irgendwann, solange B1 ist und B2 ist.

Kategorie DEs4 - einfache Forderung, fremdbegrenzt

Wenn irgendwann B1 war und B2 war, dann muss irgendwann F1 werden und muss irgendwann F2 werden, bevor B3 ist und B4 ist.

Wenn irgendwann B1 war und B2 war, dann wird irgendwann F1 und wird irgendwann F2, bevor B3 ist und B4 ist.

Nachdem B1 war und B2 war, muss irgendwann F1 werden und muss irgendwann F2 werden, bevor B3 ist und B4 ist.

Nachdem B1 war und B2 war, wird irgendwann F1 und wird irgendwann F2, bevor B3 ist und B4 ist.

Kategorie DEs5 - einfache Forderung, unbegrenzt

Wenn irgendwann B1 war und B2 war, dann muss irgendwann F1 werden und muss irgendwann F2 werden.

Wenn irgendwann B1 war und B2 war, dann wird irgendwann F1 und wird irgendwann F2.

Nachdem B1 war und B2 war, muss irgendwann F1 werden und muss irgendwann F2 werden.

Nachdem B1 war und B2 war, wird irgendwann F1 und wird irgendwann F2.

F1 muss irgendwann werden und F2 muss irgendwann werden, wenn irgendwann B1 war und B2 war.

F1 wird irgendwann und F2 wird irgendwann, wenn irgendwann B1 war und B2 war.

F1 muss irgendwann werden und F2 muss irgendwann werden, wenn vorher B1 war und B2 war.

F1 wird irgendwann und F2 wird irgendwann, wenn vorher B1 war und B2 war.

Kategorie DEe1 - erweiterte Forderung, Zustand

Nur wenn B1 ist und B2 ist, dann muss gleichzeitig F1 sein und muss gleichzeitig F2 sein.

Nur wenn B1 ist und B2 ist, dann muss gleichzeitig F1 bleiben und muss gleichzeitig F2 bleiben.

F1 muss nur sein und F2 muss nur sein, wenn gleichzeitig B1 ist und B2 ist.

F1 muss nur bleiben und F2 muss nur bleiben, wenn gleichzeitig B1 ist und B2 ist.

Kategorie DEe2 - erweiterte Forderung, direkt

Nur wenn B1 ist und B2 ist, dann muss unmittelbar F1 werden und muss unmittelbar F2 werden.

Nur wenn B1 ist und B2 ist, dann muss sofort F1 werden und muss sofort F2 werden.

Nur wenn B1 ist und B2 ist, dann wird unmittelbar F1 und wird unmittelbar F2.

Nur wenn B1 ist und B2 ist, dann wird sofort F1 und wird sofort F2.

F1 muss nur unmittelbar werden und F2 muss nur unmittelbar werden, wenn B1 ist und B2 ist.

F1 muss nur sofort werden und F2 muss nur sofort werden, wenn B1 ist und B2 ist.

F1 wird nur unmittelbar und F2 wird nur unmittelbar, wenn B1 ist und B2 ist.

F1 wird nur sofort und F2 wird nur sofort, wenn B1 ist und B2 ist.

Kategorie DEe3 - erweiterte Forderung, selbstbegrenzt

Nur solange B1 ist und B2 ist, muss irgendwann F1 werden und muss irgendwann F2 werden.

Nur solange B1 ist und B2 ist, wird irgendwann F1 und wird irgendwann F2.

F1 muss nur irgendwann werden und F2 muss nur irgendwann werden, solange gleichzeitig B1 ist und B2 ist.

F1 wird nur irgendwann und F2 wird nur irgendwann, solange gleichzeitig B1 ist und B2 ist.

Kategorie DEe4 - erweiterte Forderung, fremdbegrenzt

Nur wenn irgendwann B1 war und B2 war, dann muss irgendwann F1 werden und muss irgendwann F2 werden, bevor B3 ist und B4 ist.

Nur wenn irgendwann B1 war und B2 war, dann wird irgendwann F1 und wird irgendwann F2, bevor B3 ist und B4 ist.

Nur nachdem B1 war und B2 war, muss irgendwann F1 werden und muss irgendwann F2 werden, bevor B3 ist und B4 ist.

Nur nachdem B1 war und B2 war, wird irgendwann F1 und wird irgendwann F2, bevor B3 ist und B4 ist.

Kategorie DEe5 - erweiterte Forderung, unbegrenzt

Nur wenn irgendwann B1 war und B2 war, dann muss irgendwann F1 werden und muss irgendwann F2 werden.

Nur wenn irgendwann B1 war und B2 war, dann wird irgendwann F1 und wird irgendwann F2.

Nur nachdem B1 war und B2 war, muss irgendwann F1 werden und muss irgendwann F2 werden.

Nur nachdem B1 war und B2 war, wird irgendwann F1 und wird irgendwann F2.

F1 muss nur irgendwann werden und F2 muss nur irgendwann werden, wenn irgendwann B1 war und B2 war.

F1 wird nur irgendwann und F2 wird nur irgendwann, wenn irgendwann B1 war und B2 war.

F1 muss nur irgendwann werden und F2 muss nur irgendwann werden, wenn vorher B1 war und B2 war.

F1 wird nur irgendwann und F2 wird nur irgendwann, wenn vorher B1 war und B2 war.

