FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Programmanalyse und Verifikation von SPS-Programmen

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Gernot Kucera
Matrikelnummer 6026191

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Gernot Salzer

Diese Dissertation haben begutachtet:

Em. O.Univ.-Prof. Dipl.-Ing. u. Ing. (grad.) Dr.-Ing. Gerhard-Helge Schildt

Univ.-Prof. Dr.-Ing. Prof. h.c. Stefan Böhm

Wien, 30.06.2017

Gernot Kucera

Erklärung zur Verfassung der Arbeit

Gernot Kucera
Hermesstraße 145, 1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30.06.2017 Gernot Kucera

Danksagung

Speicherprogrammierbare Steuerungen und Steuerungstechnik begleiten nahezu meinen gesamten beruflichen Werdegang. Besonderer Dank gilt meinen Betreuern, die meine Interessen für diese Arbeit geteilt und gefördert haben.

Herrn Prof. Salzer danke ich, dass er meine Arbeit durch seine Anregungen und Diskussionsbeiträge aktiv begleitet hat. Insbesondere sein kritischer Blick auf die Auswahl der von mir verwendeten Quellen hat meinen Blick geschärft und damit meine Arbeit verbessert.

Herrn Prof. Schild danke ich für seine kritischen Fragestellungen, die wesentlich zur Verbesserung und zur Verständlichkeit des Textes beigetragen haben. Auch Beispiele aus seiner langjährigen Berufserfahrung haben mir weitergeholfen, den Blick auf das Wesentliche nicht aus den Augen zu verlieren. Seine Anregungen zu Erklärungen und Detaillierungen von Sachverhalten, die ich als selbstverständlich und damit als gegeben betrachtet hatte, unterstützen die Verbreitung der Erkenntnisse in praktische Umsetzungen.

Herrn Prof. Böhm danke ich für seine Bereitschaft meine Arbeit aus der Sicht seiner Expertise zu begutachten.

Meinem Freund Herbert Paulis danke ich für sein Lektorat und den gezeigten besonderen Spürsinn, versteckte Fehler und Ungenauigkeiten in der Ausdrucksweise aufzudecken.

Meiner Familie danke ich für die mir entgegengebrachte Geduld und das Verständnis für meine Arbeit, welche neben einer beruflichen Vollauslastung nicht gezählte Wochenenden und Nachtstunden in Anspruch genommen hat.

Wien, im Juni 2017 Gernot Kucera

Kurzfassung

Speicherprogrammierbare Steuerungen (SPS) werden zur Lösung von Automatisierungsaufgaben technischer Systeme und Anlagen genutzt. Die Herausforderung an die moderne Automatisierungstechnik ist die Beherrschung der Komplexität von Prozessen. Innerhalb dieser Prozesse ist dies durch die steigende Anzahl der Signale und damit verbunden durch immer größere Datenmengen gekennzeichnet. In den dazu verwendeten Rechnersystemen nehmen SPS eine wichtige Rolle ein. Nicht nur Steuerungsaufgaben, sondern auch Betrachtungen zur Zuverlässigkeit und zur Sicherheit und die Umsetzung geeigneter Maßnahmen dazu werden immer stärker in den Mittelpunkt der Automatisierung gerückt.

Als programmgesteuerte Automatisierungscomputer werden SPS in nahezu allen Bereichen der Anlagentechnik beginnend bei chemisch technologischen Anlagen bis hin zu Maschinensteuerungen eingesetzt. SPS werden darin als sichere Rechensysteme für hoch verlässliche Steuerung und Regelung technischer Prozesse genutzt. Der ständig steigende Automatisierungsgrad neuer technischer Systeme und der Wunsch nach immer mehr Funktionalität und Flexibilität in Automatisierungseinrichtungen führen zwangsläufig zu höherer Komplexität der sie bereitstellenden Hard- und Softwaresysteme. Damit wird die Frage der Zuverlässigkeit und Sicherheit zum zentralen Problem des Entwurfs und der Realisierung neuartiger Automatisierungssysteme.

Nicht nur für in sicherheitsrelevanten Anwendungen eingesetzte SPS Systeme muss für das betrachtete System das ausfallsicherheitsgerichtete Verhalten nachgewiesen werden. Dazu sind oft zeitaufwändige Tests während der Phase der Inbetriebsetzung notwendig. Anlagefunktionen werden dabei verifiziert und validiert. Eine wesentliche Rolle spielen dabei die SPS-Programme. Weil die Leistungsfähigkeit von SPS-Systemen im Zuge der technischen Weiterentwicklung zusammen mit den Anforderungen an solche Systeme Schritt gehalten hat, sind Systeme mit mehreren Tausend Signalzuständen die Regel und nicht die Ausnahme. Das exponentielle Anwachsen der möglichen Systemzustände ist bekannt als das „state explosion problem“, die Beherrschung des Problems Gegenstand der Forschung.

Als Beitrag zur Beherrschung der Komplexität zeigt diese Arbeit, wie durch die Zerlegung der Systemfunktionalität eines umfangreichen durch eine SPS gesteuerten Systems die Gesamtaufgabe einer Verifikation automatisiert unterstützt wird, in dem durch Segmentierung SPS-Programmbereiche definiert werden, die im Sinne der Gesamtfunktion von Programmteilen überblickbar und damit handhabbar werden. Diese SPS-Programmbereiche werden einzeln einer Verifikation zugänglich gemacht. Die Zerlegung selbst erfolgt durch automatisierte Analyse des vom Anwender einer SPS erstellten Anwender-Programms.

Die Analyse von Anwender-Programmen speicherprogrammierbarer Steuerungen ist ein Teilschritt für die automatisierte Verifikation. Zweck der Analyse ist es, einerseits mögliche Probleme im Programm einer SPS aufzudecken und andererseits ein SPS-Programm einer weitgehend automatisierten Verifikation zugänglich zu machen.

In der Folge werden Programmkonstrukte einer SPS als Segmente bezeichnet, die gleichzeitig einzeln verifizierbare Einheiten sind. Frei nach dem Ausspruch „divide et impera“ kann mit Hilfe von Analysetechniken ein SPS-Programm geeignet zerlegt werden. Bezogen auf einzelne Segmente wird dadurch der Untersuchungsraum stark verkleinert. Die zulässige Größe eines Segments kann durch das Verändern seiner Grenzen an die Rechenleistung des Untersuchungssystems angepasst werden. Einzige Voraussetzung für die Verifikation ist, dass ein SPS-Programm vollständig untersucht wird. Deshalb werden bei Mehrfachverwendung von Informationen die Segmentgrenzen im Untersuchungsraum derart angepasst, dass auch Überschneidungen durch Mehrfachverwendung von Programmelementen vollständig bei der Bearbeitung berücksichtigt werden. Zweckmäßigerweise werden hierzu Minima für die Überschneidungen bestimmt.

Die Tatsache, dass bei einem SPS-Programm von „Linearität“ bei der Programmbearbeitung ausgegangen werden kann, weil speziell bei sicherheitsrelevanten Steuerungen Programmkonstrukte mit Rückwärtssprüngen oder Programmschleifen unzulässig sind, erleichtert die Aufgabe. Untersucht wird der vollständige Lösungsraum innerhalb jedes Segments. Genügen daher alle Kombinationen der als Startbedingung vorausgesetzten Eingangsabbilder (Eingangsvektoren) den Anforderungen an die Ausgangsabbilder (Ausgangsvektoren), ist die Verifikation erfolgreich. Durch Rückwärtsanalyse werden für jeden SPS-Ausgang die bestimmenden Elemente zugeordnet. Damit wird der Lösungsraum vollständig in Bezug auf die Ausgangsabbilder abgebildet.

Das zugehörige Konzept für die Verifikation folgt den Regeln klassischer Verifikationstechniken: Eingangsabbilder werden als Vorbedingungen behandelt und mathematisch untersucht, ob zugehörige Ausgangsabbilder als Ergebnis erreicht werden. Anstatt den gesamten Verifikationsschritt auf einmal durchzuführen, wird ein mehrfach gestuftes Verfahren vorgeschlagen. Grob gesehen können Schritte identifiziert werden: die Voruntersuchungen zur Syntax mit semantischen Anteilen, die Zerlegung mit Verifikation von Teilbereichen und die Untersuchung auf Vollständigkeit.

Das Ende einer Untersuchung, die erfolgreiche Verifikation, ist nicht von der Reihenfolge der Festlegung der Prämissen für die Verifikation abhängig. Im konventionellen Ansatz zur Verifikation werden zuerst die Bedingungen definiert zusammen mit zu erzielenden Ergebnissen, um in Relation mit dem Quellcode in möglichst wenigen Schritten eine Entscheidung herbei-

zuführen. Prinzipiell ist dies beim hier verfolgten Ansatz genauso. Im Unterschied dazu erlaubt jedoch die hier verfolgte Methode die Nutzung auch so genannter „offener“ Referenzen, die erst in einem weiteren Zwischenschritt zu Prämissen werden. Zuletzt werden Verifikationsergebnisse von Teilbereichen zu einer Verifikation des gesamten SPS-Programms rekombiniert.

Schlüsselwörter: Programmanalyse, Verifikation, SPS-Programm

Abstract

Programmable Logic Controllers (PLC) are used in automation of technical systems and machines. The challenge for modern automation is in the handling of the complexity of the processes. These processes are characterized by an increasing number of signals and linked to this is an increasing amount of data. PLC used in the processing systems play an important role. Not only controlling tasks but also reliability and safety issues and their realisation come more and more into the focus of automation.

PLC are used as programmable automation controllers in almost all areas of plant engineering all the way from chemical plants to production machinery controls. In these fields PLC are used as safe computing systems for safety relevant controls and feedback control of technical processes. The increasing level of automation in new technical systems and the demand for more and more functionality and flexibility in automation systems inevitably lead towards higher complexity of supporting hardware and software systems. This makes problems of reliability and safety a central problem in design and realisation of new automation systems.

Safeguarding against failure for systems in question must be verified not only for safety relevant applications. This often calls for time-consuming testing during installation of systems. System functionalities get verified and validated in the process. PLC software plays a critical role in this. As the performance of PLC systems has increased with modern technical enhancements together with the requirements for such systems, nowadays systems with several thousands of signal conditions are no exceptions any more but rather state of the art. The exponential growth of possible system states is known as "state explosion problem" and research currently focuses on dealing with this problem.

This thesis contributes to handling the complexity by showing how automated verification of a large PLC controlled system can be supported by breaking down the functionality of the system. This is achieved by segmenting PLC program components to make them straightforward and therefore manageable in the sense of the full functionality. These PLC program segments will now be put to verification part by part. The segmentation itself is done by means of automatic analysis of the user-generated PLC program.
Analysing user programs for programmable logic controllers (PLC) is a step towards automatic verification. Purpose of the analysis is on one hand to find possible problems in the PLC program and on the other hand to make the PLC program available for an in most parts automated program verification.

Subsequently, program constructs for PLC will be called segments, which will at the same time be stand-alone verifiable components. According to the saying "divide et impera" analysis techniques can be used to fittingly decompose a PLC program. Regarding single segments, the scope for examination will be made significantly smaller by this. The size of one segment can be adapted to the performance capabilities of the analysing system by reducing its size. The only requirement for verification is that the PLC program must be analysed completely. Therefore, if information is used repeatedly, segment borders will be adapted in such a way that overlapping of code parts is fully considered during processing. Overlapping minima will be defined appropriately for this purpose.

The task is made easier by the fact that linearity can be assumed with PLC programs as in safety relevant programs loops and backward jumps are not allowed. For each segment the full solution space will be analysed. So if all possible combinations of input images (input vectors) satisfy the requirements of the output images (output vectors) the verification is successful. By backward analysis every PLC output has its determining elements assigned. By this the solution space will be fully mapped with regard to the output images.

The associated verification concept for the verification clings to the common verification rules. Input images will be considered as preconditions and mathematically analysed if they lead towards corresponding output images as results. Instead of performing a complete verification in one single step, a multiple step approach is suggested. In a bigger picture, the following steps can be identified: syntactic pre-examinations with semantic parts, breaking the system down and verifying portions, and a test for completeness

An examination finish, the successful verification, does not depend on the order of defining the assumptions for verification. In the conventional approach first all conditions together with the expected results are defined to provide a result together with the source code in as little steps as possible. The approach discussed here basically works the same but the method described allows for so called open references which will become assumptions only at later steps. Finally, all verification results for the various parts will be recombined into a verification of the complete PLC program.

Key words: program analysis, verification, PLC program

Vorwort

Obwohl SPS bereits eine hohe Verbreitung haben, gibt es nur wenige Ansätze, Steuerungsprogramme zu verifizieren. Ein Grund dafür ist das als „*state explosion problem*“ bekannte Phänomen, dass mit jeder neu eingeführten binären Variablen sich die Anzahl der Untersuchungen verdoppelt.

Die bekannten Arbeiten nutzen Methoden, die entweder sehr komplex und damit aufwendig umzusetzen sind oder man beschränkt die Funktionalität und damit die Möglichkeiten zur Verifikation so stark, dass sie in der Praxis nur geringen Nutzen haben. Die Durchführung einer Verifikation ist ein sehr aufwendiger Vorgang. In dieser Arbeit wird eine Vorgehensweise vorgeschlagen, die weitgehend automatisiert ablaufen und darüber hinaus parallel zur Erstellung des SPS-Programms erfolgen kann.

Auch hier wurden Einschränkungen in der Funktionalität durch Beschränkung der möglichen SPS-Befehle auf die in Steuerungen am häufigsten verwendeten gemacht. Auf Grund des modularen Aufbaus der einzelnen Programme sind weitere Funktionen jedoch vergleichsweise einfach nachrüstbar.

Die vorliegende Arbeit bereitet SPS-Programme auf, um diese mit dem in „Schritte zur Verifikation von SPS-Programmen – Ist Prädikatenlogik eine Lösung?“ [KP12] vorgestellen Verifikator der Verifikation zugänglich zu machen. In [KP12] wurde ein in „*first order logic*“ (FOL) geschriebenes Programm als so genannter Verifikator eingeführt und für die Verifikation von SPS-Programmen genutzt. Die ersten Ideen zu diesem Buch stammen aus zwei Diplomarbeiten des Autors. Die Arbeiten „Entwicklung einer speicherprogrammierbaren Steuerung“ (2000) und „Ertüchtigung einer Mikrocontroller Plattform zum Einsatz in sicherheitsgerichteten Anwendungen“ (2001) führten zu Mikrocontrollersystemen, die mit einem Befehlsinterpreter das Sprachkonzept Anweisungslisten für die Anwenderprogramme genutzt hat.

Als Co-Autor von [KP12] hat Herbert Paulis das Verifikationsmodell beruhend auf dem von Loeckx und Sieber in „The Foundations of Program Verification“ entwickelten formalen Modell beschrieben [KP12, Kapitel 6]. Auch die Die PROLOG-Verifikationsmaschine und die Hilfsprogramme [KP12, Kapitel 7, Anhang] stammen im Wesentlichen von Herbert Paulis, wobei Tests der Programme zur Überprüfung der Funktionalität gemeinsam durchgeführt worden sind.

Wien, im Juni 2017 Gernot Kucera

Inhaltsverzeichnis

Abbildungen

Tabellen

Abkürzungen

Abkürzung	Anmerkung
APLC	Abstract Programmable Logical Controller, Abstraktion einer von Susta [RS02] vorgeschlagenen SPS
AWL	Anweisungsliste, Sprachkonzept für SPS nach IEC 61131 Teil 3
BBD	Binary Decision Diagram
CTL	Computation Tree Logic
FD	Function Diagram, Sprachkonzept für SPS nach IEC 61131 Teil 3
FOL	First Order Logic
FUP	Funktionsplan, Sprachkonzept für SPS nach IEC 61131 Teil 3
IL	Instruction List, Sprachkonzept für SPS nach IEC 61131 Teil 3
KOP	Kontaktplan, Sprachkonzept für SPS nach IEC 61131 Teil 3
LTL	Linear Temporal Logic
NuSMV	A new Sybolic Model Checker
OBBD	Ordered Binary Decision Diagram
PLC	Programmable Logical Controller
RLL	Relay Ladder Logic, Sprachkonzept für SPS nach IEC 61131 Teil 3
SMV	Symbolic Model Verifier
SPS	Speicherprogrammierbare Steuerung

1 Einführung

Die Verifikation von Programmen allgemein und die Verifikation von Steuerungsprogrammen sind aufwendig und erfordert umfangreiche Kenntnisse, so dass dafür in der Regel nur Experten mit derartigen Aufgabenstellungen betraut werden können. In dieser Arbeit werden Möglichkeiten untersucht, die bereits dem Programmierer von speicherprogrammierbaren Steuerungen (SPS) Werkzeuge zur Verfügung stellen, mit dem SPS-Programme automatisiert überprüft und verifiziert werden können.

1.1 Motivation

Speicherprogrammierbare Steuerungen und formallogische Verifikation sind Bereichen der Informationstechnologie zuzuordnen, die in der Regel nur wenig miteinander zu tun haben. Je kleiner und leistungsfähiger Steuerungssysteme werden, etwas, das heutzutage schon fast alltäglich passiert, umso stärker durchdringen sie alle Bereiche der modernen Fertigungstechnologie. Nahezu alle Aufgabenstellungen beinhalten auch sicherheitskritische Elemente und damit wächst der Bedarf an verifizierter und im Weiteren auch an validierter Software für derartige Systeme, die bereits vor ihrem Einsatz auf ihre Korrektheit überprüft wird.

Bis dato hat sich die Verifikation von Steuerungsprogrammen immer als extrem zeit- und kostenaufwändige Angelegenheit gezeigt und daher wurde sie auch nur sehr sparsam angewendet, wenn es sich aus technischen oder rechtlichen Gründen absolut nicht vermeiden ließ. Ebenso hat sich in der Informatik die formale Verifikation ausschließlich auf höhere Programmiersprachen konzentriert. Vielfach wurden die vergleichsweise einfacheren Strukturen von Steuerungssprachen für eine Analyse als zu simpel empfunden. Stellt man sich dieser Herausforderung trifft man dabei auf eine Vielzahl von alles andere als trivialen Problemen. Ziel ist es, einen theoretischen Oberbau zu entwickeln und zusätzlich die Verwendbarkeit in der Praxis zu ermöglichen.

1.2 Definition des Problems

Allgemein gilt, dass wenn ein Computerprogramm - im speziellen ein SPS-Programm - einer Spezifikation genügt, dieses als korrekt bezeichnet wird. Allerdings deckt dabei der Begriff Korrektheit nicht ab, ob die Spezifikation die vom Programm zu lösende Aufgabe in allen Einzelheiten vollständig beschreibt.

Die grundlegende Technik für die Verifikation geht auf Hoare zurück. Hoare führte in seinem Aufsatz „An axiomatic basis for computer programming“ [HO69] aus, dass ein Programm-

code bezüglich einer Vorbedingung $\mathcal{P}$ und der Nachbedingung $\mathcal{Q}$ dann partiell korrekt genannt wird, wenn bei einer Eingabe die erfüllende Vorbedingung $\mathcal{P}$ jedes Ergebnis der Berechnung auch die Nachbedingung $\mathcal{Q}$ erfüllt. Dabei können Programmeingaben existieren, bei denen das Programm kein Ergebnis liefert. In diesem Fall heißt das, dass das Programm nicht terminiert. Ein Programmcode wird total korrekt genannt, wenn er partiell korrekt ist und zusätzlich für jede Eingabe terminiert, die auch die Vorbedingung $\mathcal{P}$ erfüllt.

Im Wesentlichen fokussiert diese Arbeit Wirkungsabläufe in Folgesteuerungen. Solche Abläufe werden in der Steuerungstechnik von Anlagen, Maschinen bzw. Fertigungseinrichtungen als Kommandofolgen in SPS-Systemen realisiert.[1] Als Aufgabenstellung wird in dieser Arbeit die Verhaltensanalyse von gesteuerten Einheiten, wie diese z.B. in Fertigungseinrichtungen anzutreffen sind, untersucht und der Verifikation zugänglich gemacht. Das Merkmal dieses Verhaltens ist, dass während des Ablaufs des Prozesses von der SPS Stellsignale ausgegeben werden, die als Schaltsignale wie Sprungfunktionen wirken. Dieses Verhalten ist typisch für die Ausgänge. Sich daraus ergebende Sprungantworten erfolgen entsprechend des jeweiligen Zeitverhaltens in der Regel zeitlich versetzt und sind von Ereignissen abhängig. Solche Ereignisse sind z.B. das Erreichen eines bestimmten Zustands, der seinerseits Folgezustände initialisieren kann oder Eingriffe im Rahmen der Bedienung. Signalzustände werden als Eingänge in der SPS verwaltet.

Das Hauptproblem der Verifikation allgemein liegt in der Anzahl von zu berücksichtigen Zuständen, bekannt als das „state explosion problem“. Auch in SPS-Steuerungsprogrammen von automatisierten Fertigungseinrichtungen wächst die Anzahl der zu überwachenden Funktionen und damit die damit verbundenen Zustände stark an. Einhundert Variable bedeutet, dass 2^{100} oder $1{,}27 \cdot 10^{30}$ unterschiedliche Zustandskombinationen untersucht werden müssen, um sichere Aussagen zur Korrektheit eines SPS-Logikprogramms zu machen. Dabei werden bereits für mittlere Steuerungsaufgaben einige hundert verschiedene Signale zur Steuerung und Überwachung einer Fertigungseinrichtung notwendig. SPS-Programme können Verifikationsmethoden wie dem Model-Checking in der Regel nicht zugänglich gemacht werden, weil anstelle formaler Programmspezifikationen für die Aufgabenstellung nur informelle Beschreibungen üblich sind.

Nicht zu unterschätzen ist der Umstand, dass SPS-Programmierer in den seltensten Fällen Informatiker sind. Typische SPS-Programmierer sind die Konstrukteure von Fertigungsmaschinen, Techniker, die sich auf Steuerungsprogrammierung spezialisiert haben oder Automa-

[1] In der DIN-IEC 60050-351[1] ist zum Begriff Steuern wenig hilfreich definiert, dass das „Kennzeichen für das Steuern […] der offene Wirkungsweg oder ein geschlossener Wirkungsweg [ist], bei dem die durch Eingangsgrößen beeinflussten Ausgangsgrößen nicht fortlaufend und nicht wieder über dieselben Eingangsgrößen auf sich selbst wirken“.

tisierungstechniker. Vielfach ist das Servicepersonal programmierend tätig, das Eingriffe in die Programme vornehmen muss um den Betrieb aufrecht zu erhalten. Dieser Personenkreis besitzt nur eine geringe Bereitschaft, Methoden der Verifikation als Werkzeug zu nutzen. Grund dafür ist unter anderen, dass ab einer Anzahl von verschiedenen Zuständen oder Zustandskombinationen die Überblickbarkeit nicht mehr gegeben ist. Darüber hinaus sind die erforderlichen Kenntnisse der Zielgruppe SPS-Programmierer üblicherweise nicht groß genug, um die Verifikation anwenden zu können. Das Spezialwissen der typischen SPS-Programmierer reicht für die Programmerstellung gut aus, für das Verstehen der Konzepte für die Verifikation jedoch nicht.

Wird eine Verifikation notwendig, muss sich der mit Verifikation von SPS-Programmen betraute Personenkreis mit der Aufgabenstellung intensiv beschäftigen, um den Gedankengängen der Programmierer und deren Umsetzung in ein SPS-Programm zu folgen und, um auf Grund ihrer Untersuchung zur Korrektheit der Programme, diese gegebenenfalls zu verifizieren. Demgegenüber haben SPS-Programmierer den unmittelbaren Vorteil, dass sie ihre Aufgabenstellungen und deren Besonderheiten bereits sehr gut kennen. Sie mussten sich intensiv mit den für das Programm festgelegten Vorgaben bis ins Detail beschäftigen und haben damit bei korrekten SPS-Programmen alle Aspekte eines Steuerungsablaufs behandelt. Aus diesem Grund scheint es naheliegend zu sein, den SPS-Programmieren ein Werkzeug bereitzustellen, mit dem sie bereits im Zuge der Programmierung SPS-Programme verifizieren können.[2] Als wichtige Eigenschaft solcher Werkzeuge sind für deren Akzeptanz notwendig, dass Spezialkenntnisse nur in geringem Umfang erforderlich werden. Idealer Weise sollten die Kenntnisse zur SPS-Programmierung und der Wirkungsweise von SPS-Systemen ausreichen.

Bei technischen Systemen wird Sicherheit[3] als Zustand einer störungsfreien und gefahrenfreien Funktion verstanden. Dabei ist der Begriff „Sicherheit“ von seiner Definition abhängig bzw. welcher Grad von verbleibender Restunsicherheit für die Nutzung von technischen Funktionen akzeptiert wird. Die Norm IEC 61508 [1] definierte Sicherheit als „Freiheit von unvertretbaren Risiken“ und beschreibt funktionale Sicherheit als Teilaspekt eines technischen Systems.

---

[2] Bei der Programmerstellung werden die Übergänge von einem Status der Maschine in den folgenden Status definiert. Diese Definition basiert auf Zuständen von Ein- bzw. Ausgängen z.B. eine Ausgangsstellung der Maschine und der Umsetzung durch das Programm, was als nächstes auf Grund einer Aufgabenstellung passieren soll. Aus einer folgerichtigen Programmierung muss sich daher die Folgestellung der Maschine ergeben, wodurch die Zielzustände implizit gegeben sind. In „Schritte zur Verifikation von SPS-Programmen“ [KP12] wird ein Ansatz diskutiert, wo das Prinzip einer reaktionsschnellen Steuerung skizziert ist, deren Programm ausschließlich aus Übergangsfunktionen gebildet wird.

[3] Synonym wird der Begriff „Betriebssicherheit“ in diesem Zusammenhang genutzt.

Eine besondere Bedeutung für Sicherheit von technischen Systemen liegt bei der genutzten Software. Zur Entwicklung wird ein hoher Aufwand für die Sicherstellung der Fehlerarmut der Software betrieben. Dieser Aufwand ist umso mehr gerechtfertigt, wenn sicherheitskritische Prozesse überprüft werden, die in technischen Systemen ablaufen.

Stark verkürzt liegen die grundlegenden Probleme in der wachsenden Komplexität von SPS-Programmen, die auf die Steigerung von Möglichkeiten für SPS-Anwendungen zurückzuführen ist und der fehlende Zugang für typische SPS-Progammierer zu Verifikationstechniken.

Kompexität von SPS Programmen

SPS wurden ursprünglich als Ersatz für herkömmliche Steuerungssysteme konzipiert und eingeführt. Eines der Hauptziele war auch, die Konzeption und Programmierung von SPS derart einfach zu gestallten, um den für den Betrieb solcher Systeme zuständigen Personenkreis einen einfachen Zugang zu bieten. Dieses Ziel wurde so gut erreicht, so dass für Programmierung oder Wartung von Programmen bzw. für Inbetriebnahme von Steuerungen und Anlagen vielfach Praktiker eingesetzt werden können, die mit vergleichsweise geringen theoretischen Kenntnissen auskommen.

Die technische Weiterentwicklung der SPS-Systeme ist bereits derart fortgeschritten, dass solche Systeme in hochkomplexen Anwendungen einsetzbar sind, die früher Rechnersystemen vorbehalten waren. SPS-Systeme werden auch zur Kontrolle sicherheitskritischer Aufgabenstellungen genutzt. Auch aus diesem Grund rückt die Verifikation von Steuerungssystemen mehr und mehr in den Mittelpunkt des Interesses.

Zugang von SPS-Programmieren zu Verifikationtechniken

Ganz allgemein gesehen erfordert die Anwendung von Methoden für die Verifikation von Programmen besondere Kenntnisse, die auf Grund von Herkunft und Vorbildung der Programmierer von SPS-Systemen nur in Ausnahmefällen gegeben sind.

Das Hauptproblem liegt in der Diskrepanz der Denkweise von theoretischen Informatikern im Gegensatz zu Praktikern, die eine Steuerung oder Anlage und deren Erfordernisse aus der Sichtweise der Anwendung so gut kennen, dass vertiefende Kenntnisse zu den zugrundeliegenden Theorien für die Programmumsetzung nicht benötigt werden. Im Gegenzug ist die Ausdruckweise bzw. die Beschreibungssprache der Praktiker üblicherweise nicht formal genug, dass die theoretische Informatik damit etwas anfangen kann.

Formale Spezifikationen für SPS-Programme sind sehr selten anzutreffen. Beschreibungen von SPS-Programmen sind üblicherweise verbal und zum Teil nicht vollständigen Beschreibungen zugrunde gelegt, die erst durch ergänzende Informationen in Steuerungsprogramme umgesetzt werden können. Typisch ist auch, dass Informationsketten gebildet werden, beginnend bei der Konzeption durch den Maschinenbau, Transfer der Information an die Steuerungstechnik mit Abgleich zu technischen oder technologischen Gegebenheiten, die in der Folge zu Variationen in der Programmentwicklung führen. Die SPS-Programmierung wird dadurch zu einem iterativen Prozess mit Anpassungen und Veränderungen, der erst bei der Abnahme einer Maschine oder Anlage als abgeschlossen betrachtet werden kann.

SPS-Programme folgen den Regeln ihrer Programmiersprachen, sind also per se formal. Sie dienen in dieser Arbeit als das Bindeglied zwischen einer praktischen Anwendung und ihrer formalen Verifikation. Die Programmierer, die eine Aufgabenstellung umsetzen, können jedenfalls Ausgangssignalzustände beschreiben und die zugehörigen Zielzustände definieren. Mit Hilfe solcher Zuordnungen soll die Überprüfung der erstellten SPS-Programme in der Form einer Verifikation durchgeführt werden. Anders ausgedrückt: die Zielvorstellung ist, eine Methode zu definieren, mit der SPS-Programmierer ohne besondere Zusatzkenntnisse ihre Programme selbst bereits im Zuge der Programmerstellung und bei den allfälligen Programmanpassungen wiederholt automatisiert verifizieren können.

Erschwerend ist dabei die Komplexität, weil SPS-Programme auch auf Grund der immer größer werdenden Leistungsfähigkeit solcher Systeme eine große Anzahl von Zuständen einnehmen können. Das exponentielle Anwachsen von zu betrachtenden Zuständen, die Verdoppelung der Anzahl von Zuständen für jeden weiteren, ist als das state explosion problem bekannt und bildet Schranken für die Übersichtlichkeit und in Folge für die Verifikation.

1.2.1 Funktionalität von SPS-Programmen

Als Beispiel wird die typische Vorgehensweise einer Funktionsüberprüfung von SPS-Programmen herangezogen. Historisch gesehen waren die ersten SPS-Programme auf Grund der noch geringen Leistungsfähigikeit dieser Geräte vergleichsweise einfach, sowohl was die Programmierung als auch die Inbetriebsetzung betroffen hat. Die typische Überprüfung auf Korrektheit einer Anlage erfolgte durch schrittweises Testen der programmierten Funktionen. Eines der Hauptargumente für die Einführung von SPS war, dass einerseits auf Grund der Einfachheit der Verdrahtung und andererseits der zugehörigen Programme solche Steuerungssysteme mit geringem Aufwand in Betrieb genommen werden konnten.

Dem Grunde nach hat sich diese Argumentation bis heute nicht verändert. Auch die Vorgehensweise für Inbetriebsetzung ist im Wesentlichen gleichgeblieben: ausgehend von Ein-

gangsbedingungen - den Zuständen einer Maschine oder Fertigungseinrichtung - resultieren bei korrekter Funktion Ausgangsbedingungen. Die Vorgehensweise beim schrittweisen Testen ist vergleichbar mit der Verifikationstechnik von Hoare.

Dieses Schema wird auch heute noch angewendet. Als problematisch gilt dabei nur die gesteigerte Komplexität der Aufgabenstellungen. Umgangen wird das in der Praxis dadurch, dass geeignete Teilbereiche gesucht und diese dann entsprechend behandelt werden.

Untersuchungen und Ziele

In dieser Arbeit wird untersucht und gezeigt, dass durch geeignete Segmentierung von SPS-Programmen auch komplexere SPS-Programme in Teilbereiche derart zerlegt werden können, dass damit bereits bei der Erstellung solcher Programme Verifikationsschritte durchführbar sind. Im Gegensatz zur realen Inbetriebnahme muss in diesen Arbeitsschritten die zu überprüfende Anlage oder Fertigungseinrichtung noch nicht zur Verfügung stehen.

Gezeigt wird auch, dass die Untersuchung vollständig durchgeführt werden kann, dass also der Untersuchungsraum der Zustände vollständig abbildbar ist. Die für die Verifikation definierten Teilbereiche sind zusammensetzbar. Gefolgt wird bei der Prüfung von Teilbereichen dem bei der Inbetriebsetzung von Maschinen oder Anlagen üblichen Prinzipien. Zu jedem Segment des zu untersuchenden Programms werden Zustandskombinationen gebildet, die den Testszenarien einer typischen Inbetriebnahme entsprechen. Im Unterschied zur realen Inbetriebnahme können mit der vorgeschlagenen Methode aber auch Zustandskombinationen überprüft werden, die bei einer fertig gestellten Maschine oder Anlage z.B. wegen einer speziellen mechanischen Anordnung von Maschinenteilen nie erreichbar sind.

1.2.2 Spezifikation

Vielfach erfolgt die Konzeption eines SPS-Programms nach dem Schema, dass auf Grund einer informellen Spezifikation zu einer Aufgabenstellung das Programm direkt implementiert wird.

Nach der Erstellung des Programms wird in der Testphase die Funktionalität mit der informellen Spezifikation in Bezug gebracht und so die Abläufe validiert. Validieren bedeutet in diesem Zusammenhang, dass nicht nur die vorgegebene Funktionalität durch das Programm erfüllt wird, sondern auch die gesamte Anlage einschließlich der an die SPS angeschlossenen Sensoren und Aktuatoren mit einbezogen werden.

Eine weitere Möglichkeit besteht darin, dass ausgehend von der informellen Spezifikation durch Formalisieren eine formale Spezifikation gebildet wird. Diese kann verifiziert werden. Das grundsätzliche Ziel dabei ist, Ein- und Ausgangzustände der SPS in Relation mit dem SPS-Programm zu untersuchen. Zusätzlich kann gefordert werden, dass als Ergebnis dieser Untersuchung ausschließlich Zustandsübergänge erfolgen, die im Sinne der Aufgabenstellung als erwünscht zu bewerten sind. Existieren nur erwünschte Ergebnisse und umfasst die Verifikation alle Zustandsübergänge, kann von einer Vorstufe zur Validation gesprochen werden. Während beim Validieren das gesamte System betrachtet werden muss, beschränkt sich die Verifikation auf das SPS-Programm.