Kategorie POe1 - erweiterte Möglichkeit, Zustand

Nur wenn B1 ist und B2 ist, dann kann gleichzeitig F1 sein und kann gleichzeitig F2 sein.

Nur wenn B1 ist und B2 ist, dann darf gleichzeitig F1 sein und darf gleichzeitig F2 sein.

F1 kann nur sein und F2 kann nur sein, wenn gleichzeitig B1 ist und B2 ist.

F1 darf nur sein und F2 darf nur sein, wenn gleichzeitig B1 ist und B2 ist.

Kategorie POe3 - erweiterte Möglichkeit, selbstbegrenzt

Nur solange B1 ist und B2 ist, kann irgendwann F1 werden und kann irgendwann F2 werden.

Nur solange B1 ist und B2 ist, darf irgendwann F1 werden und darf irgendwann F2 werden.

F1 kann nur irgendwann werden und F2 kann nur irgendwann werden, solange gleichzeitig B1 ist und B2 ist.

F1 darf nur irgendwann werden und F2 darf nur irgendwann werden, solange gleichzeitig B1 ist und B2 ist.

Kategorie POe4 - erweiterte Möglichkeit, fremdbegrenzt

Nur wenn irgendwann B1 war und B2 war, dann kann irgendwann F1 werden und kann irgendwann F2 werden, bis B3 ist und B4 ist.

Nur wenn irgendwann B1 war und B2 war, dann darf irgendwann F1 werden und darf irgendwann F2 werden, bis B3 ist und B4 ist.

Nur nachdem B1 war und B2 war, dann kann irgendwann F1 werden und kann irgendwann F2 werden, bis B3 ist und B4 ist.

Nur nachdem B1 war und B2 war, dann darf irgendwann F1 werden und darf irgendwann F2 werden, bis B3 ist und B4 ist.

Kategorie POe5 - erweiterte Möglichkeit, unbegrenzt

Nur wenn irgendwann B1 war und B2 war, dann kann irgendwann F1 werden und kann irgendwann F2 werden.

Nur wenn irgendwann B1 war und B2 war, dann darf irgendwann F1 werden und darf irgendwann F2 werden.

Nur nachdem B1 war und B2 war, kann irgendwann F1 werden und kann irgendwann F2 werden.

Nur nachdem B1 war und B2 war, darf irgendwann F1 werden und darf irgendwann F2 werden.

F1 kann nur irgendwann werden und F2 kann nur irgendwann werden, wenn irgendwann B1 war und B2 war.

F1 darf nur irgendwann werden und F2 darf nur irgendwann werden, wenn irgendwann B1 war und B2 war.

F1 kann nur irgendwann werden und F2 kann nur irgendwann werden, wenn vorher B1 war und B2 war.

F1 darf nur irgendwann werden und F2 darf nur irgendwann werden, wenn vorher B1 war und B2 war.

Kategorie PRs1 - einfaches Verbot, Zustand

Wenn B1 ist und B2 ist, dann darf nicht gleichzeitig F1 sein und darf nicht gleichzeitig F2 sein.

Wenn B1 ist und B2 ist, dann darf niemals gleichzeitig F1 sein und darf niemals gleichzeitig F2 sein.

F1 darf nicht gleichzeitig sein und F2 darf nicht gleichzeitig sein, wenn B1 ist und B2 ist.

F1 darf niemals gleichzeitig sein und F2 darf niemals gleichzeitig sein, wenn B1 ist und B2 ist.

Kategorie PRs3 - einfaches Verbot, selbstbegrenzt

Solange B1 ist und B2 ist, darf nicht irgendwann F1 werden und darf nicht irgendwann F2 werden.

Solange B1 ist und B2 ist, darf niemals F1 werden und darf niemals F2 werden.

F1 darf nicht irgendwann werden und F2 darf nicht irgendwann werden, solange B1 ist und B2 ist.

F1 darf niemals werden und F2 darf niemals werden, solange B1 ist und B2 ist.

Kategorie PRs4 - einfaches Verbot, fremdbegrenzt

Wenn irgendwann B1 war und B2 war, dann darf nicht irgendwann F1 werden und darf nicht irgendwann F2 werden, bis B3 ist und B4 ist.

Wenn irgendwann B1 war und B2 war, dann darf niemals F1 werden und darf niemals F2 werden, bis B3 ist und B4 ist.

Nachdem B1 war und B2 war, darf nicht irgendwann F1 werden und darf nicht irgendwann F2 werden, bis B3 ist und B4 ist.

Nachdem B1 war und B2 war, darf niemals F1 werden und darf niemals F2 werden, bis B3 ist und B4 ist.

Kategorie PRs5 - einfaches Verbot, unbegrenzt

Wenn irgendwann B1 war und B2 war, dann darf nicht irgendwann F1 werden und darf nicht irgendwann F2 werden.

Wenn irgendwann B1 war und B2 war, dann darf niemals F1 werden und darf niemals F2 werden.

Nachdem B1 war und B2 war, darf nicht irgendwann F1 werden und darf nicht irgendwann F2 werden.

Nachdem B1 war und B2 war, darf niemals F1 werden und darf niemals F2 werden.

F1 darf nicht irgendwann werden und F2 darf nicht irgendwann werden, wenn irgendwann B1 war und B2 war.

F1 darf niemals werden und F2 darf niemals werden, wenn irgendwann B1 war und B2 war.

F1 darf nicht irgendwann werden und F2 darf nicht irgendwann werden, wenn vorher B1 war und B2 war.

F1 darf niemals werden und F2 darf niemals werden, wenn vorher B1 war und B2 war.

A.3 Struktur der Datei zur Definition und Zuordnung der Nomen und Werte

```
<Zuordnung>  ::= %%<Nr>-<Datentyp>-<Variablentyp>
                 „<SPS-Variable>“:„<Nomen>“
                 <Wertzuordnung>

<Nr>  ::= ’1’, ’2’, ’3’, ...

<Datentyp>  ::= BOOL | INT | REAL

<Variablentyp>  ::=   VAR
                    | VAR_ACCESS
                    | VAR_EXTERNAL
                    | VAR_GLOBAL
                    | VAR_IN_OUT
                    | VAR_INPUT
                    | VAR_OUTPUT

<SPS_Variable>  ::= undefined

<Nomen>  ::= undefined

<Wertzuordnung>  ::=   <Wertzuordnung_BOOL>
                     | <Wertzuordnung_INT>

<Wertzuordnung_BOOL>  ::= TRUE_I:„<Wert>“
                          TRUE_O:„<Wert>“
                          FALSE_I:„<Wert>“
                          FALSE_O:„<Wert>“

<Wertzuordnung_INT>  ::= <Wert1>_I:„<Wert>“
                         <Wert1>_O:„<Wert>“
                         <Wert2>_I:„<Wert>“
                         <Wert2>_O:„<Wert>“
                         …
```

A.4 Formale Definition des PreLexers

Ebene a - grundlegende Satzkonstruktionen

```
<statement> ::=   'Wenn' <Bedingungen> ', dann' <f_Folgerungen> '.'
                | 'Wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> '.'
                | 'Wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> ', bevor' <Bedingungen> '.'
                | 'Wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> ', bis' <Bedingungen> '.'
                | 'Nur wenn' <Bedingungen> ', dann ' <f_Folgerungen> '.'
                | 'Nur wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> '.'
                | 'Nur wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> ', bevor' <Bedingungen> '.'
                | 'Nur wenn irgendwann' <Bedingungen> ', dann' <f_Folgerungen> ', bis' <Bedingungen> '.'
                | 'Nachdem' <Bedingungen> ',' <f_Folgerungen> '.'
                | 'Nachdem' <Bedingungen> ',' <f_Folgerungen> ', bevor' <Bedingungen> '.'
                | 'Nachdem' <Bedingungen> ',' <f_Folgerungen> ', bis' <Bedingungen> '.'
                | 'Nur nachdem' <Bedingungen> ',' <f_Folgerungen> '.'
                | 'Nur nachdem' <Bedingungen> ',' <f_Folgerungen> ', bevor' <Bedingungen> '.'
                | 'Nur nachdem' <Bedingungen> ', dann' <f_Folgerungen> ', bis' <Bedingungen> '.'
                | 'Solange' <Bedingungen> ',' <f_Folgerungen> '.'
                | 'Nur solange' <Bedingungen> ',' <f_Folgerungen> '.'
                | <b_Folgerungen> ', wenn' <Bedingungen> '.'
                | <b_Folgerungen> ', wenn gleichzeitig' <Bedingungen> '.'
                | <b_Folgerungen> ', wenn irgendwann' <Bedingungen> '.'
                | <b_Folgerungen> ', wenn vorher' <Bedingungen> '.'
                | <b_Folgerungen> ', solange' <Bedingungen> '.'
                | <b_Folgerungen> ', solange gleichzeitig' <Bedingungen> '.'
```

Ebene b - konjunktive Verknüpfungen

```
  <Bedingungen> ::=   <Anfangsbedingung>
                    | <Anfangsbedingung> 'und' <Folgebedingungen>

<f_Folgerungen> ::=   <f_Folgerung>
                    | <f_Folgerung> 'und' <f_Folgerungen>

<b_Folgerungen> ::=   <b_Folgerung>
                    | <b_Folgerung> 'und' <b_Folgerungen>
```