Typisch für informelle Spezifikationen sind „wenn – dann“ Zusammenhänge. Derartige Zusammenhänge beschreiben gewünschte Abläufe sowie Ereignisse, die verhindert werden müssen. Während die gewünschten Abläufe in der Regel vollständig dargestellt werden, ist die Vollständigkeit in einer informellen Spezifikation für mögliches Fehlverhalten nicht immer der Fall. In Testfällen wird daher bei der Inbetriebsetzung das SPS-Programm auf Fehlerfreiheit im Sinne der Funktionalität überprüft. Nachteilig bei einer Inbetriebsetzung ist, dass eine 100%ige Testabdeckung in der Regel nicht erfolgen kann, insbesondere auch deshalb, weil nicht alle Zustände erreichbar sind.

Ein SPS-Programm besteht grundsätzlich aus Vorgaben, die für einen bestimmten Ablauf definiert werden. Derartige Vorgaben können als Algorithmen einer Steuerungsaufgabe oder einer Teilfunktion daraus gesehen werden. Ein rein formales Modell ist die Summe aller Algorithmen, die eine Aufgabenstellung vollständig beschreiben.

Modellierung

Das in dieser Arbeit verfolgte Konzept formalisiert bestehende SPS-Programme bzw. Teile davon. Dazu wird unterstellt, dass ein zu untersuchendes SPS-Programm bereits in einer Form vorliegt, die auch einer rein formalen Spezifikation genügt, also, dass darin nicht nur die funktionalen, sondern auch nicht-funktionale Anforderungen an ein System bereits programmtechnisch berücksichtigt worden sind. Damit kann ein Modell des SPS-Programms gebildet werden.

Die Modellbildung erfolgt durch das Aufstellen einer Wissensbasis, die auf Fakten, Regeln und Formeln beruht. Aufbauend auf der Wissensbasis, ist das Modell die Kette aller Instruktionen in geordneter Reihenfolge, die durch die Programmierung vorgegeben ist. Ein Weg zur Formalisierung wurde in „Schritte zur Verifikation von SPS-Programmen“ [KP12] vorgezeichnet. Theoretisch sind derartige Modelle in ihrer Größe nicht beschränkt. Der typische Verifikationsschritt ist die Überprüfung, ob ein untersuchtes Programm alle Übergänge von

Ein- zu Ausgangsbedingungen erfüllt. Notwendig werdende Begrenzungen sind aus diesem Grund nur auf das exponentielle Anwachsen mit steigender Anzahl der zu betrachtenden Zustände in Bezug auf die zu untersuchenden Testfälle zurückzuführen. Um diese Grenzen zu überwinden wird das SPS-Programm in Teilbereiche segmentiert.

1.2.3 Verfikationstechnik, Begriffsabgrenzung und Verifikationsmethoden

Model-Checking ganz allgemein gesehen wurde als formales Verfahren zur automatisierten Verifizierung nebenläufiger Systeme mit einem endlichen Zustandsraum eingeführt.[4] Mit diesem Verfahren können große Zustandsräume automatisch untersucht werden. Dazu wird ein System modelliert und dieses Modell auf seine Systemspezifikation überprüft, wobei in erster Linie die Sicherheits- und Lebendigkeitseigenschaften untersucht werden. In diesem Zusammenhang zielt die Untersuchung von Sicherheitseigenschaften auf Zustände ab, die im Modell nicht auftreten dürfen, die Lebendigkeitseigenschaften darauf, dass irgendwann ein erwünschter Zustand eintritt.

Dazu wird das zu verifizierende System in Form eines Zustandübergangsgraphen bzw. Berechnungsbaums auf Fehler untersucht. Jeder Befehl eines Prozesses bewirkt eine Veränderung des Zustandsraums. Unterschieden werden dabei das on-the-fly Model-Checking und das symbolische Model-Checking. Beim Verfahren on-the-fly Model-Checking wird der komplette Zustandsgraph mit allen Übergängen aufgebaut und jeder Zustand untersucht. Zur Überprüfung eines Modells müssen alle relevanten Zustände eines Systems gespeichert werden. Es werden entsprechende Datenstrukturen benötigt, um Zustände zu speichern. Beim symbolische Model-Checking werden Zustandsübergangsgraphen nicht explizit aufgebaut, sondern der Zustandsraum mit Entscheidungsdiagrammen (Binary Decision Diagrams, BBD) abgebildet, mit denen die Übergänge zwischen den Zuständen berechnet werden (vgl. [Cl et al.09], [Bc et.al.08]).

Mathematische Modelle von Programmabläufen, insbesondere jene für Zustandsübergänge, werden als Automaten bezeichnet. Dabei wird z.B. beim so genannten ω-Automat[5] ausgehend

---

[4] Als Zustandsraum wird hier die Menge jener Zustände verstanden, die Modelle beeinflussen. Unterscheidungen werden in Laufe der Arbeit zusätzlich getroffen. Bezogen auf das SPS-Programm existieren Eingangszustände oder initiale Zustände. Diese treten am Anfang eines SPS Zyklus in Erscheinung. Während der zyklischen Programmbearbeitung existieren temporäre Zwischenzustände, die Ausgangszustände am Ende eines Zyklus stellen Berechnungsergebnisse dar. In der Sprechweise werden später auch die Begriffe Eingangsvektor bzw. Eingangskonfiguration (von Zuständen) eingeführt, Konfiguration (von Zuständen) für temporäre Zwischenergebnisse und Ausgangsvektoren bzw. Ausgangskonfigurationen (von Zuständen).

[5] Eine spezielle Form des ω-Automaten ist der nach dem Schweizer Mathematiker Julius Richard Büchi benannte Büchi-Automat. Dieser Automatentyp ist eine spezielle Form des ω-Automaten.

von einem besonderen Zustand, der als Startzustand zu bezeichnen ist, eine Folge von Symbolen gelesen. Das Eingabewort sind Elemente des so genannten Eingabealphabets. Der ω-Automat geht schrittweise vor, wobei in jedem Schritt das jeweils nächste Symbol des Eingabewortes gelesen wird. Der Folgezustand wird durch die Zustandsübergangsfunktion in Abhängigkeit vom aktuell gelesenen Zeichen und dem momentanen Zustand bestimmt.[6]

Alle Model-Checker gehen bei der Softwareverifikation ähnlich vor.[7] Einerseits nutzen diese das zu verifizierende System als Modell, das die einzelnen Zustandsübergänge beschreibt, andererseits die Spezifikation, dargestellt z.B. als Lineare Temporale Logik (LTL). Modell und Spezifikation werden jeweils mathematisch formuliert und in Automaten umgewandelt. Die Schnittmenge dieser beiden Automaten erzeugt ein weiteres mathematisches Modell in Form eines resultierenden Automaten, der entweder der Spezifikation entspricht oder nicht. Im Wesentlichen liefert das Ergebnis der Überprüfung eine Aussage darüber, ob Übereinstimmung herrscht und damit die Korrektheit festgestellt worden ist oder nicht.

Im Grunde bedeutet das, dass ein Model-Checker zwei voneinander unabhängige Modelle benötigt. Das Erste dieser Modelle ist das geeignet formalisierte Programm, das Ist-System oder der Prüfling, das andere Modell muss aus der Spezifikation der Aufgabenstellung gewonnen werden. Wird eines dieser Modelle invertiert und die Modelle miteinander verglichen, müssen sich alle Ausgangszustände aufheben, wenn kein Fehler beim Prüfling vorliegt. Wenn das nicht der Fall ist, wird der Unterschied aufgedeckt. Ein allfälliger Fehler wird lokalisiert und kann behoben werden.

Neben den physikalischen Grenzen der für die Berechnung eingesetzten Computer liegt eine Hauptschwierigkeit bei der Erstellung korrekter Modelle und Formeln. Insbesondere das Umwandeln von LTL-Formeln in Automaten ist umfangreich und komplex.[8]

Aus der mathematischen Sichtweise wird die Problemstellung „Verifikation“ insoweit bereits als gelöst betrachtet, weil die dazu benötigten Modelle ausreichend beschrieben werden können. Damit wird die Problematik auf die programmtechnische Umsetzung reduziert.

Problematisch kann die Bildung des Modells aus der Spezifikation werden, insbesondere dann, wenn diese nur als informelle Spezifikation zur Verfügung steht. Aus der Sichtweise

---

6 Anmerkung: Obwohl bei einem SPS-Programm ein quasistationärer Zwischenzustand erreicht wird, ist die Befehlsfolge nicht als endlicher Automat modellierbar, da sich im Folgezyklus die Bearbeitung immer wieder von neuem gestartet wird.

7 Die Vorgehensweise orientiert sich am Model-Checker SPIN

8 Der Model-Checker SPIN nutzt z.B. Algorithmen und Komprimierungsverfahren, um Softwaresysteme mit großen Zustandsräumen der Verifikation zugänglich zu machen.

der Informatik müssen die entwickelten Modelle geeignet programmtechnisch verarbeitet werden. Hauptprobleme sind bei der Verarbeitung die Mächtigkeit des Zustandsraums und die benötigte Leistungsfähigkeit der dazu genutzten Rechensysteme.

1.2.4 Formalisierung einer Spezifikation

Nur in seltenen Fällen werden die Aufgabenstellungen für SPS-Programme formal spezifiziert. Ein Grund dafür könnte sein, dass die Zielgruppe der SPS-Programmierer in der Regel mit dieser Art der Darstellung zu wenig vertraut ist oder dass der Aufwand für das Erstellen formaler Spezifikationen zu hoch ist. Anstelle von Zustandübergangsgraphen herrschen im Sprachgebrauch zwischen Maschinenbauern und den Programmierern zur Definition der Aufgabenstellungen meist stark verkürzte Beschreibungen vor. Dazu werden die geplanten Übergänge des Soll-Systems als Transfer von den Eingangs- auf die Ausgangszustände sequenziell beschrieben.[9]

Grundsätzlich muss auch eine informelle Spezifikation wenigstens die auf Eingangzustände zu erwartenden Ausgangszustände implizieren, um auf diese Art das „erwünschte“ Systemverhalten sicher zu stellen. Ob ein erreichter Zustand erwünscht ist muss aus der Spezifikation folgen. Ob dieser Zustand aus der Programmlogik des SPS-Programms folgt, kann durch die Verifikation bewiesen werden. Eine informelle Spezifikation ist in der Regel unvollständig, wenn z.B. nicht alle möglichen Eingangszustände in die Beschreibung der Aufgabenstellung mit einbezogen worden sind. In solchen Fällen ist die formale Verifikation nicht möglich.

Es ist jedoch möglich, Kombinationen von Eingangszuständen auszuwählen, um damit entsprechende Ausgangszustände zu erzeugen, um in Folge damit auch die gewünschte Funktionalität zu überprüfen. Diese können in einem weiteren Schritt mit der Spezifikation in Relation gebracht werden. Erst bei Berücksichtigung aller Eingangszustände und daraus resultierender Ausgangszustände entsteht eine vollständige Beschreibung des Programmverhaltens und damit eine formalisierte Spezifikation.

---

[9] Beispiel: Grundstellung: Zylinder 1 hinten, Zylinder 2 hinten, Klemmung offen. Schritt 1: Zylinder 1 vor bis Endlage, Schritt 2: wenn Endlage Z1 erreicht dann Zylinder 2 vor. Schritt 3: wenn Position „Kemmung schließen“ passiert wird dann Ventil für die Klemmung einschalten, Schritt 4: wenn Zylinder 2 vorne (Endlage erreicht) dann Zylinder 1 zurückfahren, Schritt 5: usw.

Um eine derartige Beschreibung in ein SPS-Programm umzusetzen sind nicht nur Kenntnisse zur Programmierung notwendig, sondern es müssen auch Grundkenntnisse zur Mechanik und zur Anordnung der Signalgeber (z.B. der Endlagenüberwachungen für die Zylinder) vorhanden sein, die oft erst implizit aus Zeichnungen entnommen werden müssen.

1.2.5 SPS-Programmiersprachen und Eigenschaften

Um SPS-Programme der Verifikation zugänglich zu machen, werden in diesem Abschnitt jene Eigenschaften und Darstellungsarten von SPS-Programmiersprachen betrachtet, die für die Formalisierung nutzbar gemacht werden. In der Literatur zur Verifikation von SPS-Programmen sind bevorzugt untersuchte Sprachkonzepte für die SPS-Programmierung der Funktionsplan und die Anweisungsliste, seltener auch der Kontaktplan.

Vergleichsweise ist sicher, dass die Anweisungsliste auf Grund der inhärent gegebenen Reihenfolge in der Programmbearbeitung für die Verifikation gut nutzbar gemacht werden kann. In einer Reihung würde danach der Funktionsplan als SPS-Sprachkonzept folgen. Obwohl der Kontaktplan insbesondere bei Steuerungstechnikern die erste Wahl ist, muss bei einer Formalisierung auch die Art der Programmbearbeitung berücksichtigt werden (Stichwort horizontal mit Verzweigungen oder vertikal ohne Verzweigungen). Interessanter Weise: es gab eine Art der Programmierung, die sich an Zustandsübergangsgraphen orientierte, die jedoch vermutlich wegen mangelnder Akzeptanz wieder „verschwunden“ ist. Wenn Model-Checking sich bei SPS-Programmierung erfolgreich behaupten soll, ist der Knackpunkt der Transfer einer Spezifikation für die Erstellung eines SPS-Programms in ein Modell. Eine formale Spezifikation bei Aufgabenstellungen für SPS-Programme lässt sich aus mehreren Gründen eher nur schwer durchsetzen. Ein Grund dafür ist, dass wenn es eine formale Spezifikation gibt, daraus auch automatisiert ein SPS-Programm erzeugt werden könnte. In so einem Fall muss eine Verifikation im Vorhinein erfolgreich sein und wird damit überflüssig.

Die Berechenbarkeit kann auch als Funktion benötigter Berechnungsschritte aufgefasst werden, wenn das erwünschte Ergebnis in Abhängigkeit zur Berechnungszeit gesetzt wird. Insbesondere bei exponentiell wachsenden Zustandsräumen kann der Aufwand zu groß werden, um die benötigten Festlegungen und Definitionen für die Verifikation eines SPS-Programms zu bestimmen.

Wird der Zustandsraum beschränkt, bleibt eine gewisse Unsicherheit, ob tatsächlich die wesentlichen zu berechnenden Zustandsübergänge erfasst worden sind oder nicht. Ein weiterer Schwachpunkt liegt in der informellen Spezifikation, der üblicherweise die SPS-Programme zu Grunde liegen.

1.2.6 Lösungsansätze verwandter Arbeiten

Die Lösungsansätze verwandter Arbeiten sind deshalb von Interesse, weil in der aktuellen Forschung Model-Cheking einen vergleichsweise breiten Raum einnimmt. Dabei wird unterstellt, dass formale Spezifikationen für SPS-Programme gegeben sind. In diesem Abschintt werden die Unterschiede dieser Ansätze zur vorgeschlagenen Methodik untersucht und ver-

deutlicht. Verwandte Arbeiten konzentrieren sich auf SPS als reaktive Systeme. SPS-Programme sind dort die fortgesetzte Interaktion des Programms mit seinem Umfeld, in der die Eingänge dynamisch verändert werden und so die Ausgänge ständig verändern.

Die meisten dieser Arbeiten legen Programmkonzepte wie Kontaktplan oder Funktionsblock orientierte Programmiersprachen zu Grunde. In der Arbeit von Rausch und Krogh [RK98] wurde die Transformation von SPS-Programmen untersucht. Sie entwickelten einen Algorithmus, der in Kontaktplan dargestellte SPS-Programme in SMV (*symbolic model verifier*) konvertieren konnte. Ihr Algorithmus war auf Boolesche Algebra beschränkt.

Weitere Beschränkungen betrafen die statische Zuordnung von Variablen, die keine speziellen Funktionen und keine Sprünge ermöglichen. Mit dieser Methode können Programme nur für SPS-Steuerungen der ersten Generation überprüft werden.[10] Die Umwandlung in symbolische Schreibweise wird normalerweise durch Definieren von Kripke Strukturen und das Übertragen von diesen über funktionelle Strukturen in SMV oder NuSMV (*a new symbolic model checker*) Modelle gemacht. Die Beweisführung erfolgt mit Hilfe der gegebenen Spezifikation zusammen mit dem Modell.

O. Pavlovic, R. Pinger und M. Kollmann [Po et al. 07] beschreiben in ihrer Arbeit ein Werkzeug, das vollständig automatisiert die Transformation von SPS-Programmen für sicherheitskritische Systeme in Modelle des NuSMV Model-Checkers herstellt. Als Grundlage dafür nützen sie die Ergebnisse von Fendoglu [FG07], der die Transformation eines AWL-Modells für den NuSMV Model-Checker beschreibt, um eine Metabeschreibung für AWL-Modelle zu erstellen.

Im Ausblick der Arbeit von Pavlovic und anderen [Po et al. 07] wird auch darauf hingewiesen, dass Methoden zur Minderung des state explosion Problems auch im Zusammenhang mit der Modellgröße erst noch gelöst werden müssen. Diese Aussage widerspricht zum Teil dem Anspruch von Model-Checking, die Grenzen des state explosion Problems weitgehend überwunden zu haben. Auf das Formalisieren der Spezifikation wird in der genannten Arbeit nicht eingegangen.

---

[10] Diese frühen Konzepte der SPS-Programmarchitektur (bis etwa 1975) definierten so genannte „Rungs“ anstelle der in späteren Konzepten frei konfigurierbaren Netzwerke. Als Rungs wurden sehr kleine Netzwerke verstanden, in den nur vier, in einer späteren Weiterentwicklung fünf Programmelemente in vordefinierten Anordnungen zugelassen waren. Jeder Rung besaß eine genau diesem Rung zugeordnete Ausgabe. Nur wenige Arten dieser Rungs existierten und durch die festgelegte Konfiguration ihrer Elemente reichten wenige unterschiedliche BDDs zur Funktionsbeschreibung aus. Weitere frühe Konzepte der SPS-Programmarchitektur bis etwa 1985 überprüften den Gebrauch von zugewiesenen Ausgaben und erlaubten internen oder externen Ausgaben nur die einmalige Verwendung. In späteren Konzepten wurde diese Einschränkung fallen gelassen.

Susta [RS02] zeigt eine Abstraktion von einer SPS, die er APLC genannt hat, die eine universelle Basis für das Modellieren von SPS und die Umwandlung ihrer Programme schafft. Er zeigt, wie eine binäre SPS als Mealy-Automat modelliert werden kann, dessen Programm als ein APLC-Programm vorliegt.

Kleuker behandelt in seiner Arbeit „Formale Modelle der Softwareentwicklung, Model-Checking, Verifikation, Analyse und Simulation“ [KS09] formale Modelle der Softwareentwicklung, Model-Checking, Verifikation, Analyse und Simulation. Für ihn sind auch komplexe Eigenschaften von Programmen mit einiger Erfahrung auf Basis der Programmsemantik beweisbar. Insbesondere ist dies für ihn der Fall bei imperativen Programmiersprachen. In seinem Werk werden die Befehlsarten Nacheinander-Ausführung, Alternativen und Schleifen hervorgehoben, mit denen alle Programme geschrieben werden können.

In der vorliegenden Arbeit ist es insbesondere die Nacheinander-Ausführung von Befehlszeilen, die genau den Ablauf eines SPS-Programms beschreibt. Kleuker führt weiter aus: „Dass eine automatische Verifikation nicht möglich ist, folgt aus der Unentscheidbarkeit des Halteproblems. Trotzdem kann man zeigen, dass man interaktiv, also von Hand, sehr komplexe Eigenschaften von Programmen mit einiger Erfahrung auf Basis der Programmsemantik beweisen kann [KS09, Seite 201].“ Die Unentscheidbarkeit des Halteproblems wird dadurch umgangen, dass dezidierte Haltepunkte jeweils am Ende eines SPS-Zyklus zu Grunde gelegt werden können. Dadurch wird immer ein kompletter Zyklus nach dem anderen verifiziert.

Auch Olderog und Wilhelm [OW12] haben in ihrem Beitrag „Weitere Entwicklung der Programmverifikation“ die axiomatische Methode zur Programmverifikation als Weiterentwicklung mit der von Turing vorgeschlagenen Vorgehensweise in Zusammenhang gebracht. „Diese Methode benötigt eine geeignete Sprache zur Formulierung von Programmeigenschaften, z.B. Formeln der Prädikatenlogik. Mit diesen Formeln lässt sich das gewünschte Programmverhalten spezifizieren. Aus der Logik wird auch der Begriff des Beweissystems übernommen, das aus Axiomen und Regeln besteht, mit denen sich formal beweisen lässt, ob ein gegebenes Programm dem spezifizierten Programmverhalten genügt [OW12, Seite 271]“.

In [KP12] wird ein derartiges Beweissystem in PROLOG vorgestellt. Olderog und Wilhelm führen weiter aus, dass eine automatisierte Methodik nicht gleichzeitig vollständig und korrekt sein kann, sondern dass Einschränkungen gemacht werden müssen (vgl. [OW12]). Weil eine zu fordernde Vollständigkeit sehr schnell in der „state explosion“ Problematik mündet, wird diesem Umstand in der vorliegenden Arbeit durch die Zerlegung des untersuchten SPS-Programms in beherrschbare Teilbereiche entgegengewirkt.

Soliman und Frey behandeln in ihrem Beitrag „Verifikation und Validierung sicherheitsgerichteter SPS-Programme“ [SF13] eine Methodik zur Verifikation und Validierung sicherheitsgerichteter SPS-Programme, die zuerst in ein System zeitbehafteter Automaten überführt und anschließend durch den UPPAAL-Model-Checker formal verifiziert und simulativ validiert werden. Dazu werden Funktionsblöcke einerseits durch Simulation validiert und andererseits durch einen Model-Checker verifiziert. Die genutzten Bausteine sind als Sicherheitsfunktionen definiert (z.B. Notabschaltsysteme und Sicherheitsüberwachungen) und benötigen deshalb nur eine vergleichsweise geringe Anzahl von Signalen. Typisch für sicherheitsgerichtete Steuerungen ist die Verwendung derartiger konfigurierbarer Funktionen bzw. Funktionsbausteine. Interessant ist die gezeigte Möglichkeit, Testsequenzen in das formale Modell zu importieren.

Canet und andere zeigen in ihrer Arbeit „Towards the Automatic Verification of PLC Programs Written in Instruction List“ [Cg. et al. 00] die automatische Verifikation eines SPS-Programms in Anweisungsliste mittels Model-Checkings. Das zugrunde gelegte Programm besitzt eine nur geringe Komplexität und kommt daher in Summe mit fünf Variablen aus. Neben der Formalisierung des SPS-Programms werden für den Model-Checker Algorithmen für den Programmzähler und den Akkumulator der SPS formalisiert. Die Aufgabenstellung wird dazu in linearer temporaler Logik (LTL) dargestellt. Canet zeigt auch, dass die Formalisierung trotz einer vergleichsweise einfachen Aufgabenstellung in der Umsetzung schwierig ist.

Abgesehen von der geringen Komplexität des Beispielprogramms stehen in der Regel bei Aufgabenstellungen für SPS-Steuerungssysteme fast nie formale Spezifikationen der Anwenderprogramme zur Verfügung. Der Aufwand für die Formalisierung wächst mit der Komplexität der Programme. Hilfreich ist dabei, die Aufgabenstellung in funktionale Gruppen zu gliedern.

Bhoi und andere behandeln in ihrer Arbeit „PLC verifcation using symbolic model checking” [Br et al 08] das Sprachkonzept der Kontaktplanprogrammierung (Relay ladder logic, RLL) und stellen einen Logik-Analysator als Werkzeug für die Verifikation vor. Dazu wird das SPS-Programm als Modell der Computation Tree Logic (CTL) formal beschrieben und untersucht. Als Model-Checker werden finite state machines (FSMs) genutzt, die durch Kripke Strukturen repräsentiert werden. Als Studie wird ein sehr einfaches Programm für einen Mikrowellenofen diskutiert. Ein offensichtlicher Programmfehler wird dabei erkannt.[11]

Biallas und andere [Bs et al. 12] stellen in ihrer Arbeit „Arcade. PLC: A Verification Platform for Programmable Logic Controllers“ unter der Bezeichnung Arcade.PLC ein Software

[11] Kurzbeschreibung Programm Mikrowellenofen: Türe zu, Heizung ein, Heizzeit verstrichen, Heizung aus.

Werkzeug vor, das so genannte „program organization units“ (POU) verifiziert. Unter POUs werden in diesem Zusammenhang abgeschlossene Funktionen eines SPS-Programms verstanden, die mittels Model-Checking verifiziert werden. Biallas führt weiter aus, dass beginnend von einem Startzustand (initial state) iterativ Nachfolgezustände kreiert werden, bis so genannte fixierte Punkte (fixed point) erreicht sind. Diese bilden den erreichbaren Vergleichsbereich für das Model-Checking, wobei die Eingangszustände nicht genutzt werden. Weiters wird ausgeführt, dass Arcade PLC zur Ermittlung seiner Leistungsfähigkeit für sicherheitskritische Funktionen mit bis zu 14 Eingangssignalen untersucht worden ist.

Diskussion: Mit dem Einbeziehen einer Spezifikation müssen auch die Eingangszustände berücksichtigt werden. Die vorgestellte Methodik stellt nach Anwendung der Spezifikation fest, ob der Bereich der Ausgangszustände erreicht wird, unabhängig von der Konfiguration der Eingangszustände. Aus dieser Sichtweise müssten weitere Untersuchungsschritte folgen.

Der in „Schritte zur Verifikation für SPS-Programme“ [KP12] vorgestellte Verifikator besitzt zum Teil ähnliche Funktionalitäten. Sind die Eingangszustände als „offene“ Referenzen definiert, werden alle Variationen expandiert und auf Ausgangszustände abgebildet. So genutzt emuliert der Verifikator alle erreichbaren Ausgangszustände. Aus Sicht des Verifikators ist das untersuchte Programm dann und nur dann verifiziert, wenn auch die logische Zuordnung von Ein- auf Ausgangszustände der Übergänge gegeben ist.

Hametner unterstreicht mit seiner Arbeit „Test Driven Software Development for Improving the Quality of Control Software for Industrial Automation Systems“ [Hr13] die Notwendigkeit, die Qualität von Steuerungssoftware zu steigern. Insbesondere ist durch Wiederverwendung mit Anpassungen an eine bestehende Steuerungssoftware zu beobachten, dass unter anderem durch die Überarbeitung die Gesamtqualität einer SPS-Software auch beeinträchtigt werden kann. Hametner stellt dazu Testmethoden vor. Ein unmittelbarer Bezug zur vorliegenden Arbeit ist nicht gegeben jedoch kann die Notwendigkeit aus seinen Ausführungen abgeleitet werden, dass insbesondere bei der Überarbeitung und Anpassungen zur Optimierung von Steuerabläufen von SPS-Software ein erhöhter Bedarf an Verifikation gegeben ist.

Bender und andere stellen in ihrer Arbeit „Ladder Metamodeling and PLC Program Validation through Time Petri Nets“ [Bd et al. 08] einen Ansatz vor, in dem ein in Kontaktplan erstelltes SPS-Programm mit Hilfe von Metamodellen in eine zeitbehaftete Petri Netz Semantik überführt wird. Begründet wird die Vorgehensweise auch damit, dass in Programmen, die in Kontaktplan erstellt worden sind, Programm Konstrukte enthalten sein können, die in jedem Zyklus ihren Zustand wechseln. Solche Zustandswechsel können nicht einfach erkannt werden. Das aus der zeitbehafteten Petri Netz Semantik entstandene Metamodell wird mit dem

Model-Checker Tina auf zeitliche Inkonsistenzen überprüft. Bender spricht in diesem Zusammenhang sogar von Validation.[12]

Das Einbeziehen zeitlicher Aspekte in die Verifikation ist im in dieser Arbeit vorgeschlagenen Konzept auch gegeben, benötigt aber wenigstens zwei Überprüfungsschritte. Weiter oben wurde ausgeführt, dass für die Validation das SPS-Programm alleine nicht ausreicht.

Mader und Wupper stellen in ihrer Arbeit „Timed Automaton Models for Simple Programmable Logic Controllers“ [MW99] die Modellierung von Zeitfunktionen vor. Das vorgestellte Modell wurde entwickelt, um bei Echtzeit SPS-Programmen die zeitlichen Aspekte in die Formalisierung der Programme einzubringen.

In den Beispielanwendungen für Programmfragmente von SPS-Programmen wird die Modellierung von Zeitfunktionen in der Regel vernachlässigt. Grund dafür könnte sein, dass bei der Betrachtung einzelner Zyklen des SPS-Programms die Frage nach einer Zeitfunktion sich auf die Frage reduzieren lässt, ob eine Zeitfunktion bereits abgelaufen ist oder nicht.

Kowalewski und andere setzen sich in ihrer Arbeit „An Environment for Model-Checking of Logic Control Systems with Hybrid Dynamics“ [Ks et al. 99] mit der Verifikation eines Ablaufes für einen chemisch technologischen Prozess auseinander. Vorgestellt wird die Modellierungsumgebung „VERDICT“ (Verification of Discrete Controllers for Continuous Processes). Es werden Teilfragestellungen betrachtet, wie z.B. der Füllprozess eines Reaktors verifiziert werden kann oder die Kontrolle über die exotherme Reaktion im Reaktor gesichert werden kann. VERDICT ist dabei kein Analysewerkzeug, sondern ein Interface. VERDICT ist eine Plattform, die als Architektur für Sicherheitseinrichtungen interpretiert werden kann. Es werden dabei die Zusammenhänge modellierter hierarchischer Prozesse mit Steuerungsmodellen betrachtet mit dem Ziel, die Prozesssicherheit zu gewährleisten.

Xiying und Litz nutzen in ihrer Arbeit „Model checking of signal interpreted Petri nets“ [XL01] Signal interpretierte Petri-Netze und den Cadence-SMV Model-Checker. Einer Aufgabenstellung zu Grunde gelegt sind Ketten von Transitionen, die aneinandergereiht sind und auch mittels SPS abgearbeitet werden.

Das Formalisieren eines SPS-Programms aus dem Quellcode um daraus ein Modell zu generieren, wird in der Literatur für mehrere Sprachkonzepte nach IEC 61131-3 für SPS diskutiert. Weitere Arbeiten beschäftigen sich damit, das Formalisieren eines SPS-Programms zu verein-

[12] Die als so genannte „Instabilitäten“ in einem SPS-Programm bezeichneten zeitlichen Inkonsistenzen müssen nicht notwendigerweise Programmierfehler sein. In der Praxis kann ein sich von Zyklus zu Zyklus ständig wechselnder Zustand auch gewollt zum Zählen von Zyklen genutzt werden.

fachen, indem die zu lösenden Steuerungsaufgaben bereits in der Form von formalen Beschreibungskonzepten als eine Vorstufe des SPS-Programms definiert werden. Im Wesentlichen entsteht damit aus der informellen Spezifikation eine formale Spezifikation der Steuerungsalgorithmen, die in einem weiteren Schritt in ein lauffähiges SPS-Programm übertragen werden müssen.

1.2.7 Zusammenfassung

Zusammenfassend ist anzumerken, dass in allen der genannten Arbeiten Einschränkungen hinsichtlich des Befehlsumfanges der SPS gemacht worden sind. Darüber hinaus werden nur vergleichsweise einfache, manchmal sogar nur triviale, Programm Konstrukte zu Grunde gelegt und untersucht. Ihr Nutzen liegt aus dieser Sicht in der theoretischen Betrachtung des Problems, für die praktische Anwendung sind nicht unerhebliche Erweiterungen notwendig.

Darüber hinaus fällt auf, dass einerseits das state explosion Problem in den in der Literatur genannten Arbeiten als gelöst betrachtet wird, jedoch andererseits auch auf weiteren Handlungsbedarf zur Lösung zu dieser Problematik verwiesen wird. Die in der Literatur untersuchten Fallbeispiele zeichnen sich nicht durch eine hohe Komplexität aus.

Alle heute verwendeten Programmkonzepte für die SPS-Programmierung sind mehrsprachig. Fast alle diese Konzepte unterstützen die Programmierung in Anweisungslisten. Das Sprachkonzept AWL ist vergleichbar mit Assembler als eine maschinennahe Programmierung. Deshalb sind fast alle verwendeten anderen Sprachkonzepte nach AWL übertragbar. Daher wird auch das Hauptaugenmerk in dieser Arbeit auf AWL als der universelle Programmdialekt für SPS gelegt und weil vielfach auch andere SPS-Sprachkonzepte automatisch in AWL übertragen werden können.

Für die Modellierung existieren im Allgemeinen keine festen Regeln, wie welches System als Transitionssystem zu modellieren ist. Zustände sind üblicherweise auf natürliche Weise gegeben, wobei Abstraktionen möglich sind. Sind z.B. bei einem Schaltkreis Eingabe, Ausgabe und ein interner Speicher gegeben, können die Zustände durch die Belegung der Eingabe und des Speichers eindeutig bestimmt werden. Auch die Übergänge sind auf natürliche Weise gegeben. Abstraktion sind in der Form von if-then-else Konstruktionen möglich.[13]

---

[13] Insbesondere SPS-Anweisungen eignen sich gut für die Modellierung nach diesem Schema. Seien die Eingabe x, die Ausgabe y und der interne Speicher r boolesche Variable, dann gilt:
$\{x, r\} \rightarrow \{y\}$ und weiter für Logisch UND:
$\{x = 0, r = 0\} \rightarrow \{y = 0\}$,
$\{x = 0, r = 1\} \rightarrow \{y = 0\}$,
$\{x = 1, r = 0\} \rightarrow \{y = 0\}$,

Für die Modellbeschreibung von SPS-Programmen werden bei Model-Checkern einerseits Petri-Netze zugrunde gelegt, andererseits mittels Computation Tree Logic oder Linear Temporal Logic die Modelle beschrieben.