Ebene c

```
<Anfangsbedingung> ::=   <Nomen> <Wert> 'ist'
                       | <Nomen> <Wert> 'war'

<Folgebedingungen> ::=   <Anfangsbedingung>
                       | <Folgebedingung>
                       | <Anfangsbedingung> 'und' <Folgebedingungen>
                       | <Folgebedingung> 'und' <Folgebedingungen>

  <Folgebedingung> ::=   <Nomen> 'ist' <Wert>
                       | <Nomen> 'war' <Wert>


     <f_Folgerung> ::=   <Nomen> 'muss gleichzeitig' <Wert> 'sein'
                       | 'muss' <Nomen> 'gleichzeitig' <Wert> 'sein'
                       | 'muss gleichzeitig' <Nomen> <Wert> 'sein'
                       | <Nomen> 'muss gleichzeitig' <Wert> 'bleiben'
                       | 'muss' <Nomen> 'gleichzeitig' <Wert> 'bleiben'
                       | 'muss gleichzeitig' <Nomen> <Wert> 'bleiben'
                       | <Nomen> 'muss unmittelbar' <Wert> 'werden'
                       | 'muss' <Nomen> 'unmittelbar' <Wert> 'werden'
                       | 'muss unmittelbar' <Nomen> <Wert> 'werden'
                       | <Nomen> 'muss sofort' <Wert> 'werden'
                       | 'muss' <Nomen> 'sofort' <Wert> 'werden'
                       | 'muss sofort' <Nomen> <Wert> 'werden'
                       | <Nomen> 'wird unmittelbar' <Wert>
                       | 'wird' <Nomen> 'unmittelbar' <Wert>
                       | 'wird unmittelbar' <Nomen> <Wert>
                       | <Nomen> 'wird sofort' <Wert>
                       | 'wird' <Nomen> 'sofort' <Wert>
                       | 'wird sofort' <Nomen> <Wert>
                       | <Nomen> 'muss irgendwann' <Wert> 'werden'
                       | 'muss' <Nomen> 'irgendwann' <Wert> 'werden
                       | 'muss irgendwann' <Nomen> <Wert> 'werden
                       | <Nomen> 'wird irgendwann' <Wert>
                       | 'wird' <Nomen> 'irgendwann' <Wert>
                       | 'wird irgendwann' <Nomen> <Wert>
                       | <Nomen> 'kann gleichzeitig' <Wert> 'sein'
                       | 'kann' <Nomen> 'gleichzeitig' <Wert> 'sein'
                       | 'kann gleichzeitig' <Nomen> <Wert> 'sein'
                       | <Nomen> 'darf gleichzeitig' <Wert> 'sein'
                       | 'darf' <Nomen> 'gleichzeitig' <Wert> 'sein'
                       | 'darf gleichzeitig' <Nomen> <Wert> 'sein'
                       | <Nomen> 'kann irgendwann' <Wert> 'werden'
                       | 'kann' <Nomen> 'irgendwann' <Wert> 'werden'
                       | 'kann irgendwann' <Nomen> <Wert> 'werden'
                       | <Nomen> 'darf irgendwann' <Wert> 'werden'
                       | 'darf' <Nomen> 'irgendwann' <Wert> 'werden'
                       | 'darf irgendwann' <Nomen> <Wert> 'werden'
                       | <Nomen> 'darf nicht gleichzeitig' <Wert> 'sein'
                       | 'darf' <Nomen> 'nicht gleichzeitig' <Wert> 'sein'
                       | 'darf nicht gleichzeitig' <Nomen> <Wert> 'sein'
                       | <Nomen> 'darf niemals gleichzeitig' <Wert> 'sein'
                       | 'darf' <Nomen> 'niemals gleichzeitig' <Wert> 'sein'
                       | 'darf niemals gleichzeitig' <Nomen> <Wert> 'sein'
```

```
                        | <Nomen> 'darf nicht irgendwann' <Wert> 'werden'
                        | 'darf' <Nomen> 'nicht irgendwann' <Wert> 'werden'
                        | 'darf nicht irgendwann' <Nomen> <Wert> 'werden'
                        | <Nomen> 'darf niemals' <Wert> 'werden'
                        | 'darf' <Nomen> 'niemals' <Wert> 'werden'
                        | 'darf niemals' <Nomen> <Wert> 'werden'

<b_Folgerung> ::=       <Nomen> 'muss gleichzeitig ' <Wert> 'sein'
                        | <Nomen> 'muss gleichzeitig ' <Wert> 'bleiben'
                        | <Nomen> 'muss nur' <Wert> 'sein'
                        | <Nomen> 'muss nur' <Wert> 'bleiben'
                        | <Nomen> 'muss unmittelbar' <Wert> 'werden'
                        | <Nomen> 'muss sofort' <Wert> 'werden'
                        | <Nomen> 'wird unmittelbar' <Wert>
                        | <Nomen> 'wird sofort <Wert>
                        | <Nomen> 'muss nur unmittelbar' <Wert> 'werden'
                        | <Nomen> 'muss nur sofort' <Wert> 'werden'
                        | <Nomen> 'wird nur unmittelbar' <Wert>
                        | <Nomen> 'wird nur sofort' <Wert>
                        | <Nomen> 'muss irgendwann' <Wert> 'werden'
                        | <Nomen> 'wird irgendwann' <Wert>
                        | <Nomen> 'muss nur irgendwann' <Wert> 'werden'
                        | <Nomen> 'wird nur irgendwann' <Wert>
                        | <Nomen> 'kann gleichzeitig' <Wert> 'sein'
                        | <Nomen> 'darf gleichzeitig' <Wert> 'sein'
                        | <Nomen> 'kann nur' <Wert> 'sein'
                        | <Nomen> 'darf nur' <Wert> 'sein'
                        | <Nomen> 'kann irgendwann' <Wert> 'werden'
                        | <Nomen> 'darf irgendwann' <Wert> 'werden'
                        | <Nomen> 'kann nur irgendwann' <Wert> 'werden'
                        | <Nomen> 'darf nur irgendwann' <Wert> 'werden'
                        | <Nomen> 'darf nicht gleichzeitig' <Wert> 'sein'
                        | <Nomen> 'darf niemals gleichzeitig' <Wert> 'sein'
                        | <Nomen> 'darf nur nicht gleichzeitig' <Wert> 'sein'
                        | <Nomen> 'darf nicht irgendwann' <Wert> 'werden'
                        | <Nomen> 'darf niemals' <Wert> 'werden'
                        | <Nomen> 'darf nur nicht irgendwann' <Wert> 'werden'
                        | <Nomen> 'darf nur niemals' <Wert> 'werden
```

Umsetzung (Nomen und Werte werden umgestellt und zusammengefasst)