Als automatisierte Verifikationstechnik wird in der Literatur Model-Checking favorisiert. Model-Checking ist eine Methode für die formelle Prüfung, basierend auf einem Transitionssystem oder einer Kripke Struktur. Die formelle Spezifikation ist eine Art mathematischer Ausdruck. Eine Spezifikation eines Systems kann ausgedrückt werden als eine temporäre logische Formel, wie diese bei Model-Checking angewendet wird. Model-Checking verwendet Modelle M, um zu überprüfen, ob gegebene logische Formeln φ erfüllbar sind oder nicht. Werkzeuge für das Model-Checking wie SPIN [HG97] und SMV [KM96], [MC10], oder NuSMV [CC10] benötigen Modelle, die in die Sprache des verwendeten Model-Checkers übersetzt sind. SPS arbeiten verschieden zu anderen Prozessrechnersystemen. Ihre spezielle Arbeitsweise bei der Programmausführung, ein sich ständig wiederholender Zyklus, begünstigt die in der Folge vorgestellte Methode für die Verifikation. Die Struktur eines SPS-Programms und die Kenntnis, wie eine SPS arbeitet, scheint es relativ einfach zu machen, solche SPS-Programme in die lineare temporale Logik (*engl. Linear Temporal Logic*, LTL) zu übertragen um die entsprechenden LTL-Formeln zu generieren. Dort wird die Programmausführung einer SPS als ein Pfad π beschrieben, in dem die Statusinformationen schrittweise weiterentwickelt werden.

Sei $\pi = s_0 \to s_1 \to s_2 \to ....$ Pfad, der durch einen Satz von Statusinformationen führt. Wenn diese Statusinformationen s_α als die Kombination der Eingänge (Sensoren), Ausgänge (Aktoren) und interner Zwischenergebnisse definiert sind, gilt:

$$s_\alpha = s_{in} \times s_{out} \times s_{inter},\ \alpha = 0,1...i...j...k;\ in = 0,1...i;\ out = (i+1)...j;\ inter = (j+1)...k$$

*(engl. **inp**uts (sensors), **out**puts (actuators), **inter**nal intermediate results)*

Eine typische Bedingung für Überprüfung von SPS-Programmen ist gegeben durch eine Formel φ in temporaler Logik in einem Modell M mit einem Anfangszustand s Dabei muss entschieden werden, ob im Modell M für alle Anfangszustände s diese Bedingung erfüllt ist oder nicht. LTL Formeln werden in der BNF Form definiert als: [14]

$\{x = 1, r = 1\} \to \{y = 1\}$.

[14] Die lineare temporaler Logik ist definitionsgemäß aus Aussagevariablen (p), Wahrheitswerten ($\perp$ false, T true), logischen Verknüpfungen ($\neg$, $\wedge$, $\vee$, $\to$) und temporalen Operatoren aufgebaut. Der Operator X bezeichnet den Nachfolger (next), G steht für global (globally), F wird für einen künftigen Zustand verwendet (finally, future), U bedeutet bis zu einen Zustand (until) und R steht für die Auflösung (release). Durch geeignete logische Verknüpfungen sind einige Operatoren durch andere ausdrückbar.

$$\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \mathsf{U} \varphi) \mid (\mathsf{G}\varphi) \mid (\mathsf{F}\varphi) \mid (\mathsf{X}\varphi).$$

Im vorliegenden Fall wird der LTL-Operator X (mit der Bedeutung „next“) angewendet, zu interpretieren als: $\mathsf{X}\varphi : \varphi$ der Formelausdruck φ muss für den nachfolgenden Zustand gültig sein.

Im darauffolgenden Schritt muss überprüft werden, ob der Ausdruck $\pi_{\alpha,\beta}$ für einen Pfad durch das Programm erfüllbar ist oder nicht. Dazu werden alle Zustandsübergänge geprüft. Es muss gelten:

$$\pi_{\alpha,\beta} \vDash \sum_{\alpha,\beta=1}^{n} \left(s_\alpha \rightarrow s_\beta \right)$$

$\pi_{\alpha,\beta}$ ist dabei jener Pfad, der erfüllt sein muss, wenn ein Übergang aus dem Zustandes s_α in den Folgezustand s_β stattfindet.

Normalerweise findet die Änderung der Eingangsinformation spätestens nach einigen Zykluszeiten statt. Deshalb treten die meisten Übertragungen $s_\alpha \rightarrow s_\beta$ auch als Übertragungen $s_\alpha \rightarrow s_\alpha$ zwischenzeitig auf. Die Übertragsfunktion wird reduziert auf die Definition der auftretenden Paare von Eingangs-Status zu Ausgangs-Status. Aber nicht alle derartigen Übergänge sind mögliche Übergänge. Dazu müssen die entsprechenden Paare zuerst gewählt werden. Dieser Umstand wird später noch genauer behandelt.[15] Prinzipiell werden diese Statusinformationen durch Sätze von Eingangs- und Ausgangsvektoren repräsentiert. Programmkorrektheit ist gegeben, wenn alle möglichen Kombinationen die Pfadbedingungen, hier das SPS-Programm, erfüllen.

Als gegeben angenommen wird eine formale Spezifikation der Aufgabenstellung für die zu verifizierenden SPS-Programme. Solche formalen Spezifikationen sind in der Praxis jedoch die Ausnahme und nicht die Regel. Dazu kommt, dass ein SPS-Programm auf Grund der Fähigkeit, es rasch an neue Erfordernisse oder auch nur durch Verbesserungen in seinen Abläufen anzupassen, bereits nach vergleichsweise sehr kurzer Zeit z.B. am Ende der Inbetriebsetzungsphase und vor Beginn des eigentlichen Produktionsbetriebs, nicht mehr einer vorgegebenen Spezifikation im ursprünglichen Umfang entspricht.

---

[15] $s_\alpha \rightarrow s_\alpha$ bedeutet, dass solange sich keine Eingangsinformationen in einem realen System verändern die SPS scheinbar im zugeordneten Zustand verharrt. Tatsächlich wird jedoch das gesamte Programm zyklisch wiederholt. Erst wenn sich die Eingangsinformation verändert wird ein neuer Ausgangszustand errechnet. Daraus folgt, dass jeder Ausgangszustand von genau seiner erzeugenden Eingangsinformation abhängt, dessen Status die Funktion seiner Eingangs-Information ist. Bei der Verifikation werden solche paarweise Zusammenhänge überprüft.

Trotz der Fortschritte im Bereich der Rechnerleistung und der Einführung neuer Berechnungsmethoden ist das state explosion Problem nicht nur bei der Verifikation von SPS-Programmen noch nicht ausreichend gelöst. Die Lösungsansätze konzentrieren sich dabei darauf, Grenzen zwischen erreichbaren und nicht erreichbaren Zustandsräumen zu ziehen.

Hauptansätze in der vorliegenden Arbeit

Bereits in „Schritte zur Verifikation von SPS-Programmen“ [KP12] wurde davon ausgegangen, dass es für Steuerungsaufgaben und dazu eingesetzten Automatisierungssystemen branchenüblich ist, Aufgabenstellungen verbal zu beschreiben und gegebenenfalls durch Ablaufskizzen näher zu erklären. Die Unzulänglichkeit der natürlichen Sprache gegenüber formaler Beschreibungen ist die eine Seite, die andere Seite ist, dass die meisten Ersteller von Aufgabenstellungen formale Beschreibungssprachen gar nicht kennen. Daher wurden als Ersatz für die formale Spezifikation die Beschreibung der Anfangssituation und das zu erreichende Ergebnis definiert.

Auch der Verifikator in [KP12] ist durch das state explosion problem begrenzt. Eine kleinteilige Lösung, wie diese in der Literatur häufig bereits als finale Lösung der Problemstellung gesehen wird, ist aus Sicht dieses Ansatzes nicht ausreichend. Hauptgrund dafür ist, dass aus dem Zusammenhang einer Funktionalität gerissene verifizierte Einzelelemente keine seriöse Aussage zur gesamten Funktionalität eines SPS-Programms zulassen.

Aus diesem Grund beschäftigen sich die im Weiteren vorgestellten Ansätze damit, wie zusammenhängende Programmbereiche voneinander abgegrenzt werden können, ohne Programmzusammenhänge zu vernachlässigen. Dazu muss ein Kompromiss zwischen trivialer Kleinteiligkeit der untersuchten Bereiche zur Größe der Programmfragmente gefunden werden, um auch die Berechenbarkeit auf Grund gegebener Systemgrenzen noch zu gewährleisten.

Die formale Spezifikation der SPS-Programme wird ersetzt durch die Summe aller im gegebenen Ansatz geordneten Zustandsübergänge. Anzumerken ist, dass solche geordneten Zustandsübergänge auch den Model-Checkern zugänglich gemacht werden können.

In der vorliegenden Arbeit wird das SPS-Programm direkt formalisiert, also ohne Zwischenschritte z.B. über Petri-Netze. Bei den Spezifikationen werden bei den in der Literatur beschriebenen Ansätzen formale Spezifikationen der Aufgabenstellung zu Grunde gelegt, die in der Regel erst erstellt werden müssen.

Modelldarstellung

Die Abbildungsrelation der Eingangskonfiguration auf die Ausgangskonfiguration ist durch das SPS-Programm gegeben. Im Prinzip gibt diese Relation die Spezifikation für das SPS-Programm wider. Die Summe aller Abbildungen ist bezogen auf jede einzelne Abbildung dieser Relation insoweit eindeutig, dass zu jeder definierten Eingangskonfiguration aus der Menge der Eingangszustände zuzüglich von Ausgangszuständen des vorhergehenden Programmzyklus genau eine Ausgangskonfiguration aus der Menge der Ausgangszustände entspricht. Der weiter oben besprochene Zustandsraum beinhaltet diese Zustände.

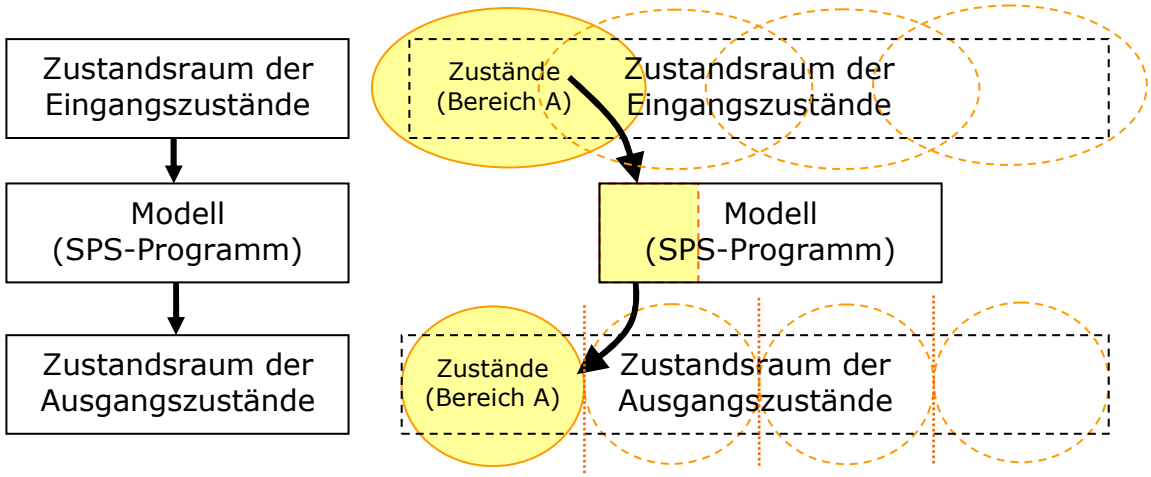


Fig. 1: Zustandsraum und Zuordnung

Fig. 1 skizziert den Zustandsraum der Ein- und Ausgangszustände und ordnet diese dem modellierten SPS-Programm zu. Hauptzweck der Darstellung ist, die Teilung des SPS-Programms in Bereiche zu veranschaulichen und die damit verbundenen Eingrenzungen. Das linke Teilbild stellt das Gesamtsystem dar. Im rechten Teilbild sind die Zustandsräume segmentiert und in Bereiche zerlegt. Das wesentliche Merkmal des Zustandsraumes der Eingangszustände ist, dass Überschneidungen einzelner Segmente zulässig sind, im Zustandsraum der Ausgangszustände hingegen nicht. Begründet ist dieser Umstand durch die Eindeutigkeit der Ausgangszustände, bei denen jeder Ausgangszustand einem bestimmten Bereich zugeordnet ist. Während der Zustandsraum der Eingangszustände Überschneidungen aufweisen kann, kann der Zustandsraum der Ausgangszustände in abgegrenzte Bereiche geordnet werden. Eine zweckmäßige Ordnung ist z.B. die Teilung in Funktionen einer Aufgabenstellung. Zusätzlich gilt, dass auch der Bereich des SPS-Programms segmentiert werden kann und zwar genau in jene Teile der Befehlsfolgen, die Zustände im abgegrenzten Bereich beeinflussen. Derartig identifizierbare Bereiche müssen im SPS-Programm nicht notwendigerweise zusammenhängen, sondern können auch im zu untersuchenden SPS-Programm verteilt ange-

ordnet sein. Dazu notwendige Eingrenzungen werden durch die SPS-Programmanalyse getroffen.

Prinzipiell ist die Abgrenzung der Bereiche beliebig. Zweckmäßigerweise wird die Abgrenzung nach den Gesichtspunkten Größe und Zugehörigkeit erfolgen. Eine Größenbegrenzung des zu untersuchenden Zustandsraums verringert den Umfang der Untersuchungsfälle (state explosion problem), die Festlegung auf funktionell voneinander abhängigen Funktionalitäten im SPS-Programm erleichtert die Bestimmung der Zustandsübergänge und stellt den Kontext zur Spezifikation her. Die einzige Forderung ist, dass der Zustandsraum der Ausgangszustände vollständig abgedeckt wird. Welche Eingangszustände des Zustandsraums der Eingangszustände die Erzeugenden der Systemreaktion sind, werden durch die später diskutierten Analysetechniken bestimmt,

Im Gegensatz zum Model-Checking wird mangels geeigneter formaler Spezifikationen nur das SPS-Programm soweit formalisiert, dass das Programm selbst als Modell betrachtet werden kann. Anstelle eines weiteren Modells wird der Zustandsraum der Ein- bzw. Ausgangszustände genutzt, der auf Grund seiner Mächtigkeit in Segmente zerlegt wird. So gesehen erfolgt im Verifikationsschritt jeweils ein Vergleich des Modells des SPS-Programms mit den Paaren zugehöriger Ein- und Ausgangszustände aus dem Zustandsraum.

Informelle Spezifikationen eines Programms können aus mehreren Gründen nicht vollständig sein. Einerseits tragen die Redundanz der Sprache und andererseits unklare Formulierungen in der Aufgabenstellung zu Fehlstellen oder Fehlern in der Spezifikation bei. Zusätzlich können auch Teile der Aufgabenstellung gar nicht spezifiziert sein, damit existieren also weitere Fehlstellen. Die Konsequenz von Fehlern sind notwendige Korrekturen.

Eine weitere Konsequenz von Fehlstellen ist, dass auch nicht spezifizierte Zustände mit dem Programmmodell simuliert werden können. Die auf diese Art gewonnenen Resultate können retrospektiv einer kritischen Prüfung unterzogen werden, womit sich aber der Aufwand entsprechend erhöht. Grundsätzlich ist die Simulation eine verbreitete Methode für die Verifikation aber auch für die Validation. So werden bei der Methode der Fehlerunterstellung Fehlsignale erzeugt, um festzustellen, ob diese entsprechend erkannt werden und das Gesamtsystem adäquat reagiert. Dazu werden besonders, wenn es eine sehr große Anzahl von Eingangs- bzw. Ausgangssignalen gibt, kritische Bereiche eines Prozessablaufes genauer untersucht.

Vollständigkeit des Untersuchungsraums

Das Sicherstellen der Vollständigkeit des Untersuchungsraumes von Befehlsfolgen der SPS-Programme ist entscheidend für die Vollständigkeit der Verifikation. Insbesondere stellt sich

diese Frage, wenn Teile der SPS-Programme nacheinander verifiziert werden. Gleichzeitig muss auch die Frage behandelt werden, ob sich so untersuchte Programmteile gegenseitig beeinflussen.

Dazu ist es notwendig, die Art der Zerlegung von SPS-Programmen in Programmsegmente zu definieren. Ausgangspunkt ist dabei die Überlegung, dass nur jene Elemente einer SPS etwas bewirken, die an die Ausgänge angeschlossen sind. Grundsätzlich ist es daher völlig ausreichend, jene Teilbereiche im Adressraum der Ausgänge einzuteilen, die an der gesteuerten Einrichtung nach außen hin etwas bewirken.

Der erste Teil der Untersuchung dient der Feststellung, welche Programmzustände aus dem Zustandsraum der Eingangszustände benötigt werden, um einen bestimmten Ausgang zu steuern. Dieser Vorgang, später vorgestellt als Rückwärtsanalyse, wird für jeden im Programm verwendeten Ausgang durchgeführt. Dabei wird der weitere Teilbereich im Adressraum mitberücksichtigt, der für interne Verknüpfungen Zwischenergebnisse speichert.

In einem weiteren Schritt wird festgestellt, inwieweit sich Elemente aus der Menge der Eingänge des Adressraumes, der Ausgänge und Zwischenergebnisse gruppieren lassen, also nach Möglichkeit einen zusammenhängenden Bereich zugeordnet werden können. Derartige Bereiche sind die weiter oben als funktionelle Bereiche oder als Funktionen identifizierbaren Funktionalitäten einer Steuerungsaufgabe. Ihre praktische Bedeutung ist, dass solche Funktionalitäten ihre Entsprechung in einem Teilprogramm des SPS-Programms besitzen. Anzumerken ist dabei, dass solche Teilprogramme nicht notwendigerweise zusammenhängen müssen, wodurch die Eingrenzung der Bereiche erschwert wird.

Es existieren mehrere Möglichkeiten, solche Eingrenzungen praktikabel vorzunehmen. Die wichtigste Möglichkeit ist, auf Grund einer genauen Kenntnis der Aufgabenstellung oder aus der Spezifikation Teilfunktionen zu extrahieren, wo z.B. zugehörige Ausgänge und die angeschlossenen Aktuatoren einer Maschinenfunktion zugeordnet worden sind. Eine weitere Möglichkeit ist im Umstand begründet, dass bei der Programmierung einer SPS Teilfunktion üblicherweise die Befehlsfolgen im SPS-Programms gruppiert positioniert sind, obwohl der Reihenfolge von Befehlsfolgen in den meisten Fällen keine Bedeutung zuzumessen ist. Vielfach lassen sich dadurch Vorkommensbereiche von Ein- bzw. Ausgangsreferenzen im Programmkontext bestimmen.[16]

[16] Beispiel: Seien Ein- bzw. Ausgänge einer SPS mit E_i und A_j bezeichnet und ergibt die Überprüfung der Adressreferenzen, dass die Häufigkeit der Verwendung bestimmter Eingänge und Ausgänge in einem Teilbereich eines SPS-Programms groß ist, liegt die Vermutung nahe, dass solche Teilbereiche auch funktionelle Zusammenhänge besitzen. Eindeutig wird der Zusammenhang, wenn die mit E_i und A_j bezeichneten Referenzen exklusiv Programmabschnitten zugeordnet werden können.

Zusammengefasst kann festgestellt werden: wurden alle im SPS-Programm verwendeten Ausgänge erfasst, ist der Bereich der Ausgänge vollständig. Wurden alle Elemente in der Rückwärtsanalyse erfasst, die die auf diese Art behandelten Ausgänge in Relation mit den die Ausgänge steuernden Elemente zuordnen, ist der gesamte Adressierungsbereich und damit auch der gesamte Zustandsraum abgedeckt und kann für die Verifikation bereitgestellt werden.

1.2.8 Abstraktion

In dieser Arbeit wird eine SPS als ein Spezialfall von Systemen betrachtet, die mit Statusinformationen arbeiten. Dabei liegt das Hauptaugenmerk auf Zuständen, die das System einnimmt, indem eine Kette invarianter Beziehungen berücksichtigt wird, die von einem Anfangs-Zustand zu einem End-Zustand durchlaufen werden. Dynamische Aspekte sind der Zusammenhang zwischen Eingangsgrößen und Ausgangsgrößen und die internen Veränderungen von Zuständen durch die Programmbearbeitung. Später wird gezeigt, dass diese dynamischen Aspekte durch die SPS und ihre Programmoperationen bereits gegeben sind.

Die intuitive Idee, dass die Semantik einer Programmiersprache ein mehr oder weniger genaues Betrachtungsniveau für die Beobachtung der Programmausführung ist, führt dazu, Programme als kurze Zusammenfassung in Interpretationen zu formalisieren. Dieses kann auf verschiedene Sprachen angewendet werden. P. Cousot [CP97] zeigte, dass die klassische Semantik von Programmen, die als Transitions-Systeme modelliert sind, voneinander abgeleitet werden können.

Obwohl ein SPS Programm in einer Endlosschleife ausgeführt wird, genauer in einem unendlichen Pfad, existieren dennoch spezielle Zustände, die wenigstens innerhalb bestimmter Zeitschranken quasistationär bleiben.

Der hier genutzte Ansatz wendet die statische Programmanalyse an. Unter statischer Programmanalyse soll hier die automatische Bestimmung von dynamischen Vorgängen innerhalb der Laufzeit von Programmen verstanden werden. Die grundlegende Idee ist die Einführung eines Analysewerkzeugs, das überprüft, ob die Semantik in einem Programm die geforderte Spezifikation erfüllt oder nicht. Um Hinweise zum Ursprung von Fehlern oder Fehlverhalten zu liefern, kann zusätzlich z.B. ein Rückwärtsanalysator eingeführt werden, womit die zur Erfüllung einer Spezifikation notwendigen Bedingungen aufgedeckt werden können.

Im Allgemeinen definiert die Semantik einer Programmiersprache die Semantik jedes in dieser Sprache geschriebenen Programms. Darin enthalten ist ein formales mathematisches Modell vom möglichen Verhalten einer SPS, die ein Programm entsprechend ihren Eingangs-

und Ausgangszuständen beschreibt. Vereinfacht wird die Analyse, weil alle SPS Programme genau vordefinierte Programmpfade verwenden.

Ausgehend von der Art, wie Computer-Programme bearbeitet werden, können diese in solche, die transformationell arbeiten und solche, die reaktiv ablaufen, eingeteilt werden [HP85]. Eine SPS arbeitet reaktiv als Interaktion von Programm und Hardware. Werden jedoch die einzelnen Programmzyklen analysiert, ist ihr Programmverhalten transformationell. Die Aufgabenstellung in der hier vorliegenden Arbeit steht im Einklang mit Frances [NF92] als Verifikationsmodell eines Systems, das Statusinformationen von einem in den nächst folgenden Zustand transformiert. Genauer ausgedrückt wird durch die Transformation aller berechneten Statusinformationen ausgehend von jedem Anfangszustand jeweils ein Übergang in einen Endzustand als die Zusammenfassung für alle möglichen Übergänge des SPS-Programms innerhalb der Befehlsfolge des Programmdurchlaufes produziert. Die Methode ist vergleichbar mit dem Beweisen von Theoremen. Hier soll bewiesen werden, dass die zugrunde gelegte Spezifikation, festgelegt durch die Aufgabenstellung, durch Verwenden mathematischer Überlegungen erfüllt wird. Aufgrund der Tatsache, dass eine SPS nur einzelne Transformationsschritte von Eingabewerten in Ausgaben innerhalb eines einzelnen Programmzyklus ermöglicht, erlaubt es diese Transformationen unabhängig voneinander zu berechnen. Von diesem Standpunkt aus gesehen, kann eine SPS als statisches System für einzelne Zyklen betrachtet werden. Bei der Verifikation werden ausgehend von überprüften einzelnen Zyklen alle Programmdurchläufe bearbeitet, um das gesamte Programmverhalten solcher reaktiven Systeme zu überprüfen.[17]

Gegeben sei eine Spezifikation und ein entstehendes Programm, das überprüft werden muss, ob die Programmsemantik die Spezifikation erfüllt. Im Falle von Divergenzen soll der Analysator die Differenzen anzeigen und in einem weiteren Schritt sogar mögliche Ursachen für Fehler darstellen können. Verwendet wird ein in Prädikatenlogikformeln verwandeltes SPS Programm als eine abstrakte Interpretation.

In der Praxis enthält dieser Programmanalysator einen Generator, der den Originalprogrammtext anzeigt und Arbeitsfolgen bzw. Formeln entsprechend den Programmanweisungen produziert, deren Lösungen die Repräsentation der abstrakten Programmsemantik zeigen. Die abstrakte Interpretation der Spezifikation ist durch die Zustände ihrer Elemente gegeben. Die

[17] Es liegt die Vermutung nahe, dass sich Model-Checking als Verifikationswerkzeug für die Verifikation von SPS-Programmen deshalb nicht einfach einsetzen lässt, weil formale Spezifikationen von Aufgabenstellungen für SPS-Programme in der Regel die Ausnahme sind.

Unter der Annahme, es existiere eine formale Spezifikation für ein SPS-Programm, könnte auch gefragt werden, ob nicht auf Grund von Eigenschaften der formalen Beschreibungen das gesuchte SPS-Programm auch direkt - also ohne einen Programmierer zu bemühen - in den SPS-Code umgewandelt werden kann.

Gestaltung eines SPS-Programms beginnt mit einer Programmbeschreibung durch gegebene Eingangskonfigurationen zusammen mit der Frage, was mit diesen Konfigurationen geschehen soll. Das Ergebnis des Programmentwurfs sind die Konfigurationen der Ausgaben, die die erforderliche Funktionalität abdeckt.

Eine derartige statische Programmanalyse kann auch für große Programme ohne Benutzerinteraktion durchgeführt werden. Komplexitätsprobleme treten auf Grund der Anzahl von verschiedenen entsprechenden Paaren Eingabe- und Ausgabekonfigurationen auf, die unabhängig analysiert werden müssen. Da in automatisierten Geräten für Maschinen normalerweise mehr als eine Unterfunktionalität gegeben ist, können SPS-Programme in mehr oder weniger unabhängige funktionelle Abschnitte eingeteilt werden. Um die Auswirkung des so genannten „state explosion problem“ zu reduzieren, wird ein SPS-Programm in funktionelle Programmblöcke aufgetrennt. Zwischen solchen Blöcken, die unterschiedlich groß sein können, existieren Beziehungen mit gemeinsamer Funktionalität, wie z.B. Operationsmodi wie manueller oder automatischer Lauf. Andererseits sind manche der Programmteile auch völlig voneinander unabhängig.

Wie bereits ausgeführt, enspricht die in dieser Arbeit genutze Methode dem Verifikationsmodell nach Hoare, Die Methode wird schrittweise angewendet.

Sei die Menge aller auftretenden Bedingungen V_j^k gegeben mit $j\ [j ::= 0 \ldots i \ldots m]$ als Anzahl der Bedingungen und $k\ [k ::= 0 \ldots n]$ als Anzahl der Programmschritte, wobei V gleichzeitig Variable darstellen, gilt für die Programmdurchläufe:

$$V ::= V_0^0, V_0^1, \ldots V_0^n \mid V_1^0, V_1^1, \ldots V_1^n \mid \ldots\ldots \mid V_i^0, V_i^1, \ldots V_i^n \mid \ldots\ldots \mid V_m^0, V_m^1, \ldots V_m^n$$

Als Vorbedingung fungiert bei benachbarten Bedingungen jene mit dem niedrigeren Index. Die Bedingung mit dem höchsten Index von k ist die Nachbedingung. Ausgehend von einer gegebenen Startbedingung als Vaiablenwerte V_i, diese werden auch als Eingangsvektoren bezeichnet, wird die zugehörige Bedingung am Ende der zyklischen Programmbearbeitung, der Ausgangsvektor, schrittweise durch Bearbeitung des SPS-Programms entwickelt. V_i^n ist die zur Startbedingung korrespondierende Endbedingung. Zwischenergebnisse sind Konfigurationen, aus denen der Programmverlauf abgelesen werden kann. Diese entsprechen der Bezeichnung $K_i \rightarrow K_i^n$ und stellen den Übergang dar.

Der Programmzyklus i ist verifiziert, wenn gilt:

$$\{V_i\}\left(K_i \rightarrow K_i^n\right)\{V_i^n\}$$

Der Ausdruck beschreibt den i-ten Programmzyklus für ein SPS-Programm mit n Programmschritten. Startpunkt ist eine ausgewählte Speicherbelegung mit Zuständen, die der Eingangsbedingung für den untersuchten Programmzyklus i des SPS-Programms zugeordnet sind. Diese Zuordnung wird in die Eingangskonfiguration (im SPS-Programmschritt 0) übernommen. Bei jedem Befehl erfolgen entsprechend der Befehlssyntax Veränderungen in der Konfiguration. Bei jedem der folgenden Programmschritte wird eine neue Konfiguration erzeugt. Insbesondere dann, wenn die vorhergehenden Zustände in einer gerade erstellten neuen Konfiguration verschieden sind, unterscheiden sich diese Konfigurationen von jenen in anderen Programmzyklen. Dieser Vorgang wird solange fortgesetzt, bis der letzte Befehl in der Befehlskette des SPS-Programms abgearbeitet worden ist. Zuletzt wird die Endkonfiguration mit den zu erreichenden Ausgangsbedingungen verglichen. Wird Übereinstimmung festgestellt, ist dieser Verifikationsschritt erfolgreich abgeschlossen.[18]

Das gesamte Programm ist verifiziert, wenn alle Konfigurationen und damit alle Programmzyklen des untersuchten SPS-Programms verifiziert worden sind.

$$\mathbf{VERIFIKATION} = \bigwedge_{i=0}^{m} \{V_i\}\left(K_i \rightarrow K_i^n\right)\{V_i^n\}$$

Diese Methode eignet sich deshalb besonders, weil im Programmfortschritt schrittweise die Verifikationsbedingung entwickelt wird und unmittelbar im Anschluss der Verifikation zugeführt werden kann.

Eine ergänzende Möglichkeit zur Verifikation, die durch die vorgeschlagene Methode gegeben ist, ist die Überprüfung der Zwischenergebnisse jedes Programmablaufes. Er ist gegeben durch die schrittweise Entwicklung des Endzustandes der SPS. Alle Zwischenwerte eines Programmdurchlaufes können zur Auswertung zwischengespeichert werden. Diese Werte folgen eindeutig aus den Eingangs- und Ausgangsinformationen der Ein- Ausgabe-Variablen V (die Bedingungen für die Verifikationsschritte) und der Logik des SPS-Programms. Während ein Teil der Werte für die Variablen V direkt aus der Aufgabenstellung abgelesen und damit bestimmt werden kann, müssen die Zustandswerte, oben repräsentiert durch Konfigurationen, Schritt für Schritt bestimmt werden. Der dafür benötigte Aufwand ist groß und die zu untersuchenden Reaktionen der SPS müssen nach der Berechnung erst aus den ermittelten Lösungen extrahiert werden. Aus diesem Grund werden Möglichkeiten zum Einsatz dieser Methode nicht weiter untersucht.

[18] Aus den Konfigurationen lässt sich der Programmablauf des SPS-Programms zu jedem Zeitpunkt des Programmfortschrittes ablesen und nachvollziehen.

Ist die Problemstellung für ein SPS-Programm genau festgelegt, folgt aus der Aufgabenstellung unmittelbar, bei welcher Konstellation der Eingangsinformationen vorab bestimmte Ausgangsinformationen auszugeben sind. Ein Programm ist jedenfalls im Sinne einer Aufgabenstellung korrekt, wenn es genau diesen Ansprüchen genügt. Sollten etwa auf Grund von Defekten bei der Signalübermittlung an die SPS mit Fehlern behaftete Informationen übertragen werden, müssen Ausgangszustände erreicht werden, die Fehlreaktionen der SPS ausschließen.[19] Innerhalb des SPS-Programms wird dabei von so genannten Verriegelungen gesprochen, die durch die Logik des Programms zu definieren sind. Dazu werden unter anderem interne Variablen genutzt, deren Zustände vor der Verifikation bestimmt sein müssen.

Neben der Verifikation selbst und der Verringerung des dafür notwendigen Aufwandes liegt ein weiterer Schwerpunkt bei der Bestimmung von nicht direkt zur Verfügung stehender und nur innerhalb des Programms genutzter interner Variablen. Die dazu angewendete Methode ist die Definition von freien Variablen, die nach einem oder mehreren automatisierten Programmdurchläufen Werte annehmen, die bis zu deren Bestimmung nach Überprüfung durch den Anwender als bisher noch nicht berücksichtigte Vorbedingungen in die Verifikation mit einbezogen werden können.

1.3 Aufbau der Arbeit

Die Programmanalyse dient als Vorarbeit. Das zu verifizierende SPS-Programm wird analysiert, um es in Teilbereiche wie z.B. Funktionen oder abgrenzbare Funktionalitäten zu segmentieren. Damit wird erreicht, dass der Aufwand für die nachfolgende Verifikation so gering wie möglich gehalten wird. Die dazu verwendeten Werkzeuge sind auch als Hilfsprogramme für die Überprüfung der untersuchten SPS-Programme nutzbar. Bereits weitgehend strukturierte Programme werden aus dem AWL Code derart umgewandelt und aufbereitet, dass sie mit Hilfes der vorgestellten Methode schrittweise untersucht und verifiziert werden können.

Im Kapitel „SPS-Grundlagen“ wird der Aufbau, die Arbeitsweise und die Programmierung nur so weit vorgestellt, dass die zur Programmanalyse vorgestellten Untersuchungen, die nicht nur syntaktisch, sondern zum Teil auch semantisch erfolgen, nachvollziehbar werden. Für die theoretische Programmanalyse wird das SPS-Programm dazu in Programmdurchläufe und danach in Kombination von Programmdurchläufen (Zyklen) zerlegt. Dazu wird auch die

[19] Erreicht werden kann das beispielsweise durch mit in das Programm aufgenommene Sicherheitsfunktionen. Bei einer Produktionseinrichtung mit mehreren Arbeitsstationen wird z.B. gefordert, dass während des Transport eines Werkstücks von einer Station zu nächsten alle Werkzeuge in eine sichere Position zurückgezogen werden müssen, etwa um Kollisionen zu vermeiden. Fällt eine der Überwachungsfunktionen während des Transportvorgangs aus, kann durch eine programmierte Sicherheitsreaktion z.B. eine Not-Halt Funktion aktiviert werden.

Rückwärtsanalyse als Verifikationswerkzeug betrachtet.[20] Im Zuge einer erweiterten Programmanalyse werden die Aufgabenstellungen dazu genau spezifiziert und die Hilfsprogramme sowie ihre Nutzung für die folgende Verifikation beschrieben.

Ziel der Arbeit ist SPS-Programme dem in [KP12] vorgestellten Verifikationsalgorithmus zugänglich zu machen. Durch die Segmentierung sind es im Gegensatz zu den in der Literatur vorgestellten Konzepten auch umfangreichere SPS-Programme verifizierbar.