<Nomen> <Wert> 'ist' <Nomen> 'ist' <Wert>	⇒	<Nomen_Wert_Paar> 'ist'
<Nomen> <Wert> 'war' <Nomen> 'war' <Wert>	⇒	<Nomen_Wert_Paar> 'war'
<Nomen> 'muss gleichzeitig' <Wert> 'sein' 'muss' <Nomen> 'gleichzeitig' <Wert> 'sein' 'muss gleichzeitig' <Nomen> <Wert> 'sein'	⇒	'muss gleichzeitig' <Nomen_Wert_Paar> 'sein'
<Nomen> 'muss gleichzeitig' <Wert> 'bleiben' 'muss' <Nomen> 'gleichzeitig' <Wert> 'bleiben' 'muss gleichzeitig' <Nomen> <Wert> 'bleiben'	⇒	'muss gleichzeitig' <Nomen_Wert_Paar> 'bleiben'

<Nomen> 'muss gleichzeitig ' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'muss gleichzeitig sein'
<Nomen> 'muss gleichzeitig ' <Wert> 'bleiben'	⇒	<Nomen_Wert_Paar> 'muss gleichzeitig bleiben'
<Nomen> 'muss nur' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'muss nur sein'
<Nomen> 'muss nur' <Wert> 'bleiben'	⇒	<Nomen_Wert_Paar> 'muss nur bleiben'
<Nomen> 'muss unmittelbar' <Wert> 'werden' 'muss' <Nomen> 'unmittelbar' <Wert> 'werden' 'muss unmittelbar' <Nomen> <Wert> 'werden'	⇒	'muss unmittelbar' <Nomen_Wert_Paar> 'werden'
<Nomen> 'muss sofort' <Wert> 'werden' 'muss' <Nomen> 'sofort' <Wert> 'werden' 'muss sofort' <Nomen> <Wert> 'werden'	⇒	'muss sofort' <Nomen_Wert_Paar> 'werden'
<Nomen> 'wird unmittelbar' <Wert> 'wird' <Nomen> 'unmittelbar' <Wert> 'wird unmittelbar' <Nomen> <Wert>	⇒	'wird unmittelbar' <Nomen_Wert_Paar>
<Nomen> 'wird sofort' <Wert> 'wird' <Nomen> 'sofort' <Wert> 'wird sofort' <Nomen> <Wert>	⇒	'wird sofort' <Nomen_Wert_Paar>
<Nomen> 'muss unmittelbar' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss unmittelbar werden'
<Nomen> 'muss sofort' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss sofort werden'
<Nomen> 'wird unmittelbar' <Wert>	⇒	<Nomen_Wert_Paar> 'wird unmittelbar'
<Nomen> 'wird sofort <Wert>	⇒	<Nomen_Wert_Paar> 'wird sofort'
<Nomen> 'muss nur unmittelbar' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss nur unmittelbar werden'
<Nomen> 'muss nur sofort' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss nur sofort werden'
<Nomen> 'wird nur unmittelbar' <Wert>	⇒	<Nomen_Wert_Paar> 'wird nur unmittelbar'
<Nomen> 'wird nur sofort' <Wert>	⇒	<Nomen_Wert_Paar> 'wird nur sofort
<Nomen> 'muss irgendwann' <Wert> 'werden' 'muss' <Nomen> 'irgendwann' <Wert> 'werden' 'muss irgendwann' <Nomen> <Wert> 'werden'	⇒	'muss irgendwann' <Nomen_Wert_Paar> 'werden'
<Nomen> 'wird irgendwann' <Wert> 'wird' <Nomen> 'irgendwann' <Wert> 'wird irgendwann' <Nomen> <Wert>	⇒	'wird irgendwann' <Nomen_Wert_Paar>
<Nomen> 'muss irgendwann' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss irgendwann werden'
<Nomen> 'wird irgendwann' <Wert>	⇒	<Nomen_Wert_Paar> 'wird irgendwann'
<Nomen> 'muss nur irgendwann' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'muss nur irgendwann werden'
<Nomen> 'wird nur irgendwann' <Wert>	⇒	<Nomen_Wert_Paar> 'wird nur irgendwann'

<Nomen> 'kann nur' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'kann nur sein'
<Nomen> 'darf nur' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'darf nur sein'
<Nomen> 'kann nur irgendwann' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'kann nur irgendwann werden'
<Nomen> 'darf nur irgendwann' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'darf nur irgendwann werden'
<Nomen> 'darf nicht gleichzeitig' <Wert> 'sein' 'darf' <Nomen> 'nicht gleichzeitig' <Wert> 'sein' 'darf nicht gleichzeitig' <Nomen> <Wert> 'sein'	⇒	'darf nicht gleichzeitig' <Nomen_Wert_Paar> 'sein'
<Nomen> 'darf niemals gleichzeitig' <Wert> 'sein' 'darf' <Nomen> 'niemals gleichzeitig' <Wert> 'sein' 'darf niemals gleichzeitig' <Nomen> <Wert> 'sein'	⇒	'darf niemals gleichzeitig' <Nomen_Wert_Paar> 'sein'
<Nomen> 'darf nicht gleichzeitig' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'darf nicht gleichzeitig sein'
<Nomen> 'darf niemals gleichzeitig' <Wert> 'sein'	⇒	<Nomen_Wert_Paar> 'darf niemals gleichzeitig sein'
<Nomen> 'darf nicht irgendwann' <Wert> 'werden' 'darf' <Nomen> 'nicht irgendwann' <Wert> 'werden' 'darf nicht irgendwann' <Nomen> <Wert> 'werden'	⇒	'darf nicht irgendwann' <Nomen_Wert_Paar> 'werden'
<Nomen> 'darf niemals' <Wert> 'werden' 'darf' <Nomen> 'niemals' <Wert> 'werden' 'darf niemals' <Nomen> <Wert> 'werden'	⇒	'darf niemals' <Nomen_Wert_Paar> 'werden'
<Nomen> 'darf nicht irgendwann' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'darf nicht irgendwann werden'
<Nomen> 'darf niemals' <Wert> 'werden'	⇒	<Nomen_Wert_Paar> 'darf niemals werden'

A.5 Erzeugbare Beispielsätze auf verbaler Ebene

An dieser Stelle soll noch einmal kurz der Sprachumfang der Sicherheitsfachsprache gezeigt werden. Es werden bewusst nur Beispielsätze der Kategorie DEs1 (einfache Forderung, Zustand) dargestellt, um den Rahmen dieser Darstellung nicht zu sprengen. Der Leser wird sich sicher vorstellen können, wie viele Möglichkeiten der Bildung korrekter Sätze es gibt.

In Anhang A.2 wurden in der genannten Kategorie vier Mustersätze gezeigt. Durch Aufteilung des *Nomen-Wert-Paares* (dargestellt durch *B1*, *B2*, *F1*, *F2*) in einzelne *Nomen* und *Werte* (hier unspezifisch dargestellt als „*x1*“, „*x2*“, „*y1*“ sowie „*y2*“ und „*True*“) und zusätzliche Umstellung der Wortreihenfolge lassen sich die nachfolgend gezeigten verbalen Anforderungssätze erzeugen.

Wenn B1 ist und B2 ist, dann muss gleichzeitig F1 sein und muss gleichzeitig F2 sein.

- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss gleichzeitig „y1“ „True“ sein und muss gleichzeitig „y2“ „True“ sein.*
- *Wenn „x1“ „True“ ist und „x2“ „True“ ist, dann muss gleichzeitig „y1“ „True“ sein und muss gleichzeitig „y2“ „True“ sein.*
- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss „y1“ gleichzeitig „True“ sein und muss „y2“ gleichzeitig „True“ sein.*
- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss „y1“ gleichzeitig „True“ sein und „y2“ muss gleichzeitig „True“ sein.*

Wenn B1 ist und B2 ist, dann muss gleichzeitig F1 bleiben und muss gleichzeitig F2 bleiben.

- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss gleichzeitig „y1“ „True“ bleiben und muss gleichzeitig „y2“ „True“ bleiben.*
- *Wenn „x1“ „True“ ist und „x2“ „True“ ist, dann muss gleichzeitig „y1“ „True“ bleiben und muss gleichzeitig „y2“ „True“ bleiben.*
- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss „y1“ gleichzeitig „True“ bleiben und muss „y2“ gleichzeitig „True“ bleiben.*
- *Wenn „x1“ „True“ ist und „x2“ ist „True“, dann muss „y1“ gleichzeitig „True“ bleiben und „y2“ muss gleichzeitig „True“ bleiben.*

F1 muss gleichzeitig sein und F2 muss gleichzeitig sein, wenn B1 ist und B2 ist.

- *„y1“ muss gleichzeitig „True“ sein und „y2“ muss gleichzeitig „True“ sein, wenn „x1“ „True“ ist und „x2“ „True“ ist.*
- *„y1“ muss gleichzeitig „True“ sein und „y2“ muss gleichzeitig „True“ sein, wenn „x1“ „True“ ist und „x2“ ist „True“.*

F1 muss gleichzeitig bleiben und F2 muss gleichzeitig bleiben, wenn B1 ist und B2 ist.

- *„y1“ muss gleichzeitig „True“ bleiben und „y2“ muss gleichzeitig „True“ bleiben, wenn „x1“ „True“ ist und „x2“ „True“ ist.*
- *„y1“ muss gleichzeitig „True“ bleiben und „y2“ muss gleichzeitig „True“ bleiben, wenn „x1“ „True“ ist und „x2“ ist „True“.*

A.6 Zusammenfassung: Kategorien der SFS

Die Bestimmung der Kategorie erfolgt durch zwei Parameter:

1. Modalparameter
2. Zeitparameter

Eine Anforderung besteht aus zwei Teilen: einem Bedingungsteil *B* sowie einem Folgerungsteil *F*, diese sind implikativ verbunden.

Durch den Bedingungsteil wird ein Beobachtungsintervall *BI* definiert, dessen Länge durch den Zeitparameter bestimmt wird.

Der Modalparameter bestimmt den Wahrheitswert der im Folgerungsteil benannten Aussage *F* in Bezug auf das Beobachtungsintervall *BI*.

Einfache Forderung	**Einfaches Verbot**
innerhalb des *BI* **muss** es ein *F* geben außerhalb des *BI* darf es ein *F* geben	innerhalb des *BI* **darf** es **kein** *F* geben außerhalb des *BI* darf es ein *F* geben
Erweiterte Forderung	**Erweiterte Möglichkeit**
innerhalb des *BI* **muss** es ein *F* geben außerhalb des *BI* **darf** es **kein** *F* geben	innerhalb des *BI* darf es ein *F* geben außerhalb des *BI* **darf** es **kein** *F* geben

Die einzelnen Kategorien des Modalparameters lassen sich voneinander ableiten.