Die automatisierte Verifikation erfordert im Idealfall nur eine minimale Interaktion mit den Nutzern. In der Arbeit wird dargestellt, wie die Komplexität dadurch verringert wird, in dem der gesamte Verifikationsaufwand in kleinere Arbeitsschritte aufgeteilt wird, die jeder für sich und unabhängig voneinander durchgeführt werden. Ein weiterer Grund für die Aufteilung ist, dass die benötigte Rechnerkapazität nicht zu groß wird. Dazu wird ein zu untersuchendes SPS-Programm in einfach handhabbare Segmente zerlegt.

1.4 Begriffe

In der Folge werden die wichtigsten Begriffe zusammengefasst und erläutert, wie diese in dieser Arbeit zu verstehen sind.

Validierung

Mit Validierung eines SPS-Programms wird die Eignung der Software bezogen auf ihren Einsatzzweck in einer Anlage verstanden. Die Prüfung erfolgt auf Grundlage von Anforderungsprofilen in der Regel bei der Inbetriebnahme einer Anlage durch ausführliches Testen der spezifizierten Funktionen. Gefragt wird dabei nach der Effektivität der Entwicklung. Streng genommen müssen bei der Validierung auch in der Anforderungsspezifikation nicht ausdrücklich dargestellte oder spezifizierte Betriebszustände eines Systems mitberücksichtigt bzw. überprüft werden (Fragestellungen: „Ist das System richtig?“ oder „Ist es das richtige System?“ bezogen auf die Aufgabenstellung).

[20] Anmerkung: die klassische Vorgehensweise bei der Verifikation arbeitet entgegen der Richtung der Programmbearbeitung. Mit Hilfe einer ähnlichen Methode wird bei der Rückwärtsanalyse gefragt, welche Variablen ein Ergebnis bilden.

Sei a + b = c eine einfache Formel mit a, b und c als logische Variablen eines SPS-Programms.

Die Rückwärtsanalyse ergibt c ⟨a, b⟩ mit der Sprechweise „c ist abhängig von a und b“ und in weiterer Folge auch die Abhängigkeiten von a und b also c ⟨a ⟨und seinen Vorgängern⟩, b ⟨und seinen Vorgängern⟩⟩. Der Vorgang wird solange fortgesetzt, bis für alle logischen Variablen, die c beeinflussen, bestimmt sind. (Sind auch die Variablenzustände bekannt, kann der Zustand von c entsprechen den im SPS-Programm zu Grunde gelegten Befehlen bestimmt werden. Zweck der Rückwärtsanalyse ist jedoch hier nicht die Berechnung der Zustände, sondern dient der Bestimmung von Bereichsgrenzen im Zustandsraum.)

Verifikation

Formale Verifikation ist ein rein formaler Prozess, der logikbasiert für ein Programm oder ein System sicherstellt, ob das Programm oder System bezogen auf eine vorgegebene Spezifikation konform ist (Fragestellung: „Ist das System richtig konzipiert bzw. gebaut?“). Mit der Verifizierung wird der Nachweis erbracht, ob ein behaupteter Sachverhalt wahr ist. Bei der formalen Verifikation werden allgemeine formale Beweise geführt, wobei bei der in dieser Arbeit vorgeschlagenen Methode eine mögliche Interpretation der formalen Beweisführung mittels Prädikatenlogik mit einen SPS-Programm in Verbindung gebracht wird.

Eingangszustände, Eingangsabbild, Prozessabbild für Eingänge

Eingangszustände stellen das Abbild der bei einer SPS vorhandenen Eingänge dar. Diese sind bei der SPS in einem dafür reservierten Bereich des Speichers abgelegt. In der Folge wird vom Eingangsabbild gesprochen, das während eines Programmdurchlaufs, dem Zyklus, konstant gehalten wird. Die realen Eingänge, die dem Prozessfortschritt einer SPS-Anwendung folgend ihren Zustand immer wieder anpassen, werden im Eingangsabbild für die Dauer einer Bearbeitung, dem SPS-Zyklus, festgehalten. Ihre Zustände sind also temporär konstant. Das Eingangsabbild spiegelt das Prozessabbild der Eingänge wider.

Ausgangszustände, Ausgangsabbild, Prozessabbild für Ausgänge

Ausgangszustände sind Zustandswerte, die vom Programm der SPS berechnet worden sind. Am Ende einer Berechnung durch das SPS-Programm werden diese Werte über die Ausgangsschaltung ausgegeben und stehen so dem gesteuerten Prozess zur Verfügung. Ausgangszustände sind im Ausgangsabbild zusammengefasst. Darüber hinaus existieren Variable, die nur innerhalb der SPS genutzt werden. Die gewöhnlich als Merker bezeichneten Variablen sind nicht an die Ausgangsschaltungen geführt und stehen nur innerhalb des SPS-Programmfortschritts zur Verfügung. Auch für diese Zustandswerte sind bestimmte Speicherbereiche in der SPS reserviert. Im Gegensatz zu den Eingangszuständen verändern sich die Werte dieser Variablen im Prozessfortschritt. Das Ausgangsabbild spiegelt das Prozessabbild der Ausgänge wider.

Eingangsvektor, zyklische Berechnung, Ausgangsvektor

Betrachtet wird ein einzelner SPS-Zyklus: im Eingangsabbild festgehaltene Eingangszustände werden mit den Berechnungsergebnissen der Ausgangszustände des genau im vorhergehenden Zyklus berechneten Ausgangsabbildes zusammengesetzt und im Eingangsvektor zusammengefasst. Der Eingangsvektor stellt die Startbedingung für einen Zyklus dar, die Ergebnisse der Berechnung durch das SPS-Programm sind im Ausgangsvektor enthalten. Im Kapitel 2 wird die Arbeitsweise einer SPS genauer betrachtet.

Pfad, Konfiguration

In Anlehnung an die Sprechweise bei der Untersuchung von Programmen wird die sonst übliche Bezeichnung „Pfad“, der eine genaue Folge von Berechnungsschritten in Programmen bezeichnet, durch eine Kette aneinander gereihter Konfigurationen ersetzt. Der Begriff Pfad beschreibt Wege für die Berechnung, also Befehlsfolgen und deren Abhängigkeiten. Im Begriff Konfiguration wird der Berechnungspfad auf einen Index reduziert und zusätzlich der jeweilig aktuelle Variablenzustand berücksichtigt. Sind in einem SPS-Programm keine Verzweigungen vorhanden, dieser Umstand wird für die meisten SPS-Programme gelten, existiert nur ein einziger Pfad. Die später eingeführte Konfiguration betrachtet die Zustände an jeder Stelle dieses Pfades, die den Berechnungsfortschritt innerhalb eines SPS-Zyklus beschreiben.

Am einfachsten lässt sich jede Konfiguration als ein in jedem Programmschritt aktualisiertes Abbild des Variablen-Bereichs im Speicher der SPS beschreiben. Beginnend mit der Eingangskonfiguration wird die Konfiguration im Speicher Schritt für Schritt, also Befehl für Befehl, so lange weiterentwickelt, bis zuletzt die zugehörige Ausgangskonfiguration entsteht. Ein definierter Teil der so entstandenen Ausgangskonfiguration ist das Ausgangsabbild.

Werden ausschließlich binäre Signale innerhalb einer Konfiguration betrachtet, wird diese auch als binärer Vektor bezeichnet. Insbesondere im Werk „Schritte zur Verifikation von SPS-Programmen“ [KP12] wird bei der Verifikation von logischen Verknüpfungen innerhalb von SPS-Programmen von Ein- und Ausgangsvektoren gesprochen.

2 SPS Grundlagen

In diesem Abschnitt werden jene Besonderheiten von speicherprogrammierbaren Steuerungen vorgestellt, die später für die Analyse von Interesse sind. Grundlegendes zum Aufbau und zur Programmierung wird nur soweit detailliert, wie es für das Verständnis der Konzepte erforderlich ist.

2.1 Aufbau

Das Konzept der SPS wurde Ende der 60er Jahren in den USA entwickelt. Im Gegensatz zu den fest verdrahteten Regelungs- und Steuerungsanlagen sollten SPS universell einsetzbar sein. Daher wurde bei der Konzeption auf geringen Aufwand bei der Programmierung und bei der Verdrahtung Wert gelegt. Ein weiteres Merkmal einer SPS im Vergleich zu traditionellen Anlagen war der verschleißfreie Betrieb, da elektromechanische Schaltgeräte durch Elektronik ersetzt worden sind.

Eine SPS gleicht im Aufbau einer traditionellen Steuerungsanlage. Sie kann grob in die Ebenen Eingangsbereich für Eingabedaten, CPU (*central processing unit*) für die Datenverarbeitung und Ausgangsbereich für die Ausgabe der berechneten Verknüpfungsergebnisse gegliedert werden.

Das Prozessabbild der Eingänge und das Prozessabbild der Ausgänge stellen jene Speicherbereiche dar, in die vor jedem Prozess-Schritt bzw. an seinem Ende die Werte der Eingänge und Ausgänge kopiert werden. Die CPU ist in verschiedene Funktionseinheiten unterteilt. Das Rechenwerk führt die Operationen durch, das Steuerwerk entschlüsselt die Befehle und koordiniert den Datentransport. Der Speicher ist in Teilbereiche unterteilt, einerseits für die Aufnahme des Betriebssystems, andererseits für das Anwenderprogramm und Symboltabellen. Dazu gehören auch die Bereiche für Prozessabbilder, ein Bereich für die Zwischenspeicherung von Verknüpfungsergebnissen und Speicherbereiche für spezielle Programmfunktionen elementarer Datentypen.

2.2 Arbeitsweise

Die Arbeitsweise einer SPS folgt dem herkömmlichen EVA-Prinzip (**E**ingabe- **V**erarbeitung-**A**usgabe). Die Verarbeitung der Anweisungen erfolgt in einer SPS seriell. Schritt für Schritt wird eine komplexe Verknüpfung mit mehreren Parametern vollzogen. Fig. 2 zeigt einen Ausschnitt eines SPS Programms, links in Kontaktplandarstellung (KOP), rechts in Anweisungsliste (AWL). Die Reihenfolge der Befehlsbearbeitung, gleichzusetzen mit dem Bearbeitungs-

pfad durch das Programm, ist in der KOP-Darstellung durch Pfeile angedeutet, in der AWL-Darstellung zeilenweise von oben nach unten unmittelbar gegeben.

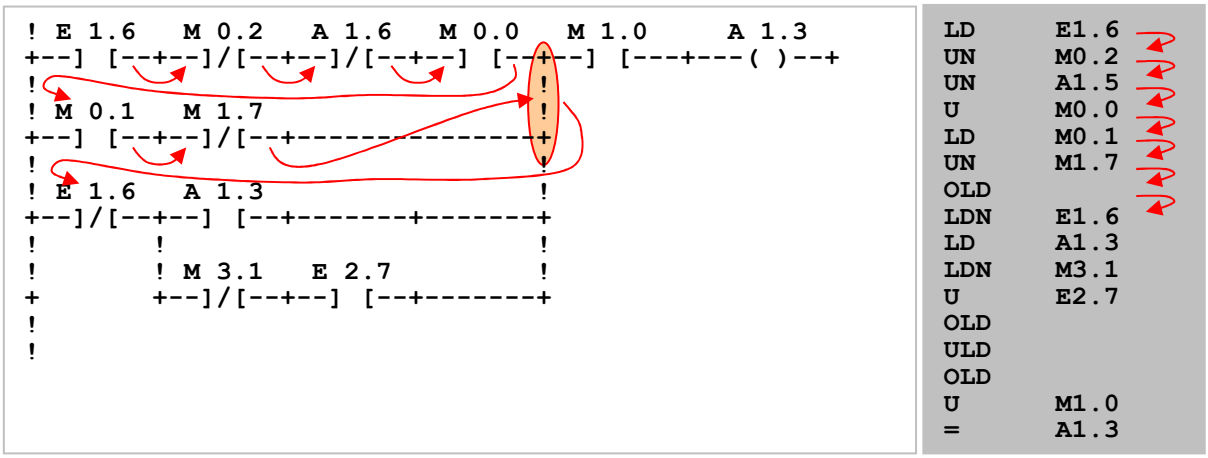


Fig. 2: SPS-Programm, links KOP, rechts AWL (**STEP 7**)

Die hohe Geschwindigkeit, mit der die einzelnen Verknüpfungen durchgeführt werden, lässt die Verarbeitung parallel erscheinen, wie diese bei einer fest verdrahteten Steuerung erfolgt. Berücksichtigt man die für elektromechanische Schaltvorgänge typischen Anzugs- und Abfallverzögerungen, ist die SPS in ihrem Reaktionsverhalten deutlich schneller.

Die Anweisungen im Programm legen fest, welche Eingangssignale für eine Verknüpfung verwendet werden, wie sie miteinander verknüpft sind und wo die Ergebnisse als Ausgangssignale abgelegt werden. Die zugehörigen Daten bezieht die CPU nur aus dem Speicher. Daher werden am Anfang jedes Programmzyklus alle Eingangszustände als Signalzustände von den Eingangs-Peripheriebausteinen in den Datenspeicher in den Bereich für das Prozessabbild der Eingangsebene kopiert. Zusammen mit den Zuständen der Ausgänge und den internen Variablen, diese stammen aus bis zu diesem Zeitpunkt durchgeführten Berechnungsschritten, bilden sie die Startbedingung für die Berechnungen des folgenden Programmdurchlaufes. Die in den Anweisungen kodierten Verknüpfungen werden seriell durchgeführt. Die Verknüpfungsergebnisse werden von der CPU im Speicherbereich für das Prozessabbild der Ausgangsebene abgespeichert und am Ende des Zyklus die gespeicherten Statuszustände in die Ausgangs-Peripheriebausteine kopiert. Neben den Ausgängen können Verknüpfungsergebnisse auch durch interne Variable zwischengespeichert werden. Interne Variable werden im Datenspeicher ähnlich den Ausgängen genutzt, besitzen jedoch keine Verbindung zu physikalischen Ausgängen. Danach folgt der nächste Zyklus in gleicher Weise.

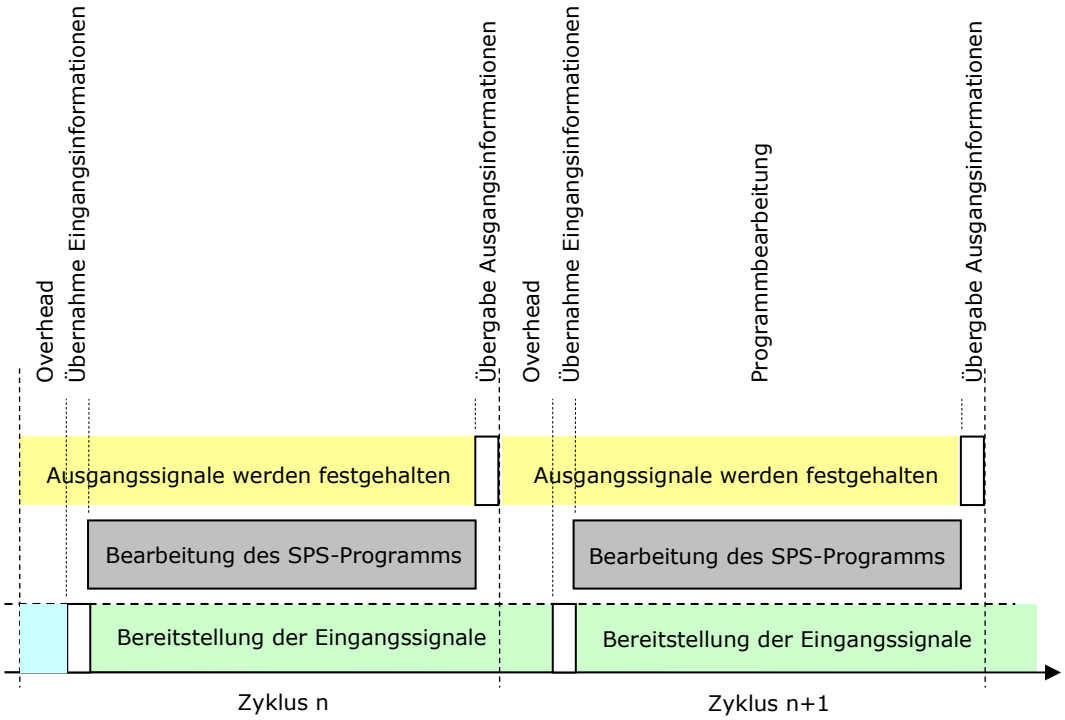


Fig. 3: Zyklische SPS-Programmbearbeitung

Fig. 3 zeigt den zeitlichen Ablauf von SPS-Zyklen. Der Arbeitsfortschritt ist im Bild von links nach rechts dargestellt. Unterschieden sind die Bereiche für Eingangssignale, SPS-Programm und Ausgangssignale. Diesen Bereichen sind in der SPS bestimmte Speicherbereiche zugeteilt. Festzuhalten ist, dass die über die Eingänge einer SPS zugeführte Information dem Prozessverlauf folgend zu beliebigen Zeitpunkten sich verändern kann, dass es aber genau festgelegte Zeitfenster in der Bearbeitung gibt, an denen diese Informationen für die weitere Bearbeitung übernommen werden. Ausgenommen davon ist die Funktionalität der eher selten verwendeten „immediaten“ Befehle.[21]

21 Bei den als „immediate“ Befehle bezeichneten Operationen kann der Zustand beispielsweise eines Eingangs direkt auch asynchron zum Bearbeitungszyklus von den Anschlussklemmen des betreffenden Eingangs abgelesen werden.

Die Problematik bei der Untersuchung derartiger Programmkonstrukte liegt darin, dass unter der Annahme, ein Zustand des betrachteten Eingangs verändert sich im Bearbeitungszyklus, wie folgt:

Im günstigen, d.h. unproblematischen Fall gilt, dass der veränderte Zustand für alle nachfolgenden Abfragen nicht relevant ist, es gibt daher keine Auswirkungen auf das Untersuchungsergebnis. Im ungünstigen Fall muss jedoch mit unterschiedlichen Zuständen für einen Teilbereich in der Programmberechnung eines gerade untersuchten Bearbeitungszyklus gerechnet werden. Als Folge davon wächst die Anzahl der notwendigen Untersuchungen und die Komplexität steigt, weil dann in der Regel auch Kombinationen von Programmdurchläufen für eine erfolgreiche Verifikation zu untersuchen sind.

Die Bereitstellung der Eingangssignale an die Anschlussklemmen erfolgt kontinuierlich. Zu genau vorbestimmten Zeitpunkten werden die darin enthaltenen Informationen in den Speicherbereich der Eingangsinformation übernommen und für die Programmbearbeitung bereitgestellt.

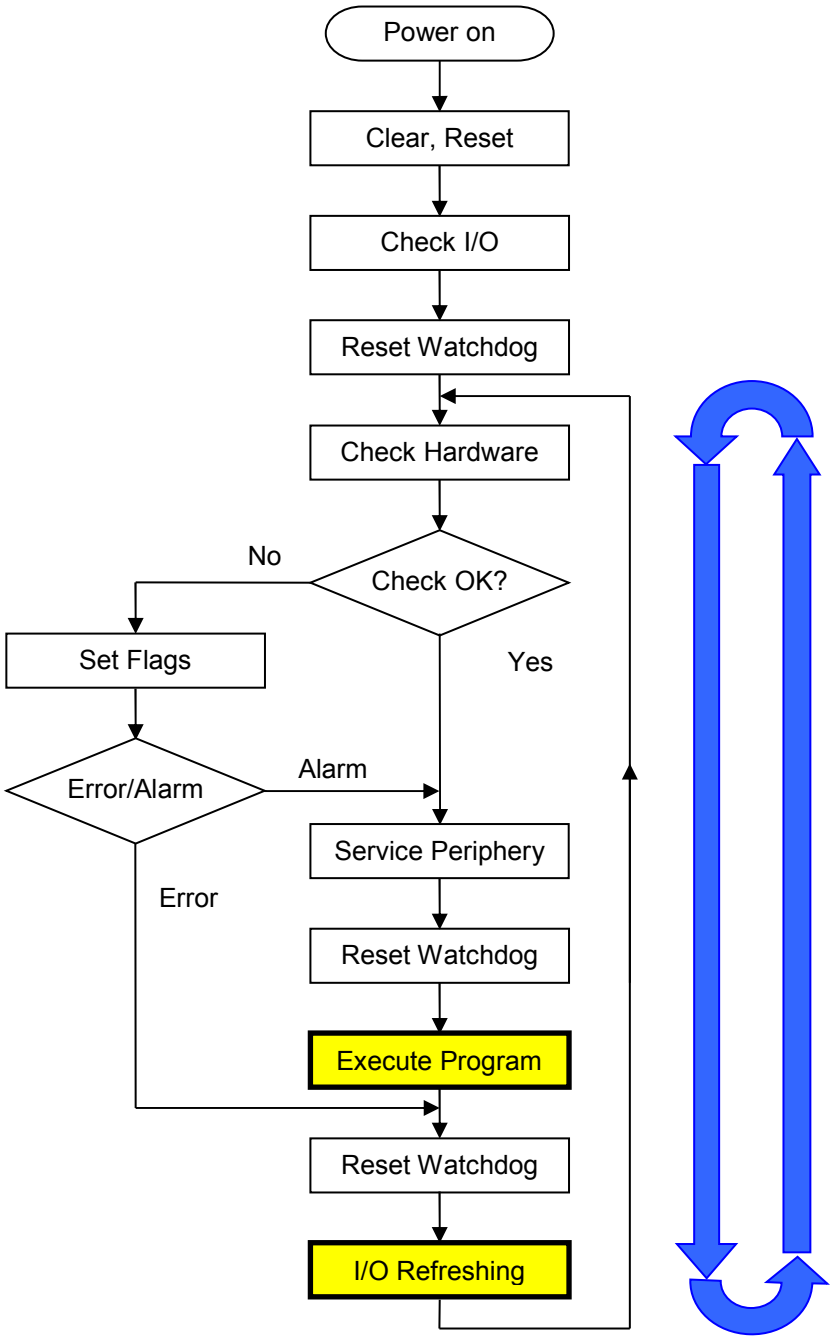


Fig. 4: SPS-Zyklus mit Überwachungsfunktionen

Zeitlich davor liegt noch ein Zeitfenster, in dem allgemeine Bearbeitungen durch das Betriebssystem der SPS vorgenommen werden. Fig. 4 zeigt in einer groben Übersicht, wie in

den SPS-Zyklus Überwachungsfunktionen eingebunden sein können. In dieser Arbeit interessieren die hervorgehobenen Blöcke (I/O Refreshing, und Execute Program).[22]

Zu den Überwachungsfunktionen zählen unter anderem die Überwachung der Zykluszeit (Watch Dog), Behandlung von als remanent definierten Speicherbereichen bei Neuanlauf und die Erzeugung interner Taktsignale. Eine weitere wichtige Funktion im Programm-Overhead ist die Überwachung der Zeitpunkte für die Übergabe und Übernahme von Eingangs- bzw. Ausgangszuständen und der Start der Bearbeitung des SPS-Programms. Bei nahezu allen SPS-Steuerungen werden beim Feststellen eines Fehlers durch die Kontrollmechanismen alle Ausgänge zurückgesetzt.

Der Fokus in dieser Arbeit liegt bei der Programmbearbeitung, die nach der Übernahme der Eingangsinformation die genau im betrachteten Zyklus zugehörige Ausgangsinformation erzeugt und in der Veränderung der zugehörigen Speicherinhalte während des betrachteten Bearbeitungsschritts. Details dazu werden in den folgenden Kapiteln vorgestellt.

2.3 Programmierung

SPS-Systeme können auf unterschiedliche Weise programmiert werden. Die in der SPS-Programmierung angewendeten Programmiersprachen sind in der IEC 61131-3 definiert. Es existieren graphische und textuelle Sprachkonzepte, die teilweise ineinander übergeführt werden können.

Eines der ältesten Sprachkonzepte ist die Kontaktplandarstellung. Diese Sprache wurde konzipiert, damit mit traditionellen Steuerungsanlagen vertraute Techniker eine Darstellungsart für SPS-Programme vorfinden, die einem Stromlaufplan einer herkömmlichen Schütz/Relais Steuerungsanlage gleicht. Die Funktionsplandarstellung nutzt die Darstellungsart logischer Funktionen mittels boolescher Algebra. Die Anweisungsliste (AWL) ist die Darstellung eines SPS-Programms ähnlich der Notation einer Assemblersprache. Insbesondere ist die maschinennahe Darstellung des Codes in AWL der Grund dafür, dass nahezu alle anderen Sprach-

[22] Anzumerken ist, dass die Art wie eine SPS initialisiert wird und wie diverse Funktionalitäten in den zyklischen Ablauf eingebunden sind, einerseits vom Hersteller und anderseits auch von der betrachteten Steuerungsfamilie abhängt.

Daher ist der Ablauf in gezeigten Modell

Schritt 1: Programmbearbeitung,

Schritt 2a: Bearbeiten der Ausgänge und

Schritt 2b: Bearbeiten der Eingänge.

Wählt man die Zählweise Schritt 2b, Schritt 1, Schritt 2a ist damit der ursprünglich diskutierte EVA Zyklus wieder hergestellt.

konzepte auch in AWL dargestellt werden können. Vielfach genügt dazu ein einfaches Umschalten der Darstellungsart im Programm Editor.

2.4 Anweisungen und Speicherbereiche

Eine SPS Anweisung entspricht einem Befehl einer Programmiersprache, der von der CPU in einem Prozessschritt ausgeführt wird. In der Folge wird die Notation an die Schreibweise der Anweisungsliste angelehnt.[23] Die Struktur einer SPS Anweisung besteht aus der Bezeichnung der Operation und den zu verwendenden Operanden. Typische Operationen sind Abfragen [**L**, **LN**, …], Verknüpfungen [**U**, **UN**, **O**, **ON**, …] und Zuweisungen [**=**, **S**, **R**, …].[24] Bei den Operanden werden Kennzeichen zum Identifizieren der Speicherbereiche verwendet (für Eingänge [**E**], Ausgänge [**A**], Merker [**M**], usw.) und Parameter für die Adresse innerhalb des Speicherbereichs.

Der Speicherbereich ist, wie eingangs erwähnt, gegliedert in Bereiche für das Prozessabbild der Eingänge [**E**], das Prozessabbild der Ausgänge [**A**] sowie in Bereiche für die Abspeicherung von Zwischenergebnissen (Merker [**M**], Zeiten [**T**] und Zähler [**Z**]). Über die Prozessabbilder werden die Zustandsänderungen an den angeschlossenen Eingängen erfasst und die Reaktion der SPS an die Ausgänge weitergeleitet.

Bei einer Abfrage wird der Zustand eines Speicherbereichs temporär in das so genannte Verknüpfungsergebnis (VKE) übertragen (geladen). Jede der folgenden Verknüpfungsoperationen manipuliert dieses Ergebnis entsprechend den zugrundeliegenden Berechnungsvorschriften und weist das Ergebnis der temporären Variablen VKE zu. In der Zuweisungsoperation wird der Wert des zuletzt berechneten VKE an den Ausgang im Ausgangsabbild, den Merker im Merkerbereich oder Variable im Bereich für Zwischenergebnisse zugewiesen. Die dort gespeicherten Zustände sind solange gültig, bis sie wieder verändert werden.

Die Zuweisung an den gleichen Speicherbereich kann innerhalb eines SPS-Zyklus mehrmals erfolgen. Im Sinne einer syntaktischen Überprüfung liegt hier in der Regel kein Programmfehler vor. Fehlt die Abfrage in einer Bearbeitungskette oder die Zuweisung, wird dies von den Programmeditoren in der Regel als Fehler erkannt.

---

[23] Trotz der Normung in der IEC 61131-3 gibt es bei der Schreibweise in AWL geringfügige Unterschiede, abhängig von der jeweils betrachteten Steuerungsfamilie.

[24] Die verschiedenen Schreibweisen sind „Dialekte“ von der in der IEC 61131-3 festgelegten Bezeichnungen. Es sind z.B. für den Befehl „Lade“ die Bezeichnungen **L**, **LD**, oder **STR** (string) je nach Hersteller der Steuerung und Programmiersoftware bekannt. Es gibt auch Programmierkonzepte, die ohne den „Lade“ Befehl auskommen. Ähnlich ist es mit der Behandlung von Klammerbefehlen. Anstelle von **U(**, **O(** und **)** wird z.B. **ULD** oder **OLD** verwendet.

3 Gliederungsstrukturen von SPS-Programmen

Ein SPS-Programm ist definiert durch die Folge seiner Kommandos. Die Kommandos werden in der Reihenfolge ihrer Anordnung im SPS-Programm der Reihe nach bearbeitet. Es entsteht ein Pfad, der in jedem Programmzyklus einmal zur Gänze durchlaufen wird. Bezogen auf das vollständige SPS-Programm gilt, dass dieser Pfad aus der Folge der SPS-Kommandos gebildet wird. Jedes Kommando bewirkt eine Veränderung im Zustandsabbild, das ausgehend von einem initialen Zustand am Ende des Pfades in einen Endzustand weiterentwickelt wird.

3.1 Zustandsübergänge

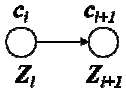


Fig. 5: Übergang 1

Ausgehend vom jedem Zustand Z_i erzeugt das folgende Kommando c_{i+1} den zugehörigen Folgezustand Z_{i+1}. Das bedeutet, dass das Abbild der Zustände durch die Wirkung der Kommandos verändert werden kann (Fig. 5).

Nicht alle Kommandos sind in der Lage, Zustandsänderungen in den Abbildern zu bewirken. Die Wirkungsweise von Kommandos wird weiter unten noch genauer betrachtet. Die Struktur eines SPS-Programms ist in der Fig. 6 als Pfad einer Kette von Kommandos mit ihren zugehörigen Zustandsübergängen dargestellt. Kommandofolgen können in Abschnitte unterteilt werden. Für solche Abschnitte hat sich der Begriff „Netzwerk“ eingebürgert.

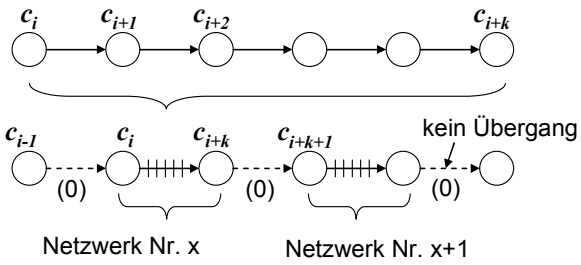


Fig. 6: Kommandofolge

Im Teilbild Fig. 6 oben ist die Kommandofolge c_i bis c_{i+k} als Teil des Gesamtpfades aller Zustandsübergänge des SPS-Programms herausgehoben. Im Netzwerk mit der laufenden Nummer x existieren in Summe k Kommandos, wobei jedes Kommando eine Zustandsänderung bewirkt.

Definitionsgemäß beginnt für SPS-Programme jedes Netzwerk mit einer Abfrage, besitzt eine Anzahl von Verknüpfungen und endet mit einer Zuweisung. Nachdem das darauffolgende Netzwerk ebenfalls mit einer Abfrage beginnt, findet genau genommen zwischen den einzelnen Netzwerken kein unmittelbarer Zustandsübergang statt.

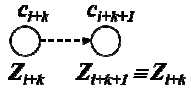


Fig. 7: Übergang 2

Ein Zwischenzustand Z_i erzeugt duch das folgende Kommando c_{i+1} den zugehörigen Folgezustand Z_{i+1}. Aufeinander folgende Zustände könnnen identisch sein, solange nur lesende Zugriffe erfolgen (Fig. 7).

Beim Übergang von einem Netzwerk zum folgenden Netzwerk bedeutet das für den Zwischenzustand Z_{i+k}, dass das folgende Kommando c_{i+k+1} keine Zustandsäderung bewirkt und dass damit der Folgezustand Z_{i+k+1} unverändert bleibt und damit auch identisch mit dem vorherigen Zwischenzustand Z_{i+k} ist. Es gilt immer, dass der initiale Zustand eines folgenden Netzwerks identisch sein muss mit dem Endzustand seines Vorgängers. Dieser Umstand besitzt deshalb Bedeutung, weil dadurch funktionelle Bereiche eines SPS-Programms voneinander vergleichsweise einfach abgegrenzt werden können und die Teilbarkeit in so genannte „funktionelle“ Abschnitte vereinfacht wird,

Dieser Umstand ist unmittelbar einleuchtend. Er hängt von der Funktionalität der Kommandos ab: ausschließlich jene Kommandos, die in das Abbild der Zustände schreibend gleichzusetzen mit verändernd eingreifen, können Aktionen des Steuerungssystems auslösen.

Bezogen auf die erste Definition eines vollständigen SPS-Programm als Folge seiner Kommandos kann die Beschreibung erweitert werden: ein SPS-Programm wird aus einer Folge von Pfadfragmenten beschrieben. Ein Netzwerk stellt ein solches Fragment innerhalb des Pfades eines SPS-Programms dar. Für jedes Netzwerk können ein Anfangszustand und ein Endzustand festgelegt werden. Auf Grund dieser Eigenschaft kann ein Netzwerk auch für sich alleine untersucht werden.

Die Interaktion über die Grenzen von Netzwerken hinaus erfolgt über die Abbilder der Zustände. Die jeweils aktuellen Zustände sind in dafür vorreservierten Speicherbereichen der SPS abgelegt. Bei jeder Abfrage oder Verknüpfung wird aus diesen Speicherbereichen die gerade zum Abfragezeitpunkt gültige Zustandsinformation abgeholt und verarbeitet. Zwischenergebnisse werden in der jedem Kommando zugeordneten Konfiguration gespeichert. Die Bearbeitung von Zuweisungen erfolgt schreibend, d.h. dort werden die Ergebnisse der Berechnungsfolge des Pfad-Fragmentes „Netzwerk“ im zugehörigen Ergebnisspeicher als aktualisierte Abbilder festgehalten und stehen für die nachfolgenden Programmteile zur Verfügung.

3.2 Programmverzweigungen

Nebenläufige Pfade (Fig. 8) sind in einer Programmstruktur von SPS-Programmen dann notwendig, wenn unterschiedliche Funktionalitäten einer Maschine gefordert werden. Es können z.B. Bearbeitungsschritte abhängig von einer Programmvorwahl in einer unterschiedlichen Reihenfolge abzuarbeiten sein. Ein sehr einfaches Beispiel ist die wahlweise zusätzliche Bewegung eines Werkzeuges bei einer durch eine SPS gesteuerten Maschine, die bei einem anderen Fertigungsverfahren nicht notwendig ist. In einem solchen Fall gibt es unterschiedliche

Kommandofolgen c_i bis c_{i+k} für den Herstellungsprozess nach Variante A (Pfad oben) bzw. c_j bis c_{j+m} für den Herstellungsprozess nach Variante B (Pfad unten).

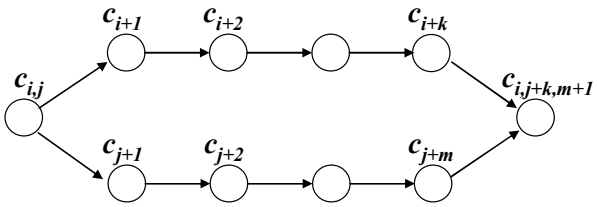


Fig. 8: Nebenläufige Abläufe

Nebenläufigkeit kann in der SPS nur durch Auslassungen bestimmter Programmteile, im untersuchten Fall durch Programmpfade, bewerkstelligt werden.

Solche Programmpfade werden jedoch immer nacheinander in die Kommandofolgen ähnlich Fig. 8 im SPS-Programm implementiert, da SPS Programme keine Parallelverarbeitung zulassen.

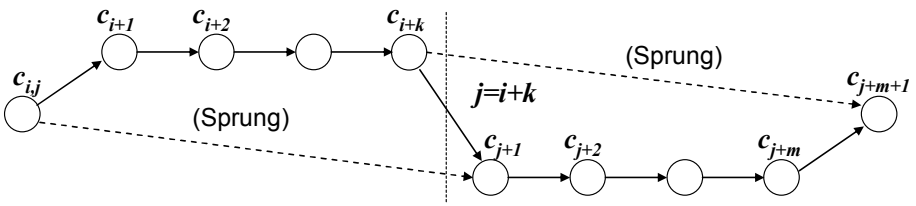


Fig. 9: Nebenläufige Abläufe sequenziert

In Fig. 9 erfolgt die Unterscheidung von gewünschten Programmfolgen durch das Einführen von Sprungbefehlen. Sprungbefehle zu nutzen ist eine Möglichkeit aber keine Bedingung. Es genügt auch, die Ansteuerung von Ausgängen der SPS programmtechnisch durch so genannte Verriegelungen derart zu verhindern, dass bestimmte Kommandofolgen keine Auswirkungen auf Zustände der Ausgangsinformation haben, Solche Kommandofolgen werden zwar im SPS-Programm bearbeitet, bewirken aber keine Veränderungen von Zuständen in den Ausgangsabbildern. Vielfach wird jedoch an Stelle von Sprüngen einfach eine Teilfolge von Kommandos derart programmiert, dass diese keine Auswirkungen auf das Ergebnis der Berechnung haben.

Das Zusammenfassen von Kommandos in Kommandofolgen zeigt bereits, dass für SPS-Programme innere Gliederungsstrukturen existieren, die voneinander unabhängig betrachtet werden können. Nicht nur die einzelnen Kommandos selbst, sondern auch Kommandofolgen sind logische Ausdrücke bzw. Formeln, die für sich gesehen bereits überprüfbare Einheiten darstellen.

3.3 Stufenmodell für die Programmüberprüfung

Die Art der Programmüberprüfung von SPS-Programmen kann als Stufenmodell betrachtet werden. In der niedrigsten Stufe kann untersucht werden, ob ein einzelnes Kommando seine Funktion erfüllt. Die Überprüfung einzelner Kommandos ist trivial, weil für jedes Kommando

seine Funktion durch eine Definition gegeben ist. Die nächst höhere Stufe betrachtet bereits eine Reihe von Kommandos, die einer Formel (z.B. in Form eines Netzwerkes) entsprechen. Bestimmt man zu Netzwerken die zugehörige Wahrheitstabelle, ist zu erkennen, dass die Überprüfung solcher Funktionsausdrücke ebenfalls noch als einfach zu betrachten ist. Im Ergebnis ist die Aussagekraft für einzelne Netzwerke aus dem Programmzusammenhang gerissen und damit als gering einzuordnen.

Interessant ist die Programmüberprüfung von Funktionsgruppen, die in der Regel bereits aus einer größeren Anzahl von Netzwerken besteht. Die Komplexität der Programmüberprüfung steigt durch die wachsende Anzahl der zu betrachtenden Signalzustände. Die praktische Bedeutung ist, dass funktionale Eigenschaften z.B. eines Maschinenablaufs überprüfbar werden. Diese Stufe kommt der Teilverifikation eines SPS-Programms bereits sehr nahe.

Bei einer Verifikation wird das gesamte Programm untersucht. Erst in dieser Stufe wird die Funktionalität einer durch die SPS gesteuerten Anlage oder Maschine durch die Verifikation bestätigt. Hier ist allerdings die Komplexität bereits derart hoch, dass die Überblickbarkeit der Funktionalität eines SPS-Programms als Gesamtheit wesentlich erschwert bis nicht mehr möglich einzuordnen ist.

Generell ist dazu festzustellen, dass bei der Inbetriebnahme von SPS-Steuerungssystemen die Gesamtfunktion auch stufenweise überprüft wird. In der niedrigsten Stufe werden zuerst alle Ein- bzw. Ausgänge auf Funktion und deren korrekte Zuordnung im SPS-Programm geprüft.

Im nächsten Schritt erfolgt die Überprüfung von Einzelfunktionen durch das Ansteuern einzelner Bewegungen. Diese Überprüfung wird soweit gesteigert, bis die Funktionalität der programmierten Funktionsgruppen sichergestellt ist. Ab diesem Zeitpunkt ist die Teilinbetriebnahme der gesteuerten Anlage oder Maschine vollzogen.

Erst zuletzt erfolgt die Überprüfung des Gesamtprogramms, wobei die Abhängigkeiten der vorher getesteten Funktionsgruppen mit einbezogen werden. In der Regel fällt dieser Arbeitsschritt bereits mit der Endabnahme durch den Nutzer der Anlage oder Maschine zusammen.

Die oben beschriebenen Stufen der Programmüberprüfung weisen eine Analogie zur realen Vorgehensweise einer Inbetriebnahme auf.

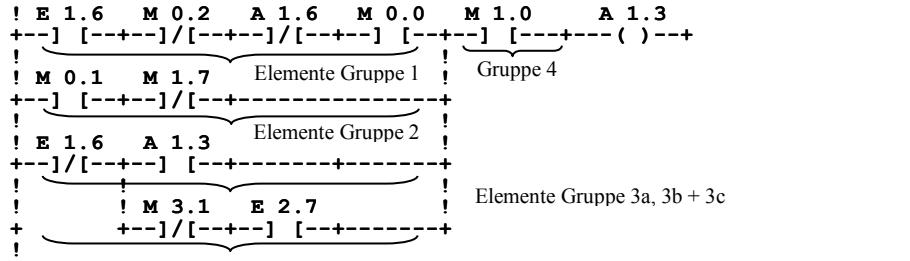
Stufe 1: Netzwerke und ihre Darstellungsarten

In der Folge wird ein und dasselbe Netzwerk eines SPS-Programms in unterschiedlichen Darstellungsarten betrachtet und weiter untersucht. Herangezogen werden die Programmierarten

Kontaktplan (KOP), Anweisungsliste (AWL) und Funktionsplan (FUP). Gefragt wird, wie ein solches Netzwerk in ein Flussdiagramm umgewandelt werden kann.

Kontaktplan

Einleitend wird in diesem Abschnitt ein Netzwerk in der Kontaktplandarstellung betrachtet (Fig. 10). Die Funktion solcher Netzwerke kann z.B. in der Form von Wahrheitstabellen dargestellt werden. Dazu werden die möglichen Zustände der ansteuernden Elemente und deren Wirkung untersucht.



Legende zur Kontaktplandarstellung: `-] [-` SCHLIESSER, `-]/[-` ÖFFNER, `-( )-` SPULE

Fig. 10: Beispiel Netzwerk 1 (KOP)

Die Fragestellung zur Untersuchung kann z.B. lauten: Bei welchen Zustandskombinationen wird der Ausgang eingeschaltet? Gesucht werden damit jene Kombinationen von Zuständen, die für den Ausgang **A1.3** logisch EIN für den Zustand „eingeschaltet“ ergeben. In der tabellarischen Aufstellung ist die Verwendung der Elemente (invertierend bzw. nicht invertierend) berücksichtigt und jene Kombinationen, die den logischen EIN Zustand des untersuchten Ausgangs bewirken, farbig hervorgehoben (Tab. 1). Das Element **M0.2**, verwendet in der Schaltung als Öffner, muss z.B. ausgeschaltet sein um bei der Verarbeitung den Zustand EIN einzunehmen. Die mit X bezeichneten Zustände sind nicht relevant und können beliebige logische Werte einnehmen.

E1.6	M0.2	A1.6	M0.0	M0.1	M1.7	E1.6	A1.3	E1.6	M3.1	E2.7	M1.0	A1.3
(1)				(2)		(3a)		(3b+c)			(4)	OUT
1	0	0	1	X	X	1	1	1	X	X	1	1
X	X	X	X	1	0	X	1	X	X	X	1	1
0	X	X	X	X	X	0	1	0	X	X	1	1
0	X	X	X	X	X	0	1	0	0	1	1	1

Legende: Schaltstellung der Elemente `EIN = 1`, `AUS = 0`, `EIN` oder `AUS = X`

Tab. 1: Wahrheitstabelle Ausgang logisch EIN für Netzwerk 1

Steuerungstechnisch wird im Zweig **(3a)** der Ausgang **A1.3** verwendet um eine so genannte „Selbsthaltung“ zu realisieren. Das bedeutet, dass wenn die Ansteuerung des Ausgangs

erfolgt ist, dieser so lange eingeschaltet bleibt, solange die mit **(3a)** und **(4)** gekennzeichneten Zustände EIN erhalten bleiben.

Wenn gefragt wird, bei welchen Zustandskombinationen der Ausgang ausgeschaltet ist, wird ähnlich vorgegangen. Gesucht sind jene Kombinationen von Zuständen, die für den Ausgang **A1.3** logisch **NULL** für den Zustand „ausgeschaltet“ ergeben (Tab. 2). Auch in dieser tabellarischen Aufstellung ist die Verwendung der Elemente (invertierend bzw. nicht invertierend) berücksichtigt und es sind jene Elemente farbig hervorgehoben, die den logischen **NULL** Zustand des untersuchten Ausgangs bewirken.

E1.6	M0.2	A1.6	M0.0	M0.1	M1.7	E1.6	A1.3	E1.6	M3.1	E2.7	M1.0	A1.3
(1)				(2)		(3a)		(3b+c)			(4)	OUT
x	x	x	x	x	x	x	0	x	x	x	0	0
0	x	x	x	0	x	0	0	0	1	x	1	0
0	1	x	x	0	x	0	0	0	1	x	1	0
0	x	1	x	0	x	0	0	0	1	x	1	0
0	x	x	0	0	x	0	0	0	1	x	1	0
0	x	x	x	x	1	0	0	0	1	x	1	0
0	1	x	x	x	1	0	0	0	1	x	1	0
0	x	1	x	x	1	0	0	0	1	x	1	0
0	x	x	0	x	1	0	0	0	1	x	1	0
0	x	x	x	0	x	0	0	0	x	0	1	0
0	1	x	x	0	x	0	0	0	x	0	1	0
0	x	1	x	0	x	0	0	0	x	0	1	0
0	x	x	0	0	x	0	0	0	x	0	1	0
0	x	x	x	x	1	0	0	0	x	0	1	0
0	1	x	x	x	1	0	0	0	x	0	1	0
0	x	1	x	x	1	0	0	0	x	0	1	0
0	x	x	0	x	1	0	0	0	x	0	1	0
1	1	x	x	0	x	1	0	1	x	x	1	0
1	x	1	x	0	x	1	0	1	x	x	1	0
1	x	x	0	0	x	1	0	1	x	x	1	0
1	1	x	x	x	1	1	0	1	x	x	1	0
1	x	1	x	x	1	1	0	1	x	x	1	0
1	x	x	0	x	1	1	0	1	x	x	1	0
1	1	x	x	0	x	1	0	1	x	x	1	0
1	x	1	x	0	x	1	0	1	x	x	1	0
1	x	x	0	0	x	1	0	1	x	x	1	0
1	1	x	x	x	1	1	0	1	x	x	1	0
1	x	1	x	x	1	1	0	1	x	x	1	0
1	x	x	0	x	1	1	0	1	x	x	1	0

Legende: Schaltstellung der Elemente `EIN = 1`, `AUS = 0`, `EIN` oder `AUS = X`

Tab. 2: Wahrheitstabelle Ausgang logisch NULL für Netzwerk 1

Der Zustand logisch **NULL** für den Ausgang **A1.3** wird immer dann gegeben sein, wenn völlig unabhängig von den übrigen Elementen im Netzwerk der Zustand vom Merker **M1.0** logisch **NULL** ist. Ist der Zustand vom Merker **M1.0** aber logisch **EINS**, müssen die übrigen Elemente in die Untersuchung mit einbezogen werden. Aus der Vorgabe folgt, dass **A1.3** im Zustand logisch **NULL** sein soll. Wenn der Eingang **E1.6** sich im Zustand EIN befindet (Gruppe 3) sind die Zustände der übrigen Elemente von Gruppe 3 nicht relevant.

Reihenfolge in der Bearbeitung der Kommandos

Wahrheitstabellen betrachten Zustandskombinationen parallel. In einem SPS-Programm werden Zustände jedoch seriell abgefragt. Aus diesem Grund ist ausgehend von einem gegebenen SPS-Programm die Reihenfolge der Bearbeitung der SPS-Kommandos von Bedeutung. Damit ergibt sich die Notwendigkeit, zusammengehörige Elemente einer Gruppe besonders zu kennzeichnen. Dazu wurden in der textuellen Darstellungsart Einklammerungen, auch als „Klammerebenen“ bezeichnet, eingeführt.

Grundsätzlich gilt, dass in den meisten Programmeditoren für SPS-Programme das Umschalten zwischen den Darstellungsarten für SPS-Programme möglich ist. Die Kontaktplanprogrammierung wurde ursprünglich deshalb eingeführt, um insbesondere den Steuerungstechnikern jene Art der Darstellung anzubieten, die den Steuerungs- (Schalt-) Plänen so stark ähnelt, dass Steuerungstechniker eine ihnen wohlbekannte und damit gewohnte Arbeitsumgebung vorfinden. Die Reihenfolge der Kommandobearbeitung erfolgt entsprechend der Gruppen und dargestellten Nummerierung im Beispielnetzwerk.

Anweisungslisten

In der Darstellungsart Anweisungsliste wird die Reihenfolge der Kommandos zeilenweise dargestellt. In der folgenden Gegenüberstellung sind einige herstellerspezifische Unterschiede von AWL herausgehoben. In der ersten Spalte „Klammern“ sind generell Einklammerungen vorgesehen. Diese Klammerebenen sind nummeriert. Diese Art der Darstellung ist vergleichsweise übersichtlicher als z.B. die Darstellungsart in der Programmiersprache STEP 7 (200er Serie) des Herstellers Siemens, in denen auf Klammern teilweise verzichtet worden ist.

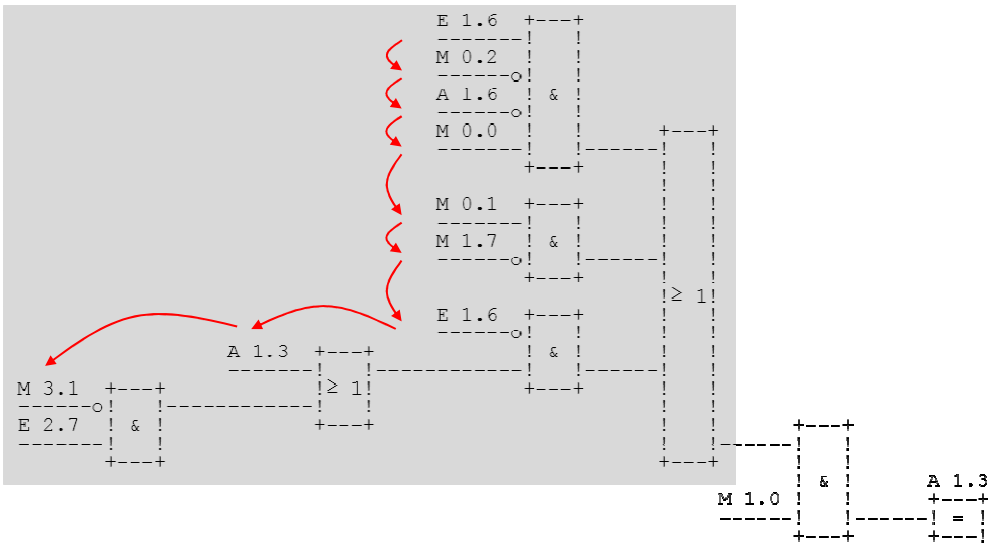
Selbst die Programmiersprache STEP 7 kennt unterschiedliche Varianten, bei denen für Klammerebenen spezielle Kommandos eingeführt worden sind. Beispielsweise ist das allein stehende Kommando „ODER“ in der Variante STEP 7 für Steuerungen der 300er-Serie als Klammerbefehl ähnlich dem Kommando „**O (**“ zu interpretieren. Die gedanklich zu ergänzenden öffnenden und schließenden Klammern sind zur besseren Übersicht in der Aufstellung mit den Bezeichnungen „[^]“ und „[v]“ gekennzeichnet. Der Vergleich der Programm-Listings zeigt auch, dass bei STEP 7 für Steuerungen der 300er-Serie für logische Verknüpfungen keine Ladekommandos verwendet werden.

In der tabellarischen Gegenüberstellung werden die genannten Kommandofolgen verglichen (Tab. 3).

Gruppe	Klammern	Gruppe	STEP 7-2	Gruppe	STEP 7-3
				1	U([1]
1	LD E 1.6	1	LD E 1.6	1	U E 1.6
1	UN M 0.2	1	UN M 0.2	1	UN M 0.2
1	UN A 1.6	1	UN A 1.6	1	UN A 1.6
1	U M 0.0	1	U M 0.0	1	U M 0.0
2	O([1]	2		2	O [^]
2	LD M 0.1	2	LD M 0.1 [1]	2	U M 0.1
2	UN M 1.7	2	UN M 1.7	2	UN M 1.7
2	) [1]	2	OLD [1]	2	[v]
3	O([1]	3		3	O [^]
3a	LDN E 1.6	3a	LDN E 1.6 [1]	3a	UN E 1.6
3	U([2]	3		3	U([2]
3b	LD A 1.3	3b	LD A 1.3 [2]	3b	O A 1.3
3	O([3]	3		3	O [^^]
3c	LDN M 3.1	3c	LDN M 3.1 [3]	3c	UN M 3.1
3c	U E 2.7	3c	U E 2.7	3c	U E 2.7
3c	) [3]	3c	OLD [3]	3c	[vv]
3c		3c		3c	) [2]
3	) [2]	3	ULD [2]	3	[v]
3	) [1]	3	OLD [1]	3	) [1]
4	U M 1.0	4	U M 1.0	4	U M 1.0
	= A 1.3		= A 1.3		= A 1.3

Tab. 3: Kommandofolgen (AWL) für Netzwerk 1

Funktionsplan



Legende zur Funktionsplandarstellung: -o: NEGATION, &: Logisch UND, ≥ 1: Logisch ODER, =: ZUWEISUNG
Fig. 11: Beispiel Netzwerk 1 (FUP)

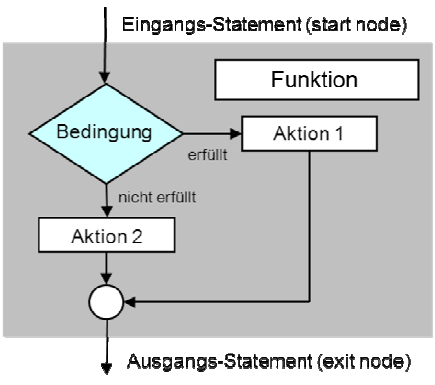
STEP 7-3 ist in der Programmdarstellung Funktionsplan in Blöcken strukturiert (Fig. 11). Gegenüber der Darstellung im Programmeditor ist die Graphik geringfügig verändert, um unter anderem die Notwendigkeit der ersten Klammerung, dargestellt als farbig hinterlegter Bereich, gegenüber den vorher dargestellten Programmstrukturen zu verdeutlichen. Die Reihenfolge der Bearbeitung der Kommandos ist durch Pfeile dargestellt und entspricht der Reihenfolge aus den bereits vorher gezeigten Darstellungsarten.

Arbeitsweise der Kommandos

Weil SPS-Programme die Folge seiner Kommandos sind gilt das auch für seine Teile. In der Folge wird die Funktion der Kommandos untersucht. Die Kommandos benötigen Hilfsgrößen, die ihren „inneren" Zustand darstellen, wie z.B. temporäre Zwischenergebnisse, die durch die Kommandos erreicht werden und in den folgenden Kommandos genutzt werden. Die Aufgabe der Kommandos ist, auch aktuelle innere Zustände zu berücksichtigen und diese gegebenenfalls auszugeben. Der aktuelle Zustand wird in der Regel durch den Zustand des vorhergehenden Kommandos beeinflusst und beeinflusst seinerseits den Zustand des folgenden Kommandos. Der gesamte Zustandsraum $\rightarrow \boldsymbol{Z}^{+}_{i-1} \rightarrow \boldsymbol{Z}^{+}_{i} \rightarrow \boldsymbol{Z}^{+}_{i+1} \rightarrow$ usw. ist die Reihe von Zuständen, die aus dem Zuständen des Prozessabbilds zusammen mit den Hilfsgrößen gebildet wird.[25]

3.4 Kommandos als Flussdiagramme

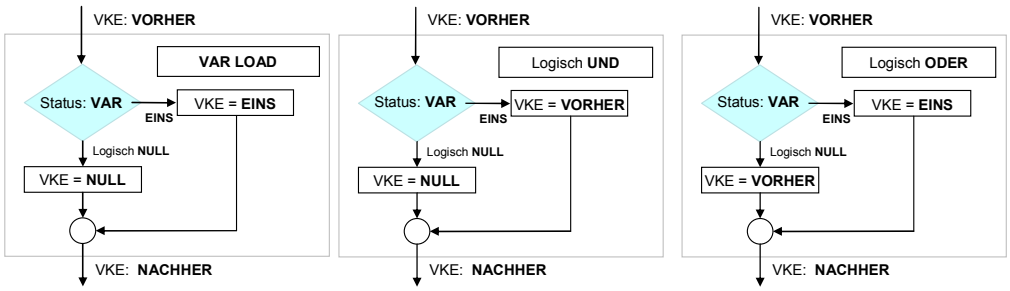
Zugrunde gelegt wird ein prinzipielles Schema, das für alle Kommandos gleichartig aufgebaut ist und in der Folge den SPS-Kommandos entsprechend derer Funktionen abgewandelt wird.



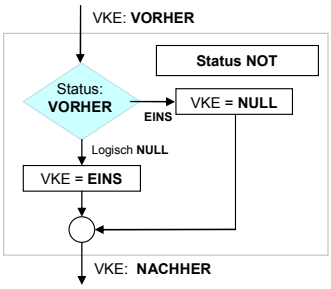
Als „Eingangs-Statement" wird jeweils ein im genau vorhergehenden Arbeitsschritt ermitteltes Ergebnis einer Berechnung genutzt, bezeichnet mit **VKE** (Verknüpfungsergebnis), als „Ausgang-Statement" das neu ermittelte Ergebnis. Als Entscheidungselemente werden neben den **VKE** auch Variable (**VAR**) aus den Ein- bzw. Ausgangsabbildern der Zustände einer SPS herangezogen. Je nachdem, ob die der Entscheidung zugrunde gelegte Bedingung erfüllt ist oder nicht werden Aktionen ausgelöst, die das Ergebnis der Berechnung beeinflussen. Ein SPS-Kommando ist abgearbeitet, wenn das Ausgangsstatement bestimmt worden ist.

[25] Solche Hilfsgrößen sind z.B. Verknüfungsergebnisse (**VKE**), Variablenzustäbe aus dem Speicherabbild (**VAR**) und Werte, die bei Klammeroperationen zwischengespeichert werden.

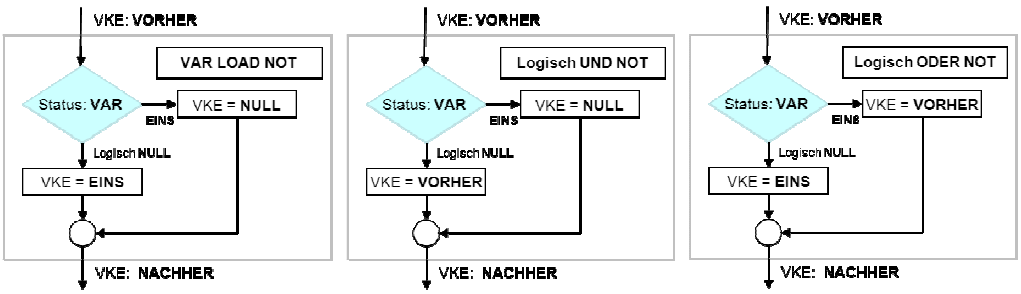
Die Verknüpfungen eines SPS-Programms werden direkt durch serielles Aneinanderreihen der dargestellten Funktionen realisiert. Dazu wird ein in Anweisungsliste gegebenes SPS-Programm herangezogen. Das einfachste SPS-Kommando ist die Lade-Funktion. Weil dabei nur ein logischer Wert übertragen werden muss, liegt keine Bedingung vor und es sind daher auch keine logischen Entscheidungen notwendig. Aus Gründen zur Vereinheitlichung der Darstellung wird das gewählte Schema beibehalten. Die Lade-Funktion benötigt kein Eingangsstatement, das in diesem Fall nicht genutzt wird,



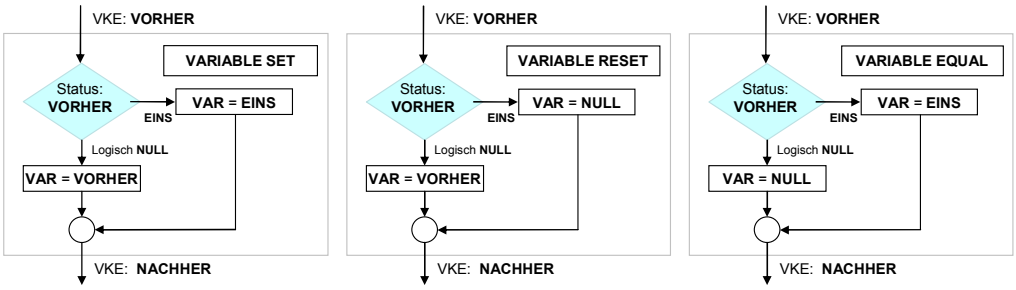
Abgefragt wird der Status einer Variablen aus dem Eingangsabbild oder dem temporären Zwischenergebnis. Im Grundbefehlssatz sind die Kommandos „**LOAD**“, „**AND**“ bzw. „**OR**“ vorgesehen. Diese Abfragen können mit der Negation, die einfach nachgeschaltet wird, in Serie verbunden werden.



Bei der Negation wird das Verknüpfungsergebnis invertiert.



Als Erweiterung können diese Kommandos als „**LOAD NOT**“, „**AND NOT**“ bzw. „**OR NOT**“ in ähnlicher Art vorgesehen werden.



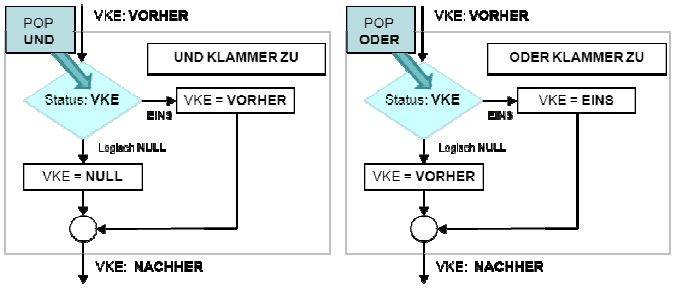
Manipuliert wird der Status einer Variablen im Ausgangsabbild im temporären Zwischenergebnis. Beim Kommando „**EQUAL**“ muss die Negation vor der Zuweisung erfolgen, oder ein zusätzliches Kommando „**NOT EQUAL**“ eingeführt werden.

Einführung von Klammerebenen

Die Syntax einer SPS kennt je nach Hersteller der Systeme unterschiedliche Darstellungen von Verzweigungen, die insbesondere in der graphischen Darstellung von komplexeren Verknüpfungen notwendig werden. In der Folge werden zwei Arten der Implementierung von Verzweigungen innerhalb von Netzwerken diskutiert. Weil in der Darstellungsart Anweisungsliste für Verzweigungen Klammern verwendet werden und derartige Einklammerungen auch ineinander verschachtelt werden können, wird häufig der Begriff „Klammerebene“ zur Unterscheidung der Reihenfolge in der Bearbeitung verwendet.

Im zuerst genannten Fall wird bei Einführung einer Verzweigung der Operator angegeben, der nach dem Schließen der Klammerebene durchgeführt werden soll. Klammerungen werden entweder konjunktiv oder disjunktiv mit der weiteren Logik verknüpft. Die zugehörigen Kommandos lauten dann „**UND KLAMMER**“ bzw. „**ODER KLAMMER**“ und werden erst mit der schließenden Klammer „**KLAMMER ZU**“ verarbeitet. Durch das Kommando „**UND KLAMMER**“ bzw. „**ODER KLAMMER**“ werden jeweils das letzte Verknüpfungsergebnis zusammen mit einer Steuerungsvariablen für „**UND**“ bzw. „**ODER**“ mit der Anweisung „**PUSH**“ in einem Zwischenspeicher (z.B. Stack) gespeichert.

Bei der Abarbeitung der schließenden Klammer wird durch die Steuerungsvariable ein Kommando kombiniert und heißt daher „**UND KLAMMER ZU**“ bzw. „**ODER KLAMMER ZU**“.



Im zweiten Fall wird bei Einführung einer Verzweigung davon ausgegangen, dass bei der Darstellungsart Kontaktplan zur Verzweigung ein weiterer parallel angeordneter Linienzug vorgesehen ist. Erst durch die Kommandos wird festgelegt, ob die Verknüpfung entweder konjunktiv oder disjunktiv durchzuführen ist. Daher wird die öffnende Klammer nicht verwendet, sondern mit einem Ladekommando der Beginn einer Verzweigung gekennzeichnet. Die zugehörigen Kommandos lauten dann „**UND KLAMMERAUSDRUCK**“ bzw. „**ODER KLAMMERAUSDRUCK**“.

In der Bearbeitung einer Kommandofolge bewirkt bzw. benötigt das erste Ladekommando eines Netzwerkes keine Stackoperation. Jedes weitere Ladekommando innerhalb eines Netzwerkes muss jedoch bewirken, dass ein vorher berechnetes Zwischenergebnis automatisch auf dem Stack zwischengespeichert wird. Damit können daher die oben genannten Kommandos „**UND KLAMMER ZU**“ bzw. „**ODER KLAMMER ZU**“ wie zuvor genutzt werden.

Die Unterscheidung, ob also das laufende Ergebnis einer Berechnung auf Grund des Berechnungsschritts zwischengespeichert werden muss oder nicht kann eindeutig zugeordnet werden: existiert in der Kommandofolge ein Kommando für das Schließen einer Klammer, muss ein vorher ermitteltes Zwischenergebnis genau vor dem Kommando für das Laden eines neuen Zustandes gespeichert werden.

Beispiel: **Klammern**

Auf den ersten Blick ist aus der Kontaktplandarstellung die Bearbeitungsreihenfolge der Kommandos nicht zu erkennen, während die zeilenweise Abarbeitung der Anweisungsliste bereits aus der Darstellung hervorgeht. Verdeutlicht wird dies in der Skizze, worin zusätzlich das Zusammenfassen von parallellaufenden Verzweigungen in der Kontaktplandarstellung hervorgehoben ist. Aus dem Programm-Listing in AWL ist die Reihenfolge der Bearbeitung eindeutig abzulesen.

Bei Verwendung von Klammerbefehlen muss jedenfalls bei den öffnenden Klammen berücksichtigt werden, dass erst bei der schließenden Klammer die Art der Verknüpfung des Zwischenergebnisses zu berücksichtigen ist.

Beispiel: **STEP 7-2**

Bei **STEP 7-2** in der Programmdarstellung AWL ist auf die sonst üblichen Klammerbefehle verzichtet worden. Stattdessen sind eine Art indirekter Darstellung von Klammerebenen eingeführt worden, die in einem Flussdiagramm die Kommandofolge des als Beispiel herangezogenen Netzwerks zeigt.

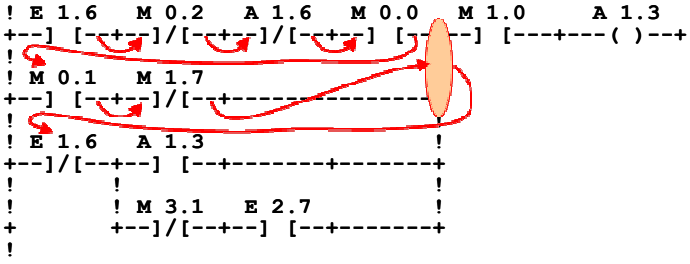
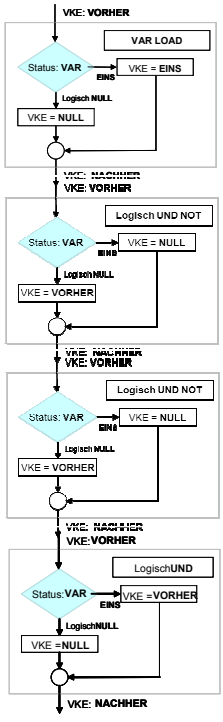


Fig. 12: Beispiel Netzwerk 1 – Kommandofolge (Ausschnitt)

Die in Fig. 12 dargestellte Kommandofolge von Netzwerk 1 wird in der Folge durch Reihung der Kommandos in der Form eines Flussdiagramms dargestellt. Es ergibt sich mit den definierten Bausteinen folgender Ablauf.