Ableitung 1: Bei der einfachen Forderung muss es innerhalb des Beobachtungsintervalls ein *F* geben.

Ableitung 2: Beim einfachen Verbot darf es innerhalb des Beobachtungsintervalls kein *F* geben (Umkehrung der einfachen Forderung)

Ableitung 3: Bei der erweiterten Möglichkeit darf es außerhalb des Beobachtungsintervalls kein *F* geben (Invertierung des Beobachtungsintervalls in Vergleich zum einfachen Verbot).

Ableitung 4: Die erweiterte Forderung kann man als Verbindung zwischen der einfachen Forderung (innerhalb des *BI* muss es ein *F* geben) und der erweiterten Möglichkeit (außerhalb des *BI* darf es kein *F* geben) betrachten.

Die Länge des Beobachtungsintervalls *BI* wird durch den Zeitparameter bestimmt. Es werden 5 Möglichkeiten unterschieden:

1. „Zustand“ - das *BI* umfasst genau einen Zustand, und zwar den durch *B* bestimmten Zustand.
2. „direkt“ - Das *BI* umfasst genau einen Zustand, und zwar den, in dem nachfolgend erst mal die Variable rdy_plc gesetzt wird.
3. „selbstbegrenzt“ - Das *BI* umfasst alle diejenigen aufeinanderfolgenden Zustände, in denen *B* erfüllt wird. Das *BI* beginnt mit der erstmaligen Erfüllung von *B* und endet, sobald *B* nicht mehr erfüllt ist.

4. „fremdbegrenzt“ - Das *BI* beginnt mit der Erfüllung einer Bedingung B_1 und endet mit der Erfüllung einer weiteren Bedingung B_2.
5. „unbegrenzt“ - Das *BI* beginnt wiederum mit der Erfüllung von *B*, hat jedoch kein festgelegtes Ende.

Innerhalb der Sicherheitsfachsprache sind insgesamt 18 Anforderungskategorien definiert.

Einfache Forderung		**Einfaches Verbot**	
DEs1	In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, muss auch eine Folgerung *F* erfüllt sein.	PRs1	In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, darf eine Folgerung *F* nicht erfüllt sein.
DEs2	Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss im nächsten Zustand, in dem rdy_plc erfüllt ist, auch eine Folgerung *F* erfüllt sein.		
DEs3	Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, muss es danach einen Zustand geben, in dem sowohl rdy_plc, die Bedingung *B* und eine Folgerung *F* erfüllt werden, bevor die auslösende Bedingung *B* wieder ungültig wird.	PRs3	In allen aufeinanderfolgenden Zuständen, in denen rdy_plc und eine Bedingung *B* erfüllt ist, darf eine Folgerung *F* nicht erfüllt sein. (äquivalent zu PRs1 - jedoch sprachlich anders)
DEs4	Sobald es einen Zustand gibt, in dem rdy_in und eine Startbedingung B_1 erfüllt werden, muss es danach einen Zustand geben, in dem rdy_plc und eine Folgerung F erfüllt werden, bevor es einen Zustand gibt, in dem rdy_in und eine Endbedingung B_2 erfüllt sind.	PRs4	Sofern es einen Zustand gibt, in dem rdy_in und eine Startbedingung B_1 erfüllt sind, darf es solange keine Zustand geben, in dem rdy_plc und eine Folgerung *F* erfüllt werden, bis es einen Zustand gibt, in dem rdy_in und eine Endbedingung B_2 erfüllt werden.
DEs5	Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss es irgendwann einen Zustand geben, in dem rdy_plc und *F* erfüllt sind.	PRs5	Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, darf es danach keinen Zustand mehr geben, in dem rdy_plc und eine Folgerung *F* erfüllt werden.