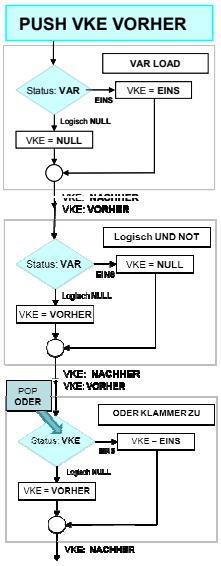
LD E 1.6

UN M 0.2

UN A 1.5

U M 0.0

In der Bearbeitungsfolge der Kommandos wird zuerst die erste Gruppe barbeitet und ein Zwischenergebnis ermittelt. Auf Grund des Umstandes, dass die gesamte Kommandofolge in diesem Programmabschnitt logisch **UND** verknüpft wird, wird zuerst der Status des Eingangs (**VAR** von **E1.6**) aus dem Speicherabbild geladen und sein Zustand als neu ermitteltes Ergebnis als **VKE** (Verknüpfungsergebnis) dem folgenden Kommando übergeben. Das nächste Kommando ist eine logische **UND** Verknüpfung. Bei negierten Ausdrücken, gekennzeichnet durch den Zusatz **NOT** beim Kommando, muss der abgefragte Status invertiert werden (**VAR** von **M0.2**). War der Status logisch **NULL** wird hier wegen des Kommandos „**UND NICHT**“ der alte Wert (**VKE VORHER**) weitergegeben. Nach der Entscheidung, die durch jedes Kommondo getroffen wird, steht ein neuer Funktionswert als Verknüpfungsergebnis fest (**VKE NACHHER**). Die weiteren Elemente werden in gleicher Art angeordnet. Zuletzt steht ein Ergebnis der ersten Gruppe bereit.




```
LD   M 0.1
UN   M 1.7
OLD
```

Die zweite Gruppe beginnt mit einem Ladekommando, das bedeutet, dass das vorher ermittelte Ergebnis zwischengespeichert werden muss. Für die Zwischenspeicherung wird hier ein bestimmter Speicherbereich (in der Form eines **STACK**) reserviert. Wie bereits vorher ausgeführt, wird an dieser Stelle die Einklammerung des folgenden Ausdrucks notwendig. Die Kommandofolge wird danach genauso wie zuvor beschrieben bearbeitet. Mit dem Kommando **OLD** (Oder Klammer schließen) wird diese Gruppe abgeschlossen und mit dem Ergebnis aus Gruppe 1 verknüpft und das Ergebnis für die weitere Programmbearbeitung bereitgestellt.

Die weiteren Kommandos werden in der gleichen Art aneinandergereiht. Auf diese Art und Weise kann das gesamte SPS-Programm zu einem Flussdiagramm zusammengesetzt werden.

Laut der Definition eines Netzwerkes beginnt jedes Netzwerk mit einer Abfrage. Die typische Abfrage ist das Ladekommando. Daher wird bei der Zählung der Klammerebenen der erste Ladebefehl nicht berücksichtigt. Jedes weitere Ladekommando wirkt wie eine öffnende Klammer, die durch die Kommandos **ULD** bzw. **OLD** wieder geschlossen werden.

Beispiel: **STEP 7-3**

Das Problem ist, dass der Formalisierung einer Reihenfolge von Kommandos die entscheidende Bedeutung für die Funktionalität einer SPS zukommt. Gleichgültig, in welcher Schreibweise der vom Programmierer erzeugte Quelltext dargestellt wird und welche syntaktischen Regeln für das Erstellen festgelegt sind: innerhalb der SPS wird vom Prozessor der SPS jede Befehlszeile interpretiert. Hilfreich wäre dazu, den Mikrocode des Prozessors zu kennen.

Während die Formalisierung in den vorher diskutierten Schreibweisen logisch schlüssig ist, gibt es in der Programmdarstelllung AWL von **STEP 7-3** einige Besonderheiten, die nicht unmittelbar in Flussdiagramme umsetzbar sind. Insbesondere sind das die Verknüpfungen am Beginn von Netzwerken. In der Darstellungsart **STEP 7-3** kann unmittlbar als erstes Kommando ein logisches UND, ein logisches ODER oder sogar ein UND KLAMMER AUF Kommondo vorkommen.

Folgt man der bisher diskutierten Vorgehnsweise der Formalisierung, wo ein vorher ermitteltes Verknüpfungsergebnis („**VKE VORHER**“) entsprechend des folgenden Kommandos ver-

knüpft wird, müsste zu Beginn eines neuen Netzwerkes das „**VKE VORHER**“ mit logisch **EIN** vorbesetzt werden, wenn eine logische UND Verknüpfung das erste Kommando einer Kommandofolge ist oder mit logisch **NULL** vorbesetzt werden, wenn eine logische ODER Verknüpfung das erste Kommando einer Kommandofolge ist. Beim Kommando UND KLAMMER AUF als erster Befehl in einem Netzwerk müsste das „**VKE VORHER**“ mit logisch **EINS** vorbesetzt werden, damit der folgende Klammerausdruck korrekt in die Kette der Verknüpfungen eingebunden wird.

Ähnlich ist der Fall für das ODER Kommando ohne Argumente. Grundsätzlich arbeitet dieses Kommando wie ODER KLAMMER AUF. Das Problem ist, die zugehörige Position der schließenden Klammer festzustellen.

Für die Untersuchung wird das Beispielnetzwerk von vorher in der AWL Programmdarstellung herangezogen (Tab. 4, mittlere Spalte).

STEP 7-3 sieht in der Programmdarstellung AWL direkte und indirekte Klammerbefehle vor. Das Kommando „**ODER**“ ohne Adressierung wirkt wie „ODER Klammer auf“. In der tabellarischen Darstellung in AWL ist das Kommando mit einem Zusatz als „**O [^]**“ gekennzeichnet, die zugehörige schließender Klammer mit „**[v]**“. Programmtechnisch ist die schließende Klammer unmittelbar vor einem weiteren **ODER** Kommando ohne Adressierung oder unmittelbar vor der schließenden Klammer des logisch **UND** Kommandos zu berücksichtigen.

Eine weitere Besonderheit liegt bei **STEP 7-3** im Umstand begründet, dass das Ladekommando fehlt. Folgt in AWL eine konjunktive Verknüpfung beginnt die Kommandofolge mit den Operatoren **U** bzw. **UN**, folgt eine disjunktive Verknüpfung sind es die Operatoren **O** bzw. **ON**. Nachdem in der normalen Bearbeitung jeweils ein vorher ermittelter Zustand in die Berechnung eines Verknüpfungsergebnisses einfließt, müsste der Status der mit „**VKE VORHER**“ bezeichneten Variablen entweder mit logisch **EIN** oder logisch **NULL** vorbesetzt werden. Einfacher ist eine Transformation der Kommandos derart vorzunehmen, dass als Zwischenschritt an der geeigneten Programmstelle solche Kommandos als „**LADE**“ Kommandos interpretiert werden.

In der Gegenüberstellung ist die transformierte AWL dargestellt (Tab. 4, letzte Spalte). Felder im Programm-Listing, wo Veränderungen vorgenommen worden sind, sind farbig hinterlegt. Mit der transferierten AWL in **STEP 7-3*** sind jedoch noch nicht alle Probleme gelöst: entgegen der Definition, dass als erstes Kommando in einem Netzwerk eine Abfrage stehen muss, beginnt diese Kommandokette mit einem Klammerbefehl. Der einzig mögliche Klammerbefehl in **STEP 7-3** am Netzwerkanfang ist das Kommando **U(**. Die Lösung ist, den Netzwerkwechsel als Kommando in die Kommandofolge des SPS-Programms mit

aufzunehmen und bei jedem Netzwerkwechsel den Status der mit „**VKE NACHHER**“ bezeichneten Variablen auf logisch **EIN** zu setzen.

Gruppe	Klammern	Gruppe	STEP 7-3	STEP 7-3*	Zeile
		1	U([1]	U(	1
1	LD E 1.6	1	U E 1.6	LD E 1.6	2
1	UN M 0.2	1	UN M 0.2	UN M 0.2	3
1	UN A 1.6	1	UN A 1.6	UN A 1.6	4
1	U M 0.0	1	U M 0.0	U M 0.0	5
2	O([1]	2	O [^]	O(	6
2	LD M 0.1	2	U M 0.1	LD M 0.1	7
2	UN M 1.7	2	UN M 1.7	UN M 1.7	8
2	) [1]	2	[v]	)	9
3	O([1]	3	O [^]	O(	10
3a	LDN E 1.6	3a	UN E 1.6	LDN E 1.6	11
3	U([2]	3	U([2]	U(	12
3b	LD A 1.3	3b	O A 1.3	LD A 1.3	13
3	O([3]	3	O [^^]	O(	14
3c	LDN M 3.1	3c	UN M 3.1	LDN M 3.1	15
3c	U E 2.7	3c	U E 2.7	U E 2.7	16
3c	) [3]	3c	[vv]	)	17
3c		3c	) [2]	)	18
3	) [2]	3	[v]	)	19
3	) [1]	3	) [1]	)	20
4	U M 1.0	4	U M 1.0	U M 1.0	21
	= A 1.3		= A 1.3	= A 1.3	22

Tab. 4: Kommandofolge transformiert (AWL) für Netzwerk 1

Es ergibt sich folgende Vorschrift für die Transformation:

- Bei einem neuen Netzwerk wird das **VKE** auf logisch **EIN** gesetzt. Das ist grundsätzlich unproblematisch, weil unmittlbar als erstes Kommando nur ein logisches UND, ein logisches ODER oder ein UND KLAMMER AUF Kommando vorkommen kann.
- Ist das erste Kommando kein UND KLAMMER AUF Kommondo sondern ein logisches UND oder ein logisches ODER, werden diese Kommandos durch ein LADE Kommando ersetzt (vgl. **STEP 7-3***, Zeile 2 in Tab. 4).
- Allgemein gilt: das auf eine öffnende Klammer folgende Kommando wird durch ein LADE Kommando ersetzt (vgl. **STEP 7-3***, Zeile 2, 7, 11, 13 und 15 in Tab. 4).
- Das Kommando ODER ohne Argument wird durch das ODER KLAMMER AUF Kommando **O(** ersetzt (vgl. **STEP 7-3***, Zeile 6, 10 und 14 in Tab. 4).
- Prinzipiell müssen die Klammern mitgezählt werden. für jede neu hinzugenommene öffnende Klammer muss eine schließende Klammer gesetzt werden. Dabei muss gelten: die zuletzt geöffnete Klammer muss zuerst geschlossen werden. Öffnende Klammern werden ausschließlich beim Kommando ODER ohne Argument eingefügt. Weil im Beispiel drei

Klammern geöffnet werden, müssen die Positionen der schließenden Klammern bestimmt werden.

- Existiert ein ODER Kommando ohne Argument muss die schließende Klammer spätestend vor einer Zuweisung eingefügt werden. Als Zuweisungen gelten neben den bereits genannten Zuweisungen (**S**, **R**, **=**) auch die später betrachteten Zuweisungen für die Flankenbildung, Zeitfunktionen und Zähler.
- Folgt auf das Kommando ODER ohne Argument ein weiteres Kommando ODER ohne Argument, muss das Zwischenergebnis gespeichert werden. Daher wird vor den ODER KLAMMER AUF Kommando eine schließende Klammer eingefügt (vgl. **STEP 7-3***, Zeile 9 in Tab. 4). Anmerkung: im Beispiel die erste schließende Klammer.
- Innerhalb des eingeklammerten logischen UND Ausdrucks ist ein ODER Kommando ohne Argument enthalten. Das Zwischenergebnis muss gespeichert werden und daher muss eine schließende Klammer eingefügt werden (vgl. **STEP 7-3***, Zeile 17 in Tab. 4). Anmerkung: im Beispiel die zweite schließende Klammer.
- Im Bereich des ersten Klammerausdruckes wird ebenfalls eine zusätzliche Klammer geöffnet, weil auch dort ein ODER Kommando ohne Argument enthalten ist (vgl. **STEP 7-3***, Zeile 6 in Tab. 4). Diese Klammer muss innerhalb des ersten Klammerausdrucks geschlossen werden (vgl. **STEP 7-3***, Zeile 19 in Tab. 4). Anmerkung: im Beispiel die dritte schließende Klammer.

Beispiel: **Zwischenmerker an Stelle von Klammern**

Das Konzept, Zwischenergebnisse temporär in reservierten Speicherplätzen bereitzustellen, ist ein Konzept, das sich nicht durchgesetzt hat. Zweck dieses Konzepts ist, auf Klammerungen von Ausdrücken völlig zu verzichten. Vorteilhaft ist dabei, dass für die Klammerungen bereits vorhandene Kommandos genutzt werden. Soll das als ursprüngliches Netzwerk herangezogene Beispiel in die Darstellung „Verwendung von Zwischenmerkern“ (**Zwischen M**) ohne Veränderung der ursprünglich vorgesehenen Kommandofolge transferiert werden, ergibt sich das folgende Programmlisting (Tab. 5).

Benötigt werden mehrere Zwischenmerker, die eine Entsprechung zur Klammerung haben (Tab. 5, mittlere Spalte). In der variierten Darstellung (Tab. 5, rechte Spalte) wird gezeigt, dass durch ein einfaches Umstellen in der Reihenfolge der Kommandos auf allfällige Klammerungen zur Gänze verzichtet werden kann, ohne die Funktionalität des untersuchten Netzwerkes zu verändern. Dass die Funktionalität identisch ist, ist für die Gruppen 1 und 2 offensichtlich und bei Gruppe 3 einfach nachvollziehbar.

Gruppe	Klammern	Gruppe	Zwischen M
1	LD E 1.6	1	LD E 1.6
1	UN M 0.2	1	UN M 0.2
1	UN A 1.6	1	UN A 1.6
1	U M 0.0	1	U M 0.0
2	O([1]	1	= ZM-0
2	LD M 0.1	2	LD M 0.1
2	UN M 1.7	2	UN M 1.7
2	) [1]	2	O ZM0-0
3	O([1]	2	= ZM-0
3a	LDN E 1.6	3a	LDN E 1.6
3	U([2]	3a	= ZM-1
3b	LD A 1.3	3b	LD A 1.3
3	O([3]	3b	= ZM-2
3c	LDN M 3.1	3c	LDN M 3.1
3c	U E 2.7	3c	U E 2.7
3c	) [3]	3c	O ZM-2
3c		3c	U ZM-1
3	) [2]	3	O ZM-0
3	) [1]	3	= ZM-0
		3	L ZM-0
4	U M 1.0	4	U M 1.0
	= A 1.3		= A 1.3

Gruppe	Zwischen M
3c	LDN M 3.1
3c	U E 2.7
3b	O A 1.3
3a	UN E 1.6
3	= ZM

1	LD E 1.6
1	UN M 0.2
1	UN A 1.6
1	U M 0.0
1	O ZM
1	= ZM

2	LD M 0.1
2	UN M 1.7
2	O ZM
2	= ZM

	L ZM
4	U M 1.0
	= A 1.3

Tab. 5: Kommandofolge mit Zwischenmerkern (AWL) für Netzwerk 1

Ein Grund dafür, dass sich dieses Konzept nicht durchsetzen konnte, ist vermutlich der etwas erhöhte Programmieraufwand durch das Festlegen, in welcher Reihenfolge Kommandofolgen verwendet werden dürfen.

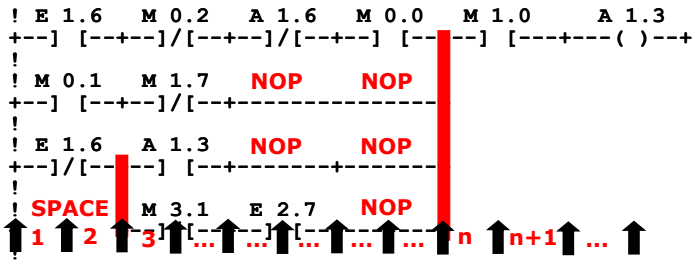
Variante in der Bearbeitung von Kommandofolgen

In der Folge wird in einer Variante gezeigt, dass die Programmbearbeitung innerhalb einer SPS auch auf andere Art erfolgen kann, ohne die Funktionalität des Programms zu verändern.

Beispiel: **Spaltenweise Programmbearbeitung**

Während die typische Darstellung von SPS-Programmen im Editor so gestaltet ist, dass je nach Art der Anordnung der Elemente auch mehrere Netzwerke im Bearbeitungsfenster angezeigt werden können, gab es Steuerungen, die für ein Netzwerk ein festes Format vorgesehen hatten. Die Größe war dabei so gewählt, dass genau ein Netzwerk auf einer Bildschirmseite des Programmiersystems angezeigt werden konnte.

Die hier untersuchte Art der Bearbeitung orientiert sich an der Programmdarstellung in Kontaktplan und kommt gänzlich ohne Klammerbefehle aus.



Die Auswertung der Verknüpfungen erfolgte parallel und für alle Spalten zeitgleich. Für die erste Zeile gilt dabei, dass von einem logisch **EIN** Status in der ersten Spalte [1] ausgegangen wird. In der zweiten Spalte wird dieser Zustand mit dem Status von **E 1.6** logisch **UND** verknüpft [2] und im Spaltenelement [3] zwischengespeichert. Parallel dazu werden Zeile 2 und Zeile 3 bearbeitet. Interessant ist die Zeile 4: hier wird der logisch **EIN** Status [1] mit keinem Element (**SPACE**) verknüpft [2]. Das Ergebnis ist als logisch **NULL** definiert. In Spalte 3 existiert eine Verbindung zur Spalte 4, die als logisch **ODER** zu verstehen ist. Daher werden dort die Ergebnisse von Zeile 3 mit Zeile 4 logisch **ODER** verknüpft und das Ergebnis sowohl in Zeile 3 als auch in Zeile 4 für die weitere Berechnung verwendet. Eine weitere **ODER** Verknüpfung stellt die senkrecht dargestellte Verbindung in der Spalte n dar. An Stellen, wo nur horizontale Verbindungen bestehen, erfolgen keine Operationen [**NOP**] und die Ergebnisse der Verknüpfungen werden einfach weitergereicht.

Weitere Kommandos

Neben den bisher untersuchten logischen Verknüpfungen haben auch so genannte Flanken-Merker, auch als Impuls-Merker bezeichnet, in Steuerungsabläufen eine gewisse Bedeutung: wenn z.B. sichergestellt werden soll, dass ein Signalzustand nur für die Dauer eines SPS-Zyklus gültig ist. Die Realisierung ist mit den genannten Kommandos bereits möglich (vgl. Programmausschnitt in AWL, Fig. 13).

```
LD    E 0.0
UN    M 0.1
=     M 0.0
LD    E 0.0
=     M 0.1
```

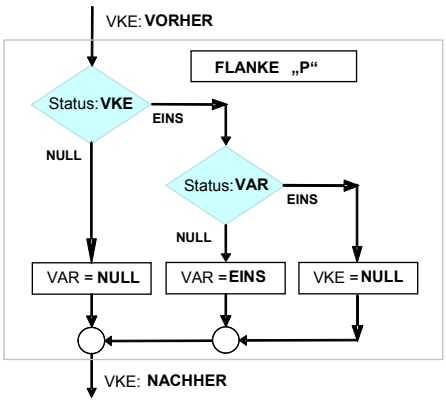
Fig. 13: Flanke AWL

Funktion: Wird der Eingang **E0.0** aktiviert, wird der Merker **M0.0** eingeschaltet (Zeile 3) weil zu diesem Zeitpunkt der Merker **M0.1** noch nicht aktiviert worden ist. Im übernächsten Schritt wird der Merker **M0.1** ebenfalls eingeschaltet (Zeile 5), der Merker **M0.0** bleibt eingeschaltet. Im darauffolgenden SPS-Zyklus sperrt der Merker **M0.1** (Zeile 2) den Merker **M0.0**, der deshalb ausgeschaltet wird (Zeile 3). **M0.0** ist daher genau für die Dauer eines Zyklus aktiviert.

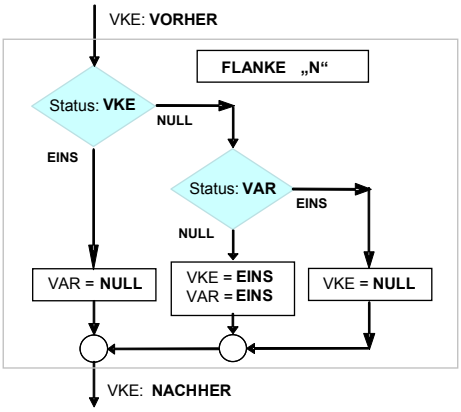
Ähnlich wird eine fallende Flanke programmtechnisch realisiert. Dagegen kennt die Programmiersprache **STEP 7-2** eigene Befehle, bezeichnet mit **-]P[-** bzw. **-]N[-** für „edge up“ (steigende Flanke) bzw. „edge down“ (fallende Flanke) **ohne** zusätzliche Adressierung für den benötigten Flankenmerker. Die Schreibweise **STEP 7-3** unterscheidet die Be-

fehle **-(P)-** bzw. **-(N)-** für die steigende bzw. fallende Flanke, benötigt jedoch die zusätzliche Festlegung einer Adresse für die Flankenmerker.

Grundsätzlich gilt, dass anstelle des in Fig. 13 genannten Eingangs **E0.0** ein bliebiger logischer Ausdruck zulässig ist.



Als Flussdiagramm dargestellt arbeitet der Ablauf der Funktion der steigenden (positiven) Flanke wie folgt: der Status eines vorher ermittelten Verknüfungsergebnis (**VKE VORHER**) wird abgefragt: ist der Status logisch EINS wird der Status des Flankenmerkers (**VAR**) abgefragt. Ist der Status von **VAR** logisch **NULL** wird die Variable auf logisch EINS gesetzt und das **VKE** unverändert für die weitere Berechnung ausgegeben. Ist der Flankenmerker gesetzt, wird das **VKE** auf logisch **NULL** zurückgesetzt. Ist der Status des vorher ermittelten Verknüpfungsergebnisses wieder logisch **NULL** wird der Flankenmerker zurückgesetzt.



Ähnlich arbeitet eine fallende (negative) Flanke, die beim Abschalten der Ansteuerung einen Impuls für die Dauer eines SPS-Zyklusses ausgibt. Wird das ansteuernde Signal ausgeschaltet (**VKE** ist logisch **NULL**) und der Flankenmerker (**VAR**) nicht gesetzt, wird **VKE** und Flankenmerker **VAR** auf logisch **EINS** gesetzt. In den folgenden SPS-Zyklen wird als Verknüpfungsergebnis logisch **NULL** ausgegeben, solange, bis der Flankenmerker wieder zurückgesetzt wird.

Zeitglieder (Timer)

Als weitere Funktionen sind Zeitglieder von Interesse. AWL kennt bei Zeitgliedern unterschiedliche Varianten. Ein Zeitglied wird in der Programmiersprache **STEP 7-3** ähnlich

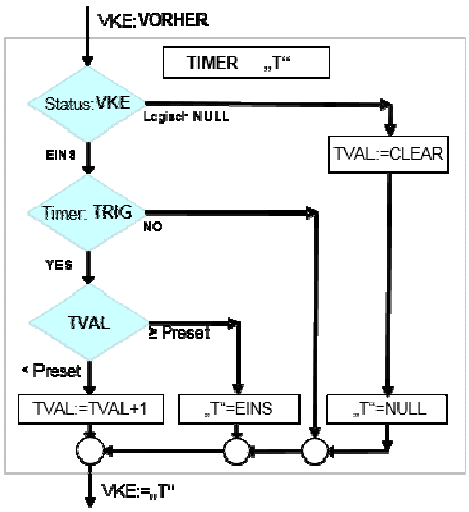
```
L     S5T#...S
SE    T <Nummer>
```

Fig. 14: Zeitglied (AWL)

Fig. 14 dargestellt. Angegeben wird dabei die Zeitvorwahl (Preset) zusammem mit der Zeitbasis (Millisekunden, Sekunden usw.) die Funktion (z.B. **SE** für einschaltverzögert) und eine Adresse für die Zeitfunktion.

Prinzipiell arbeiten Zeitglieder ähnlich wie Aufwärts-Zähler. Je nach der gewählten Zeitbasis erzeugt die SPS Zählimpulse, mit denen das Zeitzählerregister inkrementiert wird, voausge-

setzt, dass die Zeitfunktion durch das Programm aktiviert worden ist. Wird die Zeitvorwahl erreicht bzw. überschritten, wird eine Variable gesetzt (logisch EINS) und kann im weiteren SPS-Programm wie jede andere Variable logisch verknüpft werden. Fällt die Aktivierung der Zeitfunktion weg, wird das Zeitzählerregister auf NULL zurückgesetzt.



Als „Zwischenschritt“ werden bei Zeitgliedern oder Zählern Parameter geladen. Für Zeitglieder ist der Programmiersprache **`STEP 7-3`** (Serie 300) die Bezeichnbung „**`S5T#`**“ vorgesehen, danach folgt die Angabe zur vorgewählten Zeit (das **`PRESET`**) und eine Bezeichnung für die Zeitbasis, im Beispiel „**`S`**“ für Sekunden.

Als Flussdiagramm ist ein einschaltverzögertes Zeiglied dargestellt: die Zeitfunktion startet, wenn der Status eines vorher ermittelten Verknüpfungsergebnis (**`VKE VORHER`**) logisch EINS ist. Steht ein vom internen Taktgeber erzeugter Zeitzählimpuls an (**`TRIG`**) erfolgt der Vergleich zwischen Zeitvorwahl (**`PRESET`**) und dem internen Zeitzählwert (**`TVAL`**). Ist die Vorwahl noch nicht erreicht wird der Zeitzähler inkrementiert (**`TVAL+1`**). Wird die Vorwahl überschritten, wird die zugehörige Zeitvariable des Zeitzählers auf logisch EINS gesetzt. Fällt das Ansteuersignal weg, werden der Zeitzählwert gelöscht und die Zeitvariable ausgeschaltet.

Im Wesentlichen beschränken sich die Unterschiede bei verschiedenen SPS-Herstellern darin, dass bei manchen Steuerungen die Zeitbasis bestimmten Zeitzählern fest zugeordnet sind und dass es bei der Arbeitsweise Unterschiede gibt, die für die weiteren Betrachtungen keine Bedeutung haben. Betrachtet man genau einen bestimmten Zyklus eines SPS-Programms, interessieren in der Regel genau zwei Zustände: ist ein Zeitzähler bereits abgelaufen und hat eine Schaltfunktion ausgelöst oder ist der Ablauf noch nicht beendet.

Zusammenfassung

Werden die diskutierten Varianten nach der Art, wie SPS-Programme durch die Editoren dargestellt werden, gegenübergestellt, zeigt sich, dass die Reihenfolge von Kommandos in Kommandofolgen entsprechend ihrer schrittweisen Bearbeitung vergleichsweise einfach aus der Darstellungsart AWL abgelesen werden kann. Die Funktion der einzelnen Kommandos lässt sich in Flussdiagrammen veranschaulichen.

Kommandofolgen lassen sich durch das aneinander Reihen der Kommandos erzeugen und folgen einer durch die Programmierung vorgegebenen Reihenfolge.

In Bezug auf das eingangs erwähnte Zustandsabbild ist festzuhalten, dass Zwischenwerte benötigt werden, die als Verknüpfungsergebnis (**VKE**) bezeichnet worden sind. Diese Zwischenwerte sind temporäre Variable, die nicht in das Zustandsabbild aufgenommen werden.

Weitere Kommandos wie z.B. Zeitglieder bzw. Zähler oder die Auswertung von Flanken sind für die Überprüfung einzelner Logik-Zyklen eines SPS-Programms nur insoweit von Interesse, als nur Informationen zum Status solcher Funktionen zu berücksichtigen sind. Ein typisches Beispiel für die Verwendung von Zeitfunktionen ist, diese als „Quasizustände“ zu nutzen. Wenn bei einem Bewegungsablauf für eine Positionsüberwachung kein eigener Sensor angeordnet werden kann, kann es in bestimmten Anwendungsfällen ausreichen, anstelle des fehlenden Sensors eine Zeitfunktion zu nutzen.

3.5 Speicherbereiche für Zustandsabbilder

Für die Werte der Zustände des Eingangsabbildes, der Ausgangszustände und die Zustände von Merkern und weiteren Variablen sind in einer SPS eigene Speicherbereiche vorgesehen. Während die Zustände der Merker dynamisch vom Programm verändert werden können, werden die Zustände von Eingängen bzw. von Ausgängen bei den meisten SPS temporär konstant gehalten und nur zu bestimmten Zeitpunkten im Zyklusbetrieb der SPS aktualisiert.

Weil Eingangs-Zustandsabbilder wenigstens für die Dauer eines Zyklus nicht verändert werden, ist die Reaktion eines SPS-Programms für jeden Zyklus vorhersagbar. Diese Vorhersagbarkeit erlaubt die Aussage, dass wenn der Übergang von einem Eingangszustand in seinen korrespondierenden Ausgangszustand im Sinne einer Aufgabenstellung korrekt ist, dass dann auch das den Zustandsübergang erzeugende SPS-Programm zunächst eingeschränkt auf den untersuchten Übergang korrekt sein muss. Wird diese Art der Überprüfung auf alle Eingangs-Zustandsabbilder angewendet, gilt das SPS-Programm als verifiziert.

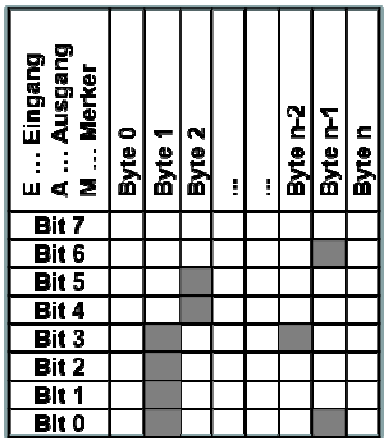


Fig. 15: Speicherbelegung

Fig. 15 zeigt den Speicherbereich für logische Zustände (z.B. Eingänge, Ausgänge, Merker). Die hinterlegten Felder symbolisieren eine Belegung solcher Speicherbereiche. Grundsätzlich ist eine solche Belegung für die Ein- bzw. Ausgänge einer SPS an die Anordnung der Hardware gebunden, d.h. ein bestimmter Eingang ist mit einem aus der Aufgabenstellung bekannten und damit festgelegten Signalgeber verbunden.

Viele SPS lassen es sogar zu, solche Adressierungsbereiche zu konfigurieren. Solche Konfigurationen sind Zuordnungstabellen, die von der SPS verwaltet werden und die den an die SPS angeschlossenen Eingangs- bzw. Ausgangsmodulen ihre Adressbereiche zuordnen. Sie sind für diese Arbeit nur von untergeordnetem Interesse. Von Interesse ist jedoch die Speicherbelegung, die wie im gezeigten Beispiel auch Lücken aufweisen kann. Wichtig ist es festzuhalten, dass jede der belegten Speicherzellen gleichwertig ist. Genutzte Speicherzellen tragen die Statusinfomationen, also jene Zustände, die ein einzelner Eingang, Ausgang usw. einnehmen kann. Der Begriff „Konfiguration“, wie er in dieser Arbeit verstanden werden soll, beschreibt die genannten Zustände. Wird etwa nur ein Teilbereich eines SPS-Programms untersucht, werden die zugehörigen Teilkonfigurationen derart zusammenengestellt, das nicht relevante Zustände im betrachteten Teilbereich unberücksichtigt bleiben und daher zur Vereinfachung und verbesserten Übersichtlichkeit weggelassen werden.

Speicherzellen werden in der SPS durch Operanden (z.B. „E“ für Eingänge) und ihrer Byte- und der Bit-Adresse definiert. Anstelle dieser Adressierung sind auch symbolische Namen zulässig. Zuordnungen erfolgen bei der Programmerstellung. Byte-Adressen sind meist auch mit bestimmten Positionen in der Anordnung von Ein- bzw. Ausgangsmodulen eines SPS-Systems verbunden, z.B. Steckplätzen einer modular aufgebauten SPS.

Auch die Zeitfunktion oder Zähler bewirken Zustände, die im SPS-Programm abgefragt werden. Hier interessieren jedoch nur solche Zustände, die als das Ergebnis von logischen Verknüpfungen zu berücksichtigen sind. Für Zeiten ist dies der Umstand, ob ein Zeitglied angesteuert wird oder nicht bzw. ob ein Zeitablauf bereits beendet worden ist oder nicht. Im Zustandsraum wird daher für Zeiten bzw. Zähler zur Abspeicherung der Zustände jeweils nur ein Byte reserviert, das nur zum Teil genutzt wird (Bit 0 für den Status angesteuert oder nicht, Bit 1 für abgelaufen oder nicht. Aktuelle Zeit- oder Zählwerte werden nicht benötigt).

Zur Vervollständigung der in der Konfiguration festgelegten Zustandsabbilder, werden diesen noch Zwischenergebnisse zugeschlagen z.B. Werte, die bei der Berechnung von Klammerausdrücken erst zu einem späteren Berechnungsschritt benötigt werden.

3.6 Gruppierung von Kommandofolgen

Einzelne Kommandofolgen wie das vorher untersuchte Netzwerk bilden Teilfunktionalitäten innerhalb von SPS-Programmen ab. Solche Programmkonstrukte können für sich einzeln verifiziert werden. Die Ergebnisse besitzen noch eine geringe Aussagekraft. Zur Steigerung der Aussagekraft werden funktionelle Zusammenhänge von (Teil-) Steuerungsaufgaben in der Folge beschrieben und weiter untersucht. Im Anhang befinden sich die zugehörigen Quellprogramme dieser Funktionsgruppen. An Hand dieser Funktionsgruppen werden die Analyseschritte demonstriert. In diesem Abschnitt wird davon ausgegangen, dass eine Art von „innerer“ Struktur bei Steuerungsprogrammen für Maschinen und Anlagen festgelegt werden kann mit der Aussicht, solche Programmfragmente isoliert zu untersuchen.

Ausgangspunkt für diese Überlegung ist die typische Art, wie infomelle Spezifikationen bzw. Aufgabenstellungen für Steuerungsprogramme vorliegen: bei Maschinenabläufen z.B. bei der Bohrmaschine 1 als einführendes Beispiel liegt die Aufgabenstellung in der Folge von Bearbeitungsschritten vor.[26] Solche Beschreibungen sind informell, beziehen sich auf Teilfunktionen einer Steuerung oder Anlage und sind in der Regel als wenn-dann-sonst Beziehungen definiert z.B. „**wenn** Zylinder X seine vordere Endlage erreicht hat **dann** soll [eine weitere Aktion] erfolgen ... **sonst** [eine alternative Aktion] erfolgen...“. Der „Sonst“-Term ist vielfach deshalb nicht beschrieben, weil etwa keine Alternative im geplanten Ablauf existiert oder der Zustand nicht erreichbar ist. Je nach der Beschreibung in der Aufgabenstellung ist eine Ablaufunterbrechung gegebenenfalls zusammen mit einer Fehlermeldung die Folge.[27]

---

[26] Ein Bearbeitungszentrum für Kohlebürsten für Elektromotoren wird als Beispiel herangezogen. In mehreren Stationen wird der Herstellungsprozess in Bearbeitungsschritten wie Bohren, Einbau einer Abschaltvorrichtung, Einstampfen eines Kupferkabels für die Zuführung der elektrischen Energie etc. durchgeführt. Dazu wird das zugehörige SPS-Programm untersucht.

[27] Beschreibung des Ablaufes für Station „Bohren 1“ (Z: ... Zylinder, ES ... Endschalter, v, h oder o, u ... vorne, hinten oder oben, unten)

Kommando Automatikablauf „Start“:
(1) Radialklemmung Zx vor
(2) wenn Zx ES vorne erreicht, ist keine Kohle in der Backe und der Ablauf wird abgebrochen;
(3) nach 0,3 Sekunden Start von Bohrzylinder 1 (Z11) bis ES unten;
(4) wenn ES unten erreicht dann Zeitstart 0,2 Sekunden zum Freischneiden des Bohrers;
(5) wenn Zeit abgelaufen Z11 zurück bis ES oben;
(6) wenn Z11 obere Endlage erreicht hat Radialklemmung Zx öffnen (bis ES) und der Ablauf „Bohren 1“ ist fertig;
(7) wenn Z11 gestartet ist: Einschalten der Absaugung für die Dauer des Bohrvorgangs und Ablasen der Bohrspäne.

Die informelle Beschreibung ist für das Erstellen eines Programms zwar ausreichend, wenn die zusätzlich notwendigen „Nebeninformationen“ zumindest dem Programmierer bekannt sind. Solche Nebeninformationen sind z.B., dass der Antriebsmotor für den Bohrer eingeschaltet sein muss, solange der Bohrvorgang abläuft.

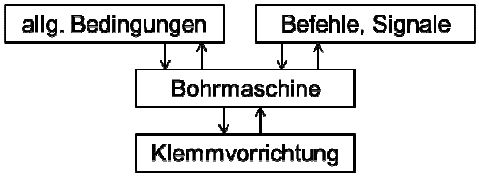


Fig. 16: Bohrmaschine Übersicht

Fig. 16 zeigt in einer Übersicht den Zusammenhang zwischen der Bohrmaschine und ihrer Ansteuerung. Allgemeine Bedingungen sind z.B. Betriebsart oder die Freigabe des Ablaufs, Befehle und Signale sind z.B. die Auslösung im Handbetrieb oder Fehlermeldungen.

Direkt durch die Bohrmaschine wird die Klemmvorrichtung (hier die Radialklemmung) angesteuert.

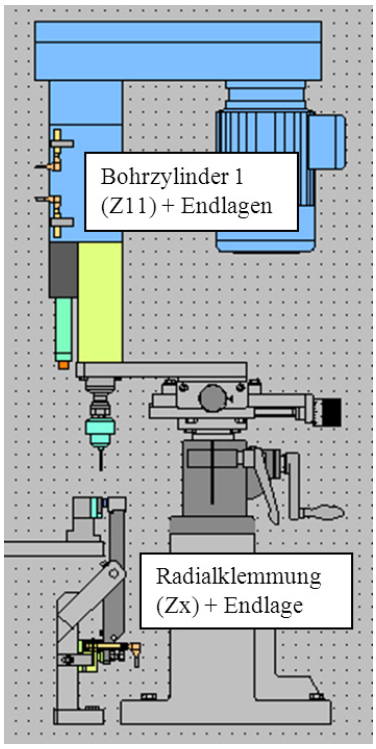


Fig. 17: Bohrstation (schematisch)

Fig. 17 zeigt die schematische Darstellung der Station. Eine Besonderheit bei dem betrachteten Ablauf ist, dass die Radialklemmung nicht nur durch die Station Bohren angesteuert wird, sondern auch von weiteren Stationen, wo das Fixieren der Kohlebürste notwendig ist. Das bedeutet, dass durch den Start des Ablaufes Bohren oder anderer Stationen dieser Ablauf mit gesteuert wird.

Obwohl der Ablauf selbst vergleichsweise einfach ist, gibt es eine Reihe von Querbeeinflussungen, die zu berücksichtigen sind. Im Wesentlichen sind das Fragen wie „Unter welchen Bedingungen darf der Ablauf Bohren überhaupt gestartet werden?“ und „Welche weiteren Steuervorgänge der Maschine werden durch den Ablauf Bohren selbst ausgelöst, welche gesperrt?“ Existierende Abhängigkeiten werden als Querverbindungen in der Fom von Bedingungen oder Steuerkommandos von Programmteilen in die Untersuchung mit einbezogen.

Bedingungen für den Ablauf Bohren werden in anderen Programmmodulen definiert. Solche Bedingungen sind im Beispiel ganz Allgemein, ob und welche Betriebsart ausgewählt worden ist, die Feststellung, ob die Gesamtanlage mit der Hilfsenergie versorgt wird oder ob die Transportvorrichtung ein Werkstück in die korrekte Bearbeitungsposition gestellt hat etc. Steuerkommandos und Statusmeldungen, die durch die Bohrstation an andere Programmmo-

dule weitergeleitet werden, sind bei diesem Programmteil z.B. die Fehlerbehandlung, die Ansteuerung der Radialklemmung, die Sperre bzw. Freigabe für die Transportvorrichtung etc.

Der Quelltext des SPS-Programms ist für eine CPU der Baureihe S7-300 in der Programmiersprache STEP 7 verfasst und als Baustein „FC21 Bohrmaschine 1“ im Anhang gelistet.[28] Vergleicht man die informelle Beschreibung des Ablaufs mit der Umsetzung im SPS-Code sind Ähnlichkeiten erkennbar, insbesondere in den Kommentierungen der Netzwerke. Teilweise stammen solche Unterschiede aus nicht genannten Bedingungen, die zum Teil aus anderen Teilen der Programmbeschreibung zu übernehmen waren.

Es gibt eine Reihe von Gründen, warum SPS-Programme nahezu nie formal definiert werden:

- SPS gesteuerte Maschinen oder Anlagen sind vielfach Prototypen. Werden ähnliche Fertigungseinrichtungen neuerlich gebaut, gibt es Unterschiede.
- Formale Spezifikationen sind dem Konstrukteur von Maschinen oder Anlagen in der Regel fremd.
- Die „Lebendigkeitseigenschaft“ von SPS-Programmen und Fexibilität für Programmanpassungen lässt vergleichweise rasch Programmveränderungen und -optimierungen zu.
- Zum Teil unvollständige Dokumentationen der Aufgabenstellung. Die letztgültige, bis in alle Details gehene Aufgabenstellung steht oft erst nach erfolgter Inbetriebnahme fest. Sie enthäht erst dann alle informellen Anmerkungen, die zum Teil zu Beginn der Programmerstellung nicht dokumentiert worden waren, dennoch zuletzt als Wünsche oder Ergänzungen Teil der Aufgabenstellung werden.[29]

---

[28] Die Darstellung ist in AWL gelistet. Durch symbolische Namen der Adressen für die verwendeten Signale zusammen mit den textuellen Kommentierungen ist der Ablauf auch ohne weitere Erklärungen ablesbar.

[29] Erfahrung aus der Praxis: Die so genannte „mündliche Überlieferung“ sind z.B. Aussagen wie „genauso wie bei der Maschine X“ oder sich aus Nachfrage zu Details ergebende Anpassungen während der Programmerstellung. Das ist insbesondere dann der Fall, wenn Abläufe optimiert werden. Sich daraus ergebende „historisch motivierte“ Verbesserungen sind in solchen Fällen auch mehr oder weniger einfach erfüllbare Wünsche, die sich im Zuge einer Inbetriebsetzung als zielführend herausgestellt haben. Dazu zählen auch jene Notwendigkeiten, die sich im Produktionsbetrieb ergeben, wie z.B. ein Teil muss zusätzlich fixiert werden, damit ein Bearbeitungsablauf fehlerfrei durchgeführt werden kann. Solche Verbesserungen werden zu Voraussetzungen, die später als Funktionen „wie bei Maschine Y“ in die gegebene und folgende Aufgabenstellung mit aufgenommen werden.

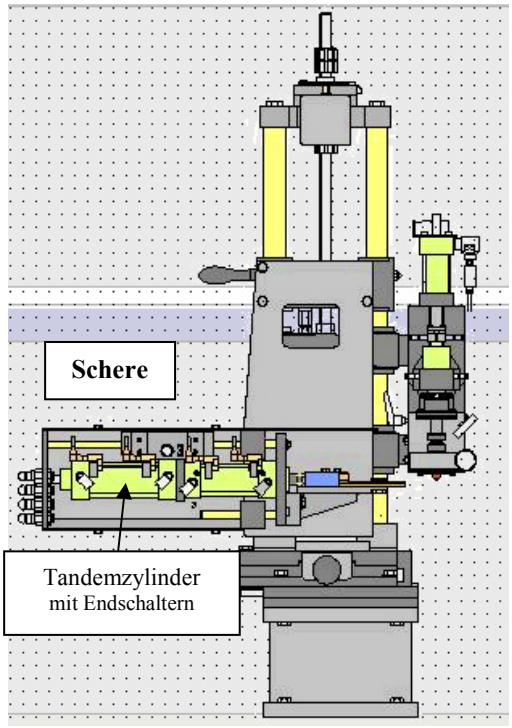


Fig. 18: Stampfstation (schematisch)

Fig. 18 zeigt die schematische Darstellung der Stampfstation. Diese und weitere Stationen werden in den Anhängen B und C weiter untersucht. Ein Teilbereich, der Programmablauf für die Schere, wird in der Folge als Beispiel zur Bestimmung der Verifikationsbedingungen herangezogen.

4 Analyse bei unbekannter Programmstruktur

Dieser Abschnitt beschäftigt sich mit Überlegungen zu SPS-Programmen, deren Struktur nur zum Teil als bekannt vorausgesetzt werden kann. Solche Programme liegen immer dann vor, wenn das im Progammeditor erzeugte Quellprogramm gegeben aber unzureichend dokumentiert worden ist oder wenn das Quellprogramm nur durch das Auslesen des SPS-Programmspeichers gewonnen werden kann. Dennoch kann es notwendig werden, solche Programme zu verifizieren. Eine sehr arbeitsintensive Methode ist die diversitäre Rückwärtsübersetzung der Programme, wo aus einem gegebenen Programmcode die ursprüngliche Aufgabenstellung gewonnen wird. Mit dieser Methode können auch versteckte Fehler aufgedeckt werden.

Müssen die Programmstruktur und Zusammenhänge von SPS-Programmen untersucht werden, ist die minimale Anforderung, dass zumindest die Verwendung der Ein- bzw. Ausgangsebene ausreichend dokumentiert ist, damit auch für solche Programme die vorgeschlagenen Konzepte für die Verifikation anwendbar sind. Konzeptuell wird für das Verifikationsmodell einer SPS festgelegt, dass als Modell-SPS eine „state machine“ zu Grunde gelegt werden kann, die in jedem Programmübergang Eingangsinformationen in resultierende Ausgangsinformationen umwandelt. Weil auch in diesem Modell die Anzahl von Überprüfungsschritten sehr groß werden kann, wird untersucht, wie durch Segmentierung des SPS-Programms eine Reduktion der Schritte ermöglicht wird.

Der grundlegende Gedanke, ein gegebenes Programm in kleinere, weitgehend voneinander unabhängige Bereiche zu gliedern, wird beibehalten.

Bei der Zerlegung eines SPS-Programms in funktionelle Bereiche, stellt sich die Frage, ob auch zeitgleich untersucht werden kann, welche zusätzlichen Informationen aus dem SPS-Programm gewonnen werden können. Die Vorgehensweise wird in zwei Schritte gegliedert: zuerst wird auf Grund des Vorkommens von SPS-Befehlen untersucht, welche Befehle innerhalb geschlossener Befehlsketten durch die Adressierung in einen Zusammenhang zu bringen sind. In einem weiteren Schritt werden zusätzliche Zusammenhänge gesucht, die später noch genauer spezifiziert werden.

Motivation

Die Motivation zur Programmanalyse liegt auch im Umstand begründet, dass es zur unbeabsichtigten, fehlerhaften Adressierungen von Elementen etwa durch Ziffernsturz oder Verwechslungen kommt. Derartige Fehler können im Programmablauf auch über längere Zeit-

räume unentdeckt bleiben und erstmalig zu Fehlreaktionen führen, wenn etwa anlagenbedingt z.B. durch Abnützung einer Mechanik diese schwergängiger als ursprünglich wird und dadurch auch der geplante Ablauf derart verändert wird, dass der Fehler schlagend wird.

Die Frage ist also, gibt es im Programm Elemente, denen in einem bestimmten Programmabschnitt keine Funktion zuordenbar ist. Direkt kann in der Programmanalyse eine derartige Konstellation nicht geortet werden. Deshalb wird in der Folge eine Methode vorgeschlagen, mit Hilfe derer Zusammenhänge in Programmteilen untersucht werden.

Eine typische Anwendung einer SPS soll als einführendes Beispiel dienen: ein getakteter Fertigungsablauf, bei dem mehrere Fertigungsstationen der Reihe nach ein Produkt bearbeiten. Grob gesehen können folgende Programmelemente geortet werden: ein allgemeiner Teil, mit Schnittstellen zur Transportsteuerung und zu den einzelnen Fertigungseinheiten, die Transportsteuerung mit Schnittstellen zur Fertigungseinheit und die Fertigungseinheit mit Schnittstellen zum Transport. Fig. 19 zeigt die Zusammenhänge.

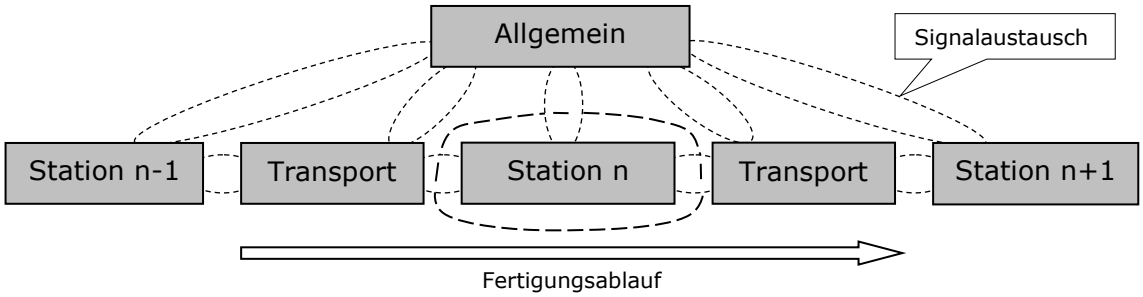


Fig. 19: Programmstruktur Fertigungsablauf

Betrachtet wird das SPS-Programm einer Station und der unmittelbare Zusammenhang zu benachbarten Programmelementen. Es gibt für diesen Anwendungsfall typisch nur eine geringe Anzahl von Verknüpfungen zum allgemeinen Teil wie etwa die Wahl einer Betriebsart, Signale zum Start des Ablaufs und die Meldung, dass der Arbeitsschritt fertig gestellt ist. Für die Transportsteuerung muss bekannt sein, ob ein Transport gerade möglich ist, etwa wenn die betrachtete Station eine Ruhestellung eingenommen hat. Ein direkter Signalaustausch zu weiteren Arbeitsstationen wird in der Regel nicht notwendig sein. Zusammen mit den direkt in der Station benötigten Verknüpfungen und den eben angeführten kann von einem definierbaren Programmsegment ausgegangen werden, dass keinen weiterern Signalaustausch benötigt, also für sich alleine betrachtet werden darf.

Die Motivation für die Programmanalyse ist, abgrenzbare Programmbereiche zu lokalisieren und festzustellen, welche Überschneidungen zu weiteren Programm Konstrukten existieren. Für weiterführende Untersuchungen werden die Überschneidungsbereiche jeweils auch den

angrenzenden Programmbereichen zugeordnet. Damit wird sichergestellt, dass die Analyse lückenlos auf Kosten einer geringen Redundanz bleibt.

4.1 Programmierumgebungen

Typische Programmierumgebungen für SPS stellen insbesondere für die Fehlersuche bei der Inbetriebnahme einer Steuerung Funktionen zur Verfügung wie das Suchen und Ersetzen von Befehlen oder Befehlsteilen, Kommentierungen, Symbolen und weitere. Die geordnete Zusammenfassung der Suchergebnisse ist dabei in der Regel nach der Adressierung sortiert, nicht jedoch nach dem logischen Zusammenhang im Programm.

Aufsteigende und absteigende Sortierung lassen sich in der Regel einstellen. Typisch bei der Programmierung ist, dass versucht wird, auch eine Art logische Ordnung dadurch herzustellen, dass bei der Programmierung etwa Merkerbereiche innerhalb einer Steuerungsfunktion zusammenhängend gewählt werden. Weil jedoch das Programm im Fortschritt der Programmierung „lebt“ und dabei auch mehrfach überarbeitet wird, kann in der Regel die ursprünglich gewählte Ordnung nicht beibehalten werden, zumal z.B. Adressen von Merkern beliebig verwendbar sind, vergleichbar mit freien Variablen. Die Änderung der Ordnung von direkten Eingangs- bzw. Ausgangsadressen hingegen erfordert auch eine Veränderung der Verdrahtung. In diesem Fall werden in der Regel bereits vorher bestimmte „Reserven“ für notwendige Ergänzungen oder Erweiterungen genutzt.

Programmierumgebungen sind herstellerspezifisch. In der Folge werden diese wichtiger SPS-Hersteller in Bezug auf ihre Marktpräsenz beleuchtet, wobei der Fokus im Speziellen auf die so genannten Querverweislisten (cross reference) gelegt wird, mit der Fragestellung, welche Möglichkeiten standardmäßig bereits angeboten werden.[30]

Unter dem Titel „Suchfunktionen in einem Projekt“ sind für die Steuerungsfamilien von Allen-Bradley Funktionen zur Unterstützung der Fehlersuche zusammengefasst. Damit können etwa alle Elemente, die im SPS-Programm vorgesehen sind, gesucht werden. Mittels Filtereinstellung werden Operanden, Befehle bis hin zu Kommentierungen als Suchkriterien definiert (vgl. [AB09]).

Der GX-Developer von Mitsubishi ist die Entwicklungsplattform für die Programmierung der SPS dieses Herstellers. Im Kapitel 6 „Suchen/Ersetzen“ von [MI12] sind Funktionen zur Unterstützung der Programmierarbeit zusammengefasst. Ähnlich zu typischen Textprogrammen können Programmelemente gesucht und gegebenenfalls auch ersetzt werden. Es können damit

[30] Hersteller in alphabetischer Reihenfolge

sowohl Operanden als auch Anweisungen gesucht werden. Mit dem Befehl „Querverweisliste“ werden gezielt Operanden gesucht und ihre Position im Programm in Form einer Liste angezeigt. Ergänzt wird die Suchfunktion durch das Auffinden verwendeter Operanden (vgl. [MI12]).

Omron bezeichnet seine Programmieroberfläche als „CX-Programmer“ [OM11]. Im Kapitel 3 „Project Reference“ sind die Möglichkeiten Programmreferenzen zu suchen und zu bearbeiten näher behandelt. Der Cross-Reference Report ermöglicht die Verwendung von Symbolen innerhalb unterschiedlicher Speicherbereiche und kann auch die Werte darstellen, die durch das Programm bearbeitet werden. Die Funktion „Address Reference“ zeigt die verwendeten Adressen innerhalb des SPS Programms. Überwachungsfenster für Variable ergänzen die Funktionalität. Die Suchfunktion mit und ohne Ersetzen wird, wie bei nahezu allen Editoren der unterschiedlichen Hersteller von SPS, ermöglicht, wobei nach Adressen und symbolischen Bezeichnungen und zusätzlich auch nach Mnemonics und Kommentaren gesucht werden kann (vgl. [OM11]).

Auch bei Schneider-Electric [SE11] wird die Suchfunktion als das Suchen nach Daten bezeichnet. Es wird dort von vordefinierter Suche gesprochen, wo ausgehend vom Daten- oder Sprachеneditor ein Objekt ausgewählt wird, dessen Referenzen angezeigt werden sollen. Die Suchergebnisse werden im Fenster „Querverweise“ angezeigt. Neben der Suche existiert auch die Option für das Ersetzen von Daten. Wie bei allen Editoren werden nur Daten oder Instanzen ersetzt, die mit den gesuchten kompatibel sind. Die Anzeige der Suchergebnisse in dekomprimierter oder komprimierter Form kann sortiert nach Referenz oder Typ erfolgen. Es existiert auch die Anzeigemöglichkeit in der so genannten flachen Form, wobei die Ergebnisse auch nach den referenzspezifischen Spalten sortiert werden können. Unterstützt wird die Sortierung der Suchergebnisse nach unterschiedlichen Kriterien (vgl. [SE11]).

SPS der Steuerungsfamilien von Siemens bieten unterschiedliche Such- und Editiermöglichkeiten für Programme an. Unterschieden werden Belegungsplan, Aufruf- und Abhängigkeitsstruktur sowie Informationen zur Speicherauslastung. Im Belegungsplan wird ein Überblick gegeben, welche Bits der Operanden bestimmter Speicherbereiche innerhalb eines SPS-Anwenderprogramms bereits belegt sind. Weiters wird angezeigt, ob eine Adresse durch einen Zugriff aus einem S7-Programm heraus belegt ist oder ob die Adresse einer Baugruppe zugeordnet ist. In der Aufrufstruktur ist die Aufrufhierarchie der Bausteine eines strukturierten SPS-Anwenderprogramms dargestellt. Sie gibt einen Überblick über verwendete Bausteine und deren Abhängigkeiten. Als Abhängigkeitsstruktur wird die Liste der im Anwenderprogramm verwendeten Bausteine bezeichnet. Sie ist hierarchisch geordnet (vgl. [SI11]).

Die meisten Programmeditoren bieten neben den Suchfunktionen auch die Überprüfung der Syntax. Für die im Folgenden definierte Programmanalyse reicht der Funktionsumfang dieser Programmeditoren jedoch nicht aus. Beispielsweise wird die Detektion von Zusammenhängen bestimmter Programmelemente aus Suchergebnissen ermöglicht, jedoch werden diese nicht als solche gekennzeichnet. Das nachträgliche Feststellen von möglichen Abhängigkeiten in SPS-Programmen für weitere Untersuchungen ist entsprechend aufwendig. Dieser Vorgang soll neben zusätzlichen Untersuchungen automatisiert durch erweiterte Analysefunktionen ermöglicht werden.

4.2 Analysetechniken

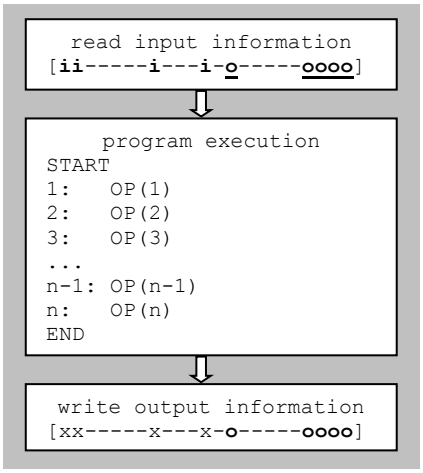
Die Verifikation eines SPS-Programms ist aufwendig. Konzeptuell wird für die Verifikation eines SPS-Programms hier festgelegt, dass jeder Programmübergang von Eingangsinformationen zu den resultierenden Ausgangsinformationen einzeln und nacheinander geprüft und dabei verifiziert wird. Die hohe Anzahl dieser Überprüfungen soll durch die eingangs erwähnte Segmentierung des SPS-Programms gemindert werden. In diesem Abschnitt werden Möglichkeiten zur Segmentierung gesucht. Dazu sind besondere Kenntnisse über die Struktur und Zusammensetzung des SPS-Programms notwendig (vgl. [KG13]).[31]

Der Suchraum für die Untersuchung eines SPS-Programms steigt exponentiell mit der Anzahl der zu untersuchenden Variablen. Aus diesem Grund ist die Teilsuche nicht nur eine Frage der Effizienz, sondern entscheidet auch darüber, ob eine Untersuchung an die Grenzen der Möglichkeit eines Werkzeugs für die Programmverifikation stößt. Mit Hilfe der Programmanalyse des zu untersuchenden SPS-Programms werden mehrere Teilziele verfolgt. Zuerst soll festgestellt werden, welche Programmelemente wo und wie im zu untersuchenden SPS-Programm verwendet worden sind. Diese Art der Untersuchung ist als die klassische Querverweisliste (cross reference) bekannt und üblicherweise in die Programmierumgebung für SPS-Steuerungen integriert. Jedes der verwendeten Programmelemente beschreibt eine Funktion und diese besitzen ihnen zugeordnete Werte und Wertebereiche. Typisch für SPS-Programme ist, dass die Wertebereiche von Funktionen einer bestimmten Ordnung unterliegen, das heißt, es existieren bereits durch die Programmierung festgelegte Bereiche, in denen die zugehörigen Werte für die Bearbeitung bereitgestellt und zwischengespeichert werden. Im vereinfachten Modell wird in der Folge nur zwischen der Abfrage lesend (**`i`** ... input) und schreibend/lesend (**`o`** ... output) unterschieden. Da im Wertebereich der gesamte Raum abgebildet ist, also einschließlich auch jener Informationen, die in einem bestimmten Anwen-

[31] Zum Teil als Konferenzbeitrag „Programmanalyse von SPS-Programmen als Teilschritt für die automatisierte Verifikation“ beim 7. Forschungsforum der österreichischen Fachhochschulen 2013 vorgestellt.

dungsfall nicht benötigt werden, soll weiters festgestellt werden, wie die Nutzung dieser Speicherbereiche erfolgt. Nach gründlicher Untersuchung über mögliche Querverbindungen können ungenutzte Speicherbereiche „eingekapselt“ und damit von einer tiefer gehenden Untersuchung ausgeschlossen werden.

Darüber hinaus erfüllt eine SPS in der Regel eine Vielzahl von Funktionen. Denkt man etwa an die Steuerung einer Fertigungsanlage, existieren dort in der Regel einerseits streng voneinander separierbare Funktionen, andererseits aber auch Funktionen, die nur wenige Signale miteinander austauschen müssen. Auch hier liegt Potenzial darin künstlich Bereichsgrenzen einzuführen, um Suchräume zu begrenzen.



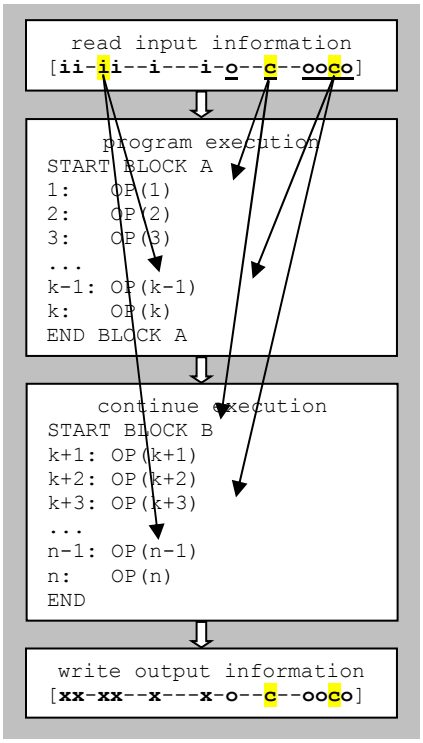
Legende:

i	…	Eingangsinformationen (input)
o	…	Ausgangsinformationen vorher (output pre execution)
o	…	Ausgangsinformationen nachher (output post execution)
x	…	Speicherbereich nicht relevant
-	…	Speicherbereich nicht verwendet (unused)

Fig. 20: Speichernutzung 1

Fig. 20 zeigt, dass im hier dargestellten Programm bestimmte Speicherbereiche nicht benutzt werden. Bei den weiteren Untersuchungen können sie daher außer Betracht bleiben.

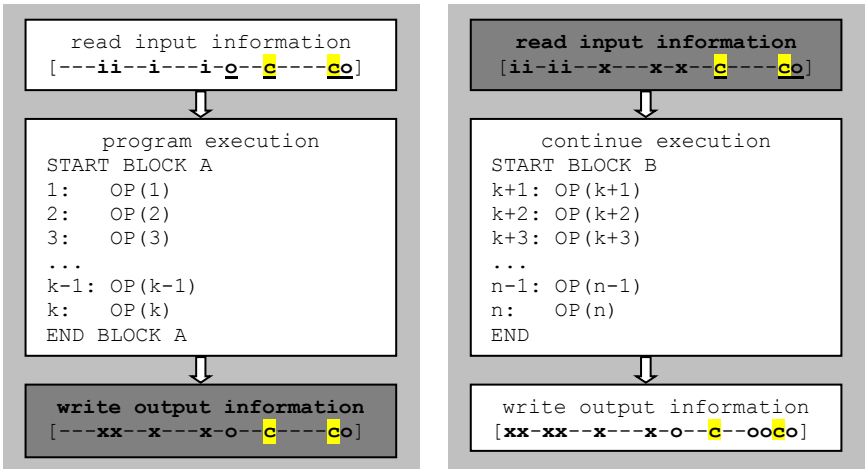
Fig. 21 zeigt, dass einige Informationen in beiden untersuchten Programmblöcken genutzt werden, der Rest sei jeweils einem der Blöcke zugeordnet. Ist die Anzahl der Elemente im Verhältnis zu den weiteren klein, bedeutet das, dass die hier dargestellten Programmfunktionen nur in einem geringen Umfang voneinander abhängen. Dann kann es sich anbieten, die zu untersuchenden Programmblöcke zu teilen um diese getrennt zu untersuchen (Fig. 22).



Legende:

i	…	Eingangsinformationen (input)
c	…	gemeinsame Ausgangsinformationen vorher (common output pre execution)
c	…	gemeinsame Ausgangsinformationen nachher (common output post execution)
o	…	Ausgangsinformationen vorher (output pre execution)
o	…	Ausgangsinformationen nachher (output post execution)
x	…	Speicherbereich nicht relevant
-	…	Speicherbereich nicht verwendet (unused)

Fig. 21: Speichernutzung 2



Legende: wie Fig. 2

Fig. 22: Speichernutzung 3, Programm in Abschnitte geteilt

Zum besseren Verständnis sind in Fig. 22 die Blöcke „write output information“ nach dem Programmdurchlauf des Teilprogramms (Block A) bzw. „read input information“ vor die Be-

arbeitung eines weiteren Teilprogramms (Block B) ergänzt worden. Dort sind auch die erst für diesen Block zusätzlich benötigten Informationen eingetragen. Am Ende der gesamten Programmbearbeitung sind zwangsweise, dem normalen SPS-Zyklus entsprechend, auch die nicht in diesem Block bearbeiteten Ergebnisse vermerkt.

Diese Art der Darstellung soll verdeutlichen, dass Elemente, die eindeutig und exklusiv einem bestimmten Programmteil zuordenbar sind, in weiteren Programmabschnitten keine Funktionalität besitzen, dort also nichts bewirken und daher von der Untersuchung abgekapselt werden können. Zur Verdeutlichung sind diese Elemente weggelassen worden. Die Art und Weise, wie die Interaktion zwischen Teilprogrammen erfolgt, muss bei der Detailuntersuchung berücksichtigt werden.

Mit dieser Methode wird es möglich, die ursprünglich umfangreiche Untersuchung zu vereinfachen. Die Abhängigkeit der so getrennten Programmteile ist gegeben durch gemeinsam genutzte Speicherbereiche einerseits, andererseits begründet durch den Umstand, dass in jedem Teiluntersuchungsfall nicht nur der zu untersuchende Teilbereich des Programms, sondern immer das gesamte SPS-Programm zu berücksichtigen ist.

Diese Programmanalyse als ein der Verifikation vorausgehender Schritt der Untersuchung kann weitgehend automatisiert stattfinden. Im Folgenden wird ein Programmanalysator definiert. Mögliche Szenarien, die bei der Programmanalyse eintreten können, sind als Anwendungsfall beschrieben (use case). Dazu wird untersucht, welche Umstände bei der Zielerreichung eintreten können. Die Ergebnisse der Untersuchungen sollen in der technischen Umsetzung berücksichtigt werden.

Zunächst werden Anforderungen an den Programmanalysator grob definiert. Hier stellt sich zunächst die Frage nach syntaktischen und semantischen Besonderheiten im SPS-Programm. Bestimmte Fehler, wie unbenutzte oder sinnwidrig mehrfach benutzte Elemente im Adressraum, werden bei der formalen syntaktischen Prüfung von SPS-Programmeditoren nicht erkannt, weil derartige Programmstrukturen innerhalb eines SPS-Programms grundsätzlich lauffähig sind. Die Validierung der Kompatibilität von Befehl und Typ bzw. Typ und Adresse ist in der Regel bereits durch die Programmeditoren erfolgt, kann jedoch durch die Programmanalyse bestätigt werden. Zuletzt sollen die gewonnenen Ergebnisse für die weitere Auswertung zur Verfügung stehen.

Eine herkömmliche (handelsübliche) Programmierumgebung für die Entwicklung von SPS-Programmen verifiziert durch syntaktische Analyse sämtliche Funktionen. Dabei wird bereits verifiziert, ob die Anweisungen der Spezifikation der gewählten Programmiersprache entsprechen. Ob darüber hinausgehend diese Anweisungen der geplanten logischen Abfolge ent-

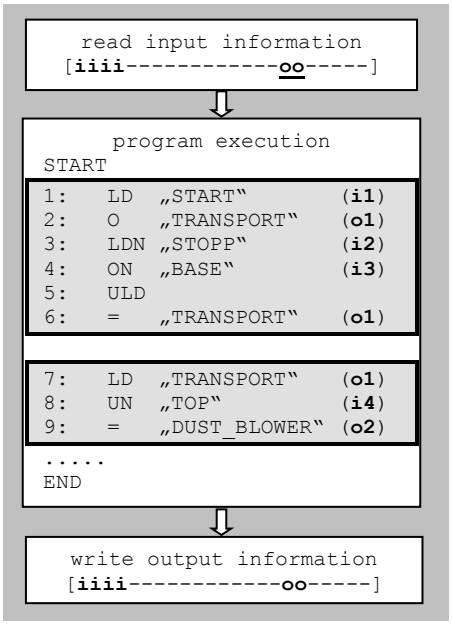
sprechen und ob diese im Gesamtprogramm konsistent ist, wird nicht untersucht. Aber erst dadurch kann festgestellt werden, ob das Endergebnis, das zu untersuchende SPS Programm, den ursprünglich gestellten Zielsetzungen genügt.