Erweiterte Forderung		Erweiterte Möglichkeit	
DEe1	In jedem Zustand, in dem rdy_plc und eine Bedingung *B* erfüllt ist, muss auch eine Folgerung *F* erfüllt sein. Zusätzlich darf in jedem Zustand, in dem zwar rdy_plc aber die Bedingung *B* nicht erfüllt ist, die Folgerung *F* nicht erfüllt sein.	POe1	In jedem Zustand, in dem zwar rdy_plc aber eine Bedingung *B* nicht erfüllt ist, darf eine Folgerung *F* nicht erfüllt sein.
DEe2	Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss im nächsten Zustand, in dem rdy_plc erfüllt ist, auch eine Folgerung *F* erfüllt sein. Zusätzlich darf die Folgerung *F* nicht erfüllt sein, wenn zuvor in dem Zustand in dem rdy_in erfüllt wurde, die Bedingung *B* nicht erfüllt war.		
DEe3	Sobald es einen Zustand gibt, in dem rdy_in und eine Bedingung *B* erfüllt werden, muss es danach einen Zustand geben, in dem sowohl rdy_plc, die Bedingung *B* und eine Folgerung *F* erfüllt werden, bevor die auslösende Bedingung *B* wieder ungültig wird. Zusätzlich darf in allen Zuständen, in denen zwar rdy_plc jedoch nicht die Bedingung *B* erfüllt wird, die Folgerung *F* auch nicht erfüllt sein.	POe3	In allen Zuständen, in denen zwar rdy_plc aber eine Bedingung *B* nicht erfüllt ist, darf eine Folgerung *F* nicht erfüllt sein. (äquivalent zu POe1 - jedoch sprachlich anders)
DEe4	Sobald es einen Zustand gibt, in dem rdy_in und eine Startbedingung B_1 erfüllt werden, muss es danach einen Zustand geben, in dem rdy_plc und eine Folgerung F erfüllt werden, bevor es einen Zustand gibt, in dem rdy_in und eine Endbedingung B_2 erfüllt sind. Zusätzlich darf es keinen Zustand mit rdy_plc und *F* geben, bevor rdy_in und B_1 erstmals gültig waren und es darf auch keinen Zustand mit rdy_plc und *F* geben darf, nachdem rdy_in und B_2 erstmals gültig waren.	POe4	Nur innerhalb eines Beobachtungsintervalls, das von einer Startbedingung B_1 und einer Endbedingung B_2 eingegrenzt wird, darf es einen Zustand geben, in dem rdy_plc und eine Folgerung *F* erfüllt sind. Das heißt anderseits, dass es keinen Zustand mit rdy_plc und *F* geben darf, bevor rdy_in und B_1 erstmals gültig waren und dass es auch keinen Zustand mit rdy_plc und *F* geben darf, nachdem rdy_in und B_2 erstmals gültig waren.
DEe5	Nach jedem Zustand, in dem rdy_in und eine Bedingung *B* erfüllt ist, muss es irgendwann einen Zustand geben, in dem rdy_plc und F erfüllt sind. Zusätzlich darf es keinen Zustand mit rdy_plc und *F* geben, bevor nicht rdy_in und eine Bedingung *B* erfüllt worden sind.	POe5	Es darf keinen Zustand mit rdy_plc und *F* geben, bevor nicht rdy_in und eine Bedingung *B* erfüllt worden sind.

Zusammenfassung der CTL-Formeln

Einfache Forderung		**Einfaches Verbot**	
DEs1	AG ((rdy_plc & *B*) -> *F*)	PRs1	AG ((rdy_plc & *B*) -> ¬*F*) AG ¬(rdy_plc & *B* & *F*)
DEs2	AG ((rdy_in & *B*) -> A[¬rdy_plc U (rdy_plc & *F*)])		
DEs3	AG($r_0 = 1 \rightarrow$ A[$r_1 = 0$ U ($F \wedge$ AF $r_1 = 1$)]) mit $r_0 := V, z$ **sat** B_0 , $r_1 := V, z$ **sat** B_1	PRs3	AG ((rdy_plc & *B*) -> ¬*F*) AG ¬(rdy_plc & *B* & *F*)
DEs4	AG(rdy_in & B_1 -> AF(*F* & rdy_plc & AF(rdy_in & B_2)))	PRs4	A[((rdy_in & B_1) -> AG ¬(rdy_plc & *F*)) U (rdy_in & B_2)]
DEs5	AG ((rdy_in & *B*) -> AF (rdy_plc & *F*))	PRs5	AG ((rdy_in & *B*) -> ¬EF (rdy_plc & *F*)) AG ((rdy_in & *B*) -> AG ¬(rdy_plc & *F*))
Erweiterte Forderung		**Erweiterte Möglichkeit**	
DEe1	AG (((rdy_plc & *B*) -> *F*) & ((rdy_plc & ¬*B*) -> ¬*F*)) AG ((rdy_plc & *B*) <-> (rdy_plc & *F*)) AG (rdy_plc -> (*B* <-> *F*))	POe1	AG ((rdy_plc & *F*) -> *B*) AG ((rdy_plc & ¬*B*) -> ¬*F*) AG ¬(rdy_plc & ¬*B* & *F*)
DEe2	AG (((rdy_in & *B*) -> A[¬rdy_plc U (rdy_plc & *F*)]) & ((rdy_in & ¬*B*) -> A[¬rdy_plc U (rdy_plc & ¬*F*)]))		
DEe3	AG(($r_0 = 1 \rightarrow$ A[$r_1 = 0$ U ($F \wedge$ AF $r_1 = 1$)]) & (A[*C* U $r_0 = 1$] $\vee$ AG *C*) & AG($r_1 = 1$ -> (A[*C* U $r_0 = 1$]) $\vee$ AG *C*))), mit $r_0 := V, z$ **sat** B_0 , $r_1 := V, z$ **sat** B_1 , *C* := rdy_plc & ¬*B* & ¬*F*	POe3	AG ((rdy_plc & *F*) -> *B*) AG ((rdy_plc & ¬*B*) -> ¬*F*) AG ¬(rdy_plc & ¬*B* & *F*)
DEe4	AG((rdy_in & B_1 -> AF(*F* & rdy_plc & AF(rdy_in & B_2))) & (A[C U B_1] $\vee$ AG *C*) & AG(B_2 -> (A[*C* U B_1] $\vee$ AG *C*))), mit *C* := rdy_plc & ¬B_1 & ¬*F*	POe4	AG (((rdy_in & B_2) -> ¬EF (rdy_plc & *F*)) & A[¬(rdy_plc & *F*) U (rdy_in & B_1)])
DEe5	AG (((rdy_in & *B*) -> AF (rdy_plc & *F*)) & A[¬(rdy_plc & *F*) U (rdy_in & *B*)])	POe5	A[¬(rdy_plc & *F*) U (rdy_in & *B*)]