Eine weitere Hilfestellung kann die Kontrolle der Plausibilität eines Programms ergeben. Dazu wird das fertig erstellte SPS-Programm auf logische Fehler in der Verwendung von Variablen und dem zugeordneten zeitlichen Ablauf von Zuweisungen und dem Auslesen dieser untersucht.

Zusammenfassend ist festzustellen, dass die hier vorgestellte Programmanalyse bezüglich der gestellten Anforderungen über die typischen, mit der Programmieroberfläche mitgelieferten Werkzeuge hinausgeht. Die weitgehend vollständige syntaktische Programmanalyse ist bereits bei nahezu allen Standard-Programmierumgebungen gegeben, die zusätzlichen Ansätze zur semantischen Analyse fehlen. Das Feststellen, mit welchem Überdeckungsgrad Eingangs- oder Ausgangsinformationen zwischen Programmblöcken genutzt werden, existiert in keiner der bekannten Programmierumgebungen. Hier wird in einem ersten Ansatz überlegt, den Überdeckungsgrad als zusätzliche Parameter einer Anfrage festzulegen, um eine Art von „Bindungsgrad“ von Programmblöcken zueinander innerhalb der gesamten Programmstruktur von SPS-Programmen zu bestimmen.

Im sehr vereinfachten Beispiel (Fig. 23) wird die Abhängigkeit (der „Bindungsgrad“) von zwei Programmblöcken untersucht. Festgelegt sind im Testprogramm in sich abgeschlossene Programmblöcke. Eine einzige Information von Block 1 wird in Block 2 als Abfrage genutzt, die im Testprogramm als „Transport“ (**`o1`**) bezeichnet worden ist. In diesem Fall ist der Bindungsgrad 1. Alle anderen Elemente sind direkt entweder dem Block 1 oder dem Block 2 zuordenbar. Dieser Umstand schließt allerdings nicht aus, dass die Zustände der hier betrachteten Elemente auch an anderen Stellen des Programms in weiteren Programmblöcken verwendet werden können.

Wird in der Folge der Bindungsgrad diskutiert, wird im Hinblick auf die geplante Nutzung der Analyseergebnisse festgelegt, dass Funktionswerte, die ausschließlich lesend zur Verfügung stehen (Elemente vom Typ **`i`**), bei der Ermittlung unberücksichtigt bleiben. Grund dafür ist, dass sie innerhalb eines Bearbeitungszyklus konstant gehalten werden.



Funktion Bohrmaschine: Genutzt werden vier Eingänge und zwei Ausgänge zur Ansteuerung des Vorschubs einer Bohrmaschine. Die Eingangsinformationen (**START**, **STOP**, **BASE** und **TOP**) steuern die Ausgänge (**TRANSPORT** und **DUST_BLOWER**). Wird „**START**“ aktiviert, wird dem „**TRANSPORT**“ ein Fahrbefehl nach abwärts zugewiesen, der solange aktiv bleibt, bis der „**STOP**“ Befehl die Bewegung anhält, jedoch erst dann, wenn zusätzlich die untere Endlage erreicht worden ist (**BASE**). Das Aufheben der Startfunktion bewirkt die Richtungsumkehr. Solange die Abwärtsbewegung erfolgt, sorgt eine Blasdüse (**DUST_BLOWER**) für das Entfernen der Bohrspäne.

Fig. 23: Testprogramm Bohrmaschine (ähnlich **STEP 7**)

4.3 Programm-Module

Der Umfang von SPS-Programmen hängt von der Aufgabenstellung ab. In der Regel lassen sich SPS-Programme strukturiert aufbauen. Dazu werden funktionell zusammenhängende Programmteile in Programmmodulen zusammengefasst. Der Bindungsgrad innerhalb dieser Module ist vergleichsweise hoch gegenüber dem Bindungsgrad zwischen einzelnen Modulen. Als Beispiel sei die Programmstruktur einer Bearbeitungsmaschine herangezogen: ein Block für übergeordnete Funktionalitäten wie Wahl der Betriebsart, Kontrolle des Ablaufes, etc. steuert und kontrolliert Subfunktionen wie den Fertigungsablauf 1 bis zum Fertigungsablauf n (Fig. 24). Im einfachsten Fall wird von der Ablaufkontrolle das Startsignal für einen Fertigungsablauf generiert und das Ende der dort stattfindenden Bearbeitung abgewartet (Bindungsgrad 2) oder entsprechend höher, wenn zusätzliche Informationen ausgetauscht werden müssen.

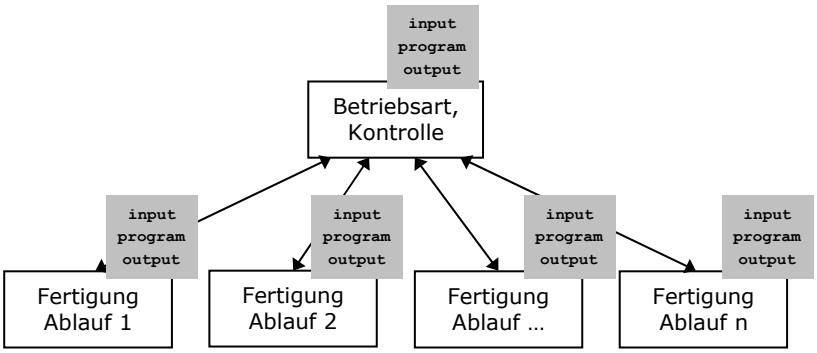


Fig. 24: Programmstruktur

Beim Programmentwurf für ein SPS-Programm wird in der Regel eine derartige Teilung schon vorgenommen. Im Zuge der Weiterentwicklung der Programme kommt es jedoch relativ häufig vor, dass derartige Strukturen durchbrochen werden. Müssen zum Zwecke der Verifikation Modulgrößen begrenzt werden, kann bei der semantischen Programmanalyse nach derartigen Programmstrukturen gesucht werden. Rein formal gesehen wird es in einem Programmteil wie etwa im Programm für den Fertigungsablauf 1 zur vermehrten Verwendung bestimmter Funktionswerte (Typ **`i`**, **`o`**) kommen. Diese werden sich in der Regel als „Häufungspunkte“ zumindest in der Entwurfsphase sogar in bestimmten Sektoren befinden, obwohl dies für die weitere Betrachtung ohne besondere Bedeutung ist. Wird als weiterer Parameter eine maximale Gesamtanzahl für die Summe der abgefragten und zugewiesenen Funktionswerte festgelegt (Summe Typ **`i`** und Typ **`o`**), kann durch Filtermechanismen die Größe zusammenhängender Programmabschnitte festgelegt werden. Als Regel muss dabei gelten, dass als kleinste Einheit unabhängig von der Anzahl der darin bearbeiteten Funktionswerte der als Netzwerk definierte Programmabschnitt festgelegt wird. Diese Größe ist mitbestimmend für die Komplexität bei der Verifikation eines SPS-Programms. Mit Hilfe einer derartigen Untersuchungsmethode können sogar umfangreichere Module in Programmabschnitte aufgeteilt werden.

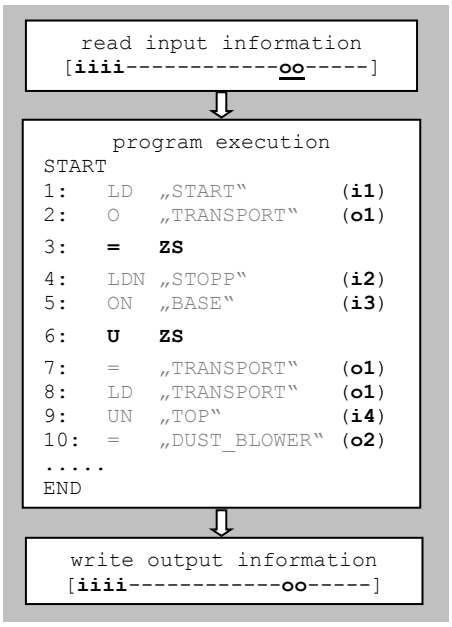
4.4 Semantische Programmanalyse

Funktionswerte vom Typ **`i`** (input) stehen nur lesend zur Verfügung. In der Regel wird die Wertezuweisung an Programmelemente vom Typ **`i`** durch die Syntaxüberprüfung bereits beim Editieren durch das Programmiersystem erkannt. Für Programmelemente, die Werte vom Typ **`o`** (output) nutzen, gilt diese Einschränkung nicht. Es ist allerdings in der Regel semantisch nicht sinnvoll, Wertezuweisungen an Funktionswerte vom Typ **`o`** bei bestimmten Programmelementen mehrfach zuzulassen oder sie ausschließlich lesend zu verwenden. Typische Beispiele dafür sind bei Zuweisungsbefehlen zu finden. Werden Setzbefehle verwendet, muss es an geeigneter Stelle auch zugeordnete Rücksetzbefehle geben. In diesem Fall muss sogar die Zuweisung wenigstens zweifach im SPS-Programm erfolgen. Werden jedoch di-

rekte Zuweisungen für Ausgänge verwendet, liegt bei mehr als einmaliger Zuweisung in der Regel ein Fehler im SPS-Programm vor.

Generelle Aussagen zur Typüberprüfung sind jedoch nicht möglich, insbesondere in Bezug auf Programmelemente, die temporäre Variablenwerte nutzen. Ein besonderes Beispiel ist bei Programmiersprachen zu finden, bei denen Klammerbefehle direkt zu programmieren sind (Fig. 25).

Ziel der semantischen Programmanalyse ist auch das Erkennen von Programmelementen, die temporäre Werte nutzen. Da temporäre Variablenwerte in der Regel nur innerhalb eines SPS-Zyklus genutzt werden, sind ihre Werte vor Beginn der Programmbearbeitung und nach dem Ende der Programmbearbeitung nicht von Interesse und können daher unberücksichtigt bleiben. Für die Verifikation bedeutet dieser Umstand, dass der Suchraum entsprechend kleiner gehalten werden kann.

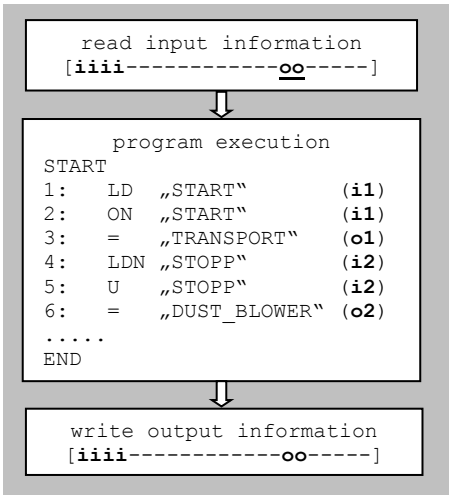


Funktion: Programmablauf wie beim Testprogramm für die Bohrmaschine. Zwischenergebnisse (**ZS**) wechseln ihre Werte je nach Programmfortschritt immer wieder. Der Ausdruck Programmzeile 1 + 2 wird zusammengefasst und mit dem nachfolgenden Programmteil verknüpft. In der Regel sind diese Zwischenergebnisse nur wenige Programmschritte gültig. Da in dieser Sprachvariante Klammerausdrücke fehlen, wird das Zwischenergebnis vielfach im Programm genutzt.

Fig. 25: Testprogramm (Zwischenspeicher)

SPS-Programmelemente können in der Anordnung der Programmausdrücke fallweise tautologisch bzw. kontradiktorisch kombiniert sein. Dies führt dazu, dass die Variablenwerte für Zuweisungen im Vorhinein bereits feststehen. In der Regel kann auch für derartige Programm Konstrukte von unrichtiger Programmierung ausgegangen werden. In der semantischen Programmanalyse sollen derartige direkte Kombinationen aufgespürt werden. Unter direkt wird

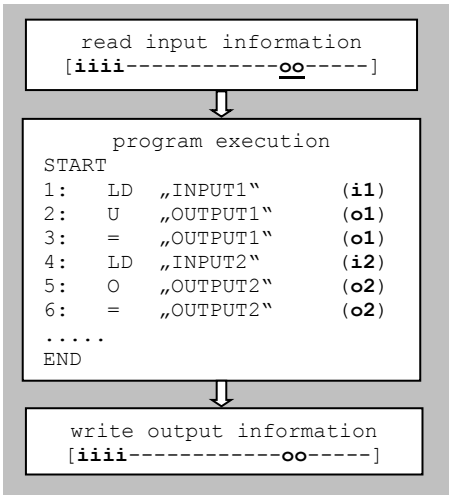
in diesem Zusammenhang verstanden, dass die Tautologie oder Kontradiktion direkt im Programm erkennbar ist (Fig. 26). Indirekte Tautologie oder Kontradiktion wäre dann gegeben, wenn etwa über Zwischenkombinationen mehrerer Programmelemente Tautologien oder Kontradiktionen gegeben sind. In diesem Fall ist eine tiefer gehende Programmanalyse erforderlich.



Funktion: Die Funktion „TRANSPORT“ ist immer eingeschaltet, die Funktion „DUST_BLOWER“ immer ausgeschaltet. Die Programmelemente müssen nicht notwendigerweise unmittelbar aufeinander folgen.

Fig. 26: Testprogramm (direkte Tautologie oder Kontradiktion)

Weiters kann mit Hilfe der semantischen Programmanalyse festgestellt werden, ob Programmelemente des Wertebereichs Typ **o** nur lesend verwendet worden sind. Diese Art der Verwendung ist in der Regel auf einen Programmierfehler zurückzuführen, im Gegensatz bei der nur schreibenden Verwendung.

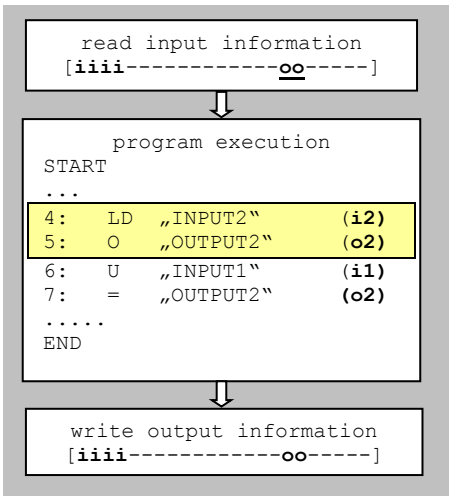


Funktion: Die Funktion „OUTPUT1“ kann nie eingeschaltet werden (kontradiktorische Verwendung), die Funktion „OUTPUT2“ nie ausgeschaltet, wenn diese einmal aktiviert worden ist. Die Programmelemente müssen nicht notwendigerweise in der im Beispiel angenommenen Anordnung verwendet worden sein.

Fig. 27: Testprogramm (Quasi-Tautologie oder Kontradiktion)

Neben SPS-Programmelementen, die offensichtlich im Programm tautologisch bzw. kontradiktorisch kombiniert sind, können Variablen Werte zugewiesen werden, die immer den Wert logisch **EINS** oder den Wert logisch **NULL** einnehmen, also wie Konstante zu behandeln sind. In der Regel ist ein Programmierfehler die Ursache. In ungünstigen Fällen tritt der Fehler nicht sofort auf, sondern erst dann, wenn programmbedingt ein bestimmter Zustand in einer Anlage oder Steuerung erreicht worden ist. Fig. 27 zeigt zwei derartige Beispiele.

Ein Analysewerkzeug wird derartige Programmkonstruktionen nicht unmittelbar entdecken. Im Beispielprogramm Fig. 28 ist die gleiche Befehlskombination verwendet worden wie zuvor. Im Gegensatz zu oben ist diese allerdings als so genannte „Selbsthaltung“ konzipiert und kann deshalb funktionell korrekt sein.



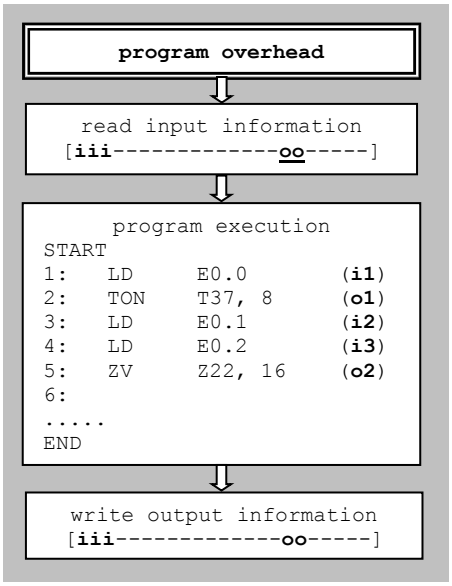
Funktion: Die Funktion „OUTPUT2“ kann über die logische Verknüpfungsoperation „U INPUT1“ auch wieder ausgeschaltet werden. Die Programmelemente müssen nicht notwendigerweise in der im Beispiel angenommenen Anordnung verwendet worden sein.

Fig. 28: Testprogramm (Selbsthaltung)

Als Selbsthaltung wird in diesem Zusammenhang jene Befehlsfolge verstanden, die den zugewiesenen Wert einer Variablen zur Ansteuerung dieser, also für sich selbst, verwendet. Ist der Wert der betrachteten Variablen logisch **EINS** zugewiesen worden, übernimmt die in der Befehlskette programmierte Anweisung die Selbsthaltung und damit die Ansteuerung auch dann noch, wenn das ursprüngliche Ansteuerereignis bereits weggefallen ist (im Beispiel die Befehlszeile … **O „OUTPUT2“**).

4.5 Sonderfunktionen

Sonderfunktionen können als eine Art von Makrobefehlen betrachtet werden. Im so genannten „Programm-Overhead“ werden generelle Steuerfunktionen kontrolliert, die auch von den Sonderfunktionen genutzt werden. Fig. 29 zeigt diese Erweiterung.



Funktion: Die Funktion „`TON T37, 8`“ stellt eine einschaltverzögerte Zeitfunktion mit der Adresse T37 dar. In dieser Programmiersprachversion ist die Zeitbasis mit 100 ms fix gegeben, der Multiplikator dazu ist laut Programm 8, was bedeutet, dass nach dem Einschalten der ansteuernden Funktion E0.0 eine Zeitdauer von 800 ms verstreichen muss, bis die Schaltfunktion von T37 ein HIGH Signal erreicht. Die Funktion „`ZV Z22, 16`“ beschreibt einen Aufwärtszähler mit der Adresse Z22. Seine Vorwahl ist hier mit 16 festgelegt, das heißt, dass nach 16-maligem Betätigen der ansteuernden Funktion E0.1 die Schaltfunktion von Z22 ein HIGH Signal erreicht. Rücksetzbar ist der Zähler durch die Funktion E0.2.

Fig. 29: Testprogramm (Zeiten, Zähler, ähnlich **`STEP 7`**)

Der Programm-Overhead wird vor jedem Programmdurchlauf (Zyklus) durchlaufen. Im Wesentlichen finden dort generelle Überprüfungen wie Status der SPS (RUN, STOP, FAIL), Zykluszeitüberwachung (WATCH_DOG_TIMER) und weitere statt. Dort wird auch das Taktsignal für die Zeiten (Zeittaktgenerator) erzeugt.

In der Folge werden die Sonderfunktionen Zeiten bzw. Zähler näher betrachtet. Den Ausschnitt eines entsprechenden Programms zeigt Fig. 29. Eine Zeitfunktion kann als ein Zähler betrachtet werden, die von einem Zeittaktgenerator erzeugte Zeitimpulse (im gegebenen Fall) inkrementiert. Die hier vorgesehen Schaltschwelle - gegeben als Festwert 8 - bedeutet, dass bei Erreichen und danach bei Überschreiten der zugehörige Funktionswert als „Zeit abgelaufen“ gesetzt wird. Die Rückstellung auf den Anfangszustand erfolgt, wenn die Ansteuerung (**`E0.0`**) wegfällt. Das bedeutet, dass für diese Funktion zwei Funktionswerte vom Typ **`o`** (output) bereitgestellt sein müssen, einerseits für den aktuellen Zählwert, andererseits für seinen Zustand nach dem Zeitablauf.

Sehr ähnlich arbeitet der dargestellte Zähler, im gegebenen Fall ein Aufwärtszähler. Beim Aufwärtszähler sind allerdings zwei Funktionswerte vom Typ **i** (input) notwendig: das Zählereignis selbst (**E0.1**) und die Möglichkeit, den Ausgangszustand durch Rückstellen des Zählwerts wieder zu erreichen (**E0.2**). In diesem Beispiel können, abhängig vom SPS-Programm, diese Funktionswerte (**E0.1**, **E0.2**) auch durch Befehlsketten ersetzt sein. Dieser Umstand bringt es mit sich, dass bei der semantischen Programmanalyse einzelne Befehle oder Befehlsketten erkannt werden, denen Zuweisungen (die Befehlsempfänger) scheinbar fehlen.[32]

Zusammenfassend lässt sich feststellen, dass in dieser Art der Anwendung die Programmelemente Werte vom Typ **o** (output) in zweifacher Hinsicht nutzen. Einerseits werden dort aktuelle Zählerwerte gespeichert, andererseits Zustände, die beschreiben, ob ein aktueller Wert kleiner oder größer/gleich einer Vorwahl ist. Wie das Makroprogramm innerhalb der Sonderfunktion arbeitet, ist nicht offen gelegt (black box), nur die Auswirkungen der Bearbeitung sind bekannt und daher für die Untersuchung zugänglich.

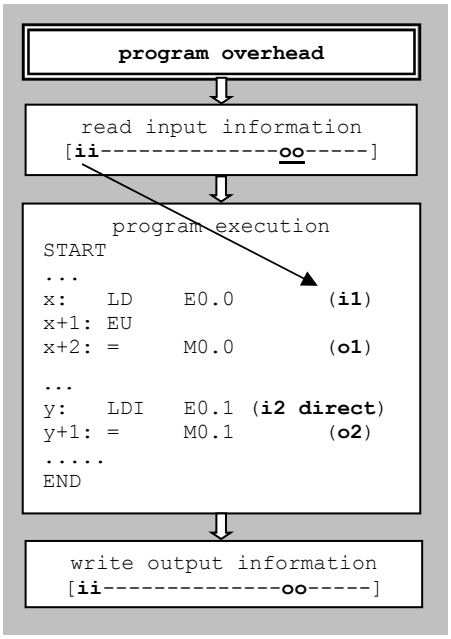
In der Regel sind Sonderfunktionen an geeigneten Stellen im SPS-Programm angeordnet, nicht notwendigerweise am Anfang oder am Ende. Das bedeutet, dass Programmteile existieren können, in denen der „alte“ Wert einer Sonderfunktion vor der Aktualisierung genutzt wird, in nachfolgenden Programmteilen ein „neuer“ Wert derselben Funktion, der sich auf Grund des Kontrolleingriffs durch den Programm-Overhead beim Bearbeiten des Programmelements auf Grund der stattgefundenen Aktualisierung verändert hat. Dieser Umstand bewirkt unterschiedliche Wertzuweisungen innerhalb eines SPS-Programmdurchlaufes für Variable. Beispiel: eine Zeitfunktion sei zunächst noch nicht abgelaufen, nach ihrer Aktualisierung schon. Das bedeutet, dass die generelle Regel des Festhaltens der Anfangssituation (read input information) für Sonderfunktionen durchbrochen sein kann.

Als Vorbereitung für die Verifikation bieten sich grundsätzlich folgende Möglichkeiten an: Mit Hilfe der semantischen Analyse wird festgestellt, an welcher Stelle des SPS-Programms die Sonderfunktion liegt. Weiters wird die Position der weiteren Elemente, die die Ergebnisse der Sonderfunktion nutzen, festgestellt. Liegen alle diese weiteren Elemente, entweder vor oder nach dieser Position im SPS-Programm, sind keine Auswirkungen auf das verwendete

[32] Syntaktisch gesehen sind „offene“ Befehlsketten in AWL grundsätzlich kein Fehler, in anderen Sprachversionen allerdings fallweise schon und diese werden bereits dort zurückgewiesen. Offene Befehlsketten in AWL bewirken einfach nichts, schaden in der Regel auch nicht, da nur Programmelemente die Funktionswerte vom Typ i (input) nutzen, verarbeitet werden. Bei der Programmanalyse von Sonderfunktionen wie beim Zähler lässt sich also erst bei der Untersuchung der Zuweisungsfunktion feststellen, ob die Anordnung der Befehle in einem Programm auch funktionell notwendig ist oder nicht.

Konzept für die Verifikation zu erwarten. Ist das nicht der Fall, muss für die Verifikation eine zusätzliche Variable eingeführt und abgefragt werden, die die Werte der Elemente vor dem Aufruf der Sonderfunktion und nach dem Aufruf der Sonderfunktion unterscheidet.

Neben den hier beispielhaft angeführten Sonderfunktionen (Zeiten und Zähler) zählen Flankenauswertungen und so genannte „immediate" Programmelemente zu den Sonderfunktionen. Diese sind in Fig. 30 dargestellt. Die Flankenauswertung erfolgt ab jener Stelle im SPS-Programm, ab der diese erstmalig angesteuert worden ist und ihr Wert ist für die Dauer eines kompletten Zyklus gültig. Unterschieden werden „steigende Flanken" (**EU**) und „fallende Flanken" (**ED**). Die immediate Funktion erfasst den Funktionswert beim Aufruf direkt vom physikalischen Eingang, wobei das durch den Block read input information abgebildete Prozessabbild nicht aktualisiert wird. Die zugehörige Information wird unmittelbar, immediate, verarbeitet. Ein Teil der syntaktischen Programmanalyse muss daher auch sein, die Sonderfunktionen geordnet auszuweisen. Die klassische Querverweisliste bietet diese Möglichkeit nicht.



Funktion: Der Funktionswert vom **M 0.0** ist ab der Programmzeile **x+2** gültig und besitzt den Flankenwert des ersten Eingangs **E 0.0 (i1)**. Dieser Wert ist auch im nachfolgenden Zyklus gültig, jedoch nur bis zur Programmzeile **x+1** (danach wird der Wert wieder zurückgesetzt). Der Eingang **E 0.1** wird nicht aus dem Funktionsblock „*read input information*" abgelesen, sondern direkt vom physikalischen Anschluss an der SPS.

Fig. 30: Testprogramm (Flanke, immediate, ähnlich **STEP 7**)

Die Problematik bei der semantischen Programmanalyse liegt bei der Flankenauswertung darin, dass Flanken nur einmalig in Abhängigkeit von der Änderung des Funktionswerts der ansteuernden Funktion ausgewertet werden. Auch hier kann, abhängig vom SPS-Programm, der Funktionswert (**E0.0**) auch durch eine Befehlskette repräsentiert sein. Aus diesem Grund ist es für die spätere Analyse notwendig, dem Befehl zur Flankenbestimmung (**EU**) eine Vari-

able zuzuweisen. Ist dies, wie im gegebenen Beispiel, nicht bereits vorgesehen, muss das als Vorbereitung für die Verifikation bei der Analyse vorgesehen werden. Eine Schwierigkeit dabei ist, dass jede Flankenbestimmung einen genau ihr zugeordneten Wert vom Typ `o` (output) benötigt und dass diese Variablen darüber hinaus verwaltet werden müssen. Der Umstand, dass im Beispiel eine direkte Zuweisung nach dieser Operation (`M0.0`) erfolgt, reicht nicht aus, da der Wert von `M0.0` an anderer Stelle im Programm weiterbearbeitet und damit verändert werden kann. Andere Programmierkonzepte sehen im Gegensatz zum hier vorliegenden Beispiel bereits einen direkt zugänglichen Wert vom Typ `o` (output) vor.

Vergleichsweise unproblematisch dazu sind die „immediate“ Programmelemente. Die Gültigkeit der ihnen zugeordneten Werte beschränkt sich auf genau jene Programmstelle im SPS-Programm, an der sie bearbeitet werden. Der im Beispiel `E0.1` zugeordnete Wert wird nicht dem Block „read input information“ entnommen, sondern direkt vom gerade aktuellen Zustand an den Anschlussklemmen der Hardware. Die Konsequenz davon ist, dass an den „normal“ im SPS-Programm abgefragten Stellen Werte verschieden zu den immediate abgefragten sein können.

4.6 Weitere Überprüfung

Als weitere semantische Überprüfung eines SPS-Programms können so genannte „Ausschluss-Kriterien“ festgelegt werden. Darunter können semantisch widersinnige Programmkonstruktionen fallen, wie etwa das zeitgleiche Ansteuern einander ausschließender Zustände. Überprüfungen dieser Art verlassen bereits das rein Formale, denn dazu muss auch die ursprüngliche Aufgabenstellung mit in die Untersuchung aufgenommen werden. Der Aufwand dafür ist daher relativ hoch, denn mögliche derartige Kombinationen von SPS-Programmelementen müssen vorher definiert worden sein. Dazu kommt noch, dass in einer Befehlskette die Befehlsreihenfolge nicht immer eine Rolle spielt und dass dadurch eine Vielfalt logisch korrekter SPS-Programmvarianten existieren kann, wodurch das Auffinden solcher Befehlsfolgen im SPS-Programm für jede Auswertung einige Male wiederholt werden muss.

4.6.1 Kombination von Programmdurchläufen (Zyklen)

Die Modellvorstellung, einzelne SPS-Programmdurchläufe zu betrachten, also ausgehend vom Eingangsblock „read input information“ zuletzt die aktualisierte Version des Blocks „write output information“ zu erhalten, ist für alle Eingangsblöcke angewendet auf ein SPS-Programm berechenbar. Naheliegend erscheint es, sicherheitshalber zu fordern, dass bei einer Verifikation auch Kombinationen von SPS-Programmdurchläufen näher zu untersuchen sind.

Wird m als die Summe der Wertebereiche der Variablen vom Typ **i** und vom Typ **o** bezeichnet, sind 2^m SPS-Programmdurchläufe zu untersuchen. Wird jeder Durchlauf mit einem weiteren kombiniert, die paarweise untersucht werden, sind $2^{m!}$ Untersuchungen notwendig. Hier stellt sich die Frage, ob die paarweise Untersuchung zusätzliche Erkenntnisse liefert oder nicht, wenn alle Kombinationen zulässig sind.

Dazu ist zunächst zu fragen, wie die Veränderung der Variablenwerte im Eingangsblock erfolgt, denn es muss ja gelten, dass die ermittelte Ausgangskonfiguration der Werte in die Anfangskonfiguration der folgenden Bearbeitung übernommen wird. Das folgt bereits aus der Festlegung des gewählten Modells für die Bearbeitung und schränkt daher die Anzahl der möglichen Paare ein. Auf Grund der Eindeutigkeit des Ergebnisses kommt nur eine einzige Ausgangskonfiguration der Variablenwerte für die nächstfolgende Berechnung in Frage.

Ob die Verifikation von einzelnen SPS-Programmdurchläufen bei Sonderfunktionen wie z.B. bei der Zeitfunktion ausreichend ist, wird hier kurz angeschnitten. Bekannt ist, dass z.B. der Zeitzähler im Laufe des Programmablaufs zu einem bestimmten Zeitpunkt nach seiner Ansteuerung seinen (End-) Wert erreichen wird. Die Abfrage dazu ist neben dem zu akkumulierenden Zählerstand ein Wert vom Typ **o** im Block „read input information“, der den aktuellen Zustand der Funktion bereitstellt. Dieser Wert wird bei der Abfrage genau zum richtigen Zeitpunkt in der Bearbeitungskette aktualisiert und steht dann der „restlichen“ Überprüfung zur Verfügung. Ob nun die weitere Überprüfung der im Programm davor angeordneten Abfragen dieser Funktion unmittelbar im nächsten Verifikationsschritt oder zu einem beliebigen Zeitpunkt erfolgt, ist unerheblich. Ist der Startwert im Block „read input information“ beispielsweise bei einer Flankenabfrage als gesetzt gekennzeichnet, wird dieser bei der Bearbeitung der SPS-Programmstelle beim Aufruf als für die weiteren Programmschritte als zurückgesetzt markiert. Weitere Details werden bei der Verifikation diskutiert.

Damit sind nur noch Variationen der Eingangskonfiguration mit den vorher ermittelten Werten für die neue Ausgabe zu berechnen. Diese Berechnungen haben jedoch bei den Einzelberechnungen der Verifikation bereits stattgefunden und müssen daher auch nicht neuerlich überprüft werden.

4.6.2 Kreuz-Korrelation

Interessant wäre es jedoch, eine Art von Kreuz-Korrelation zwischen abgrenzbaren Programm-Modulen eines bereits strukturierten SPS Programms zu definieren, um etwa die gegenseitigen Beeinflussungsmöglichkeiten zu überprüfen. Einflüsse von abgrenzbaren Programm-Modulen wie Modul A auf die Module B, C, D ... und umgekehrt zu untersuchen, muss ein Schwerpunkt der Verifikation sein.

Die Beeinflussung kann nur über gemeinsam genutzte Speicherbereiche erfolgen. Bei der semantischen Analyse wird diese Möglichkeit bereits untersucht. Die Syntax der SPS-Programmierung lässt es allerdings auch zu, dieselben Speicherbereiche oder Teile davon unterschiedlich zu bezeichnen. Die Bezeichnungen beispielsweise M 0.0, MB 0, MW 0 und MD 0 beschreiben überdeckende Bereiche, die bei der rein syntaktischen Untersuchung unterschieden werden. Fig. 31 zeigt Beispiele für Überdeckungen bei der Adressierung von Speicherbereichen.

M 0.0 $\in$ {**MB 0, MW 0, MD 0**};
MB 0 $\in$ {**MW 0, MD 0**};
MW 0 $\in$ {**MD 0**};
aber auch z.B.:
M 3.7 $\in$ {**MD 0**};
oder
MB 1 $\in$ {**MW 0,MD 0**}

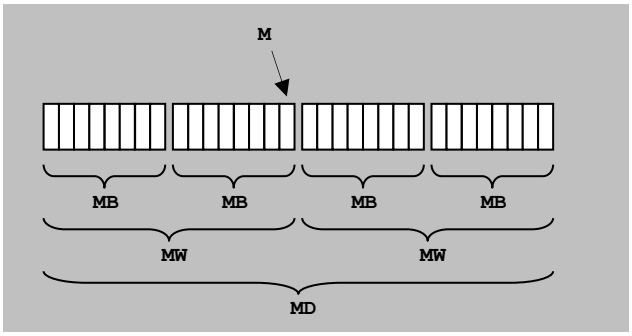


Fig. 31: Adressierung (ähnlich **STEP 7**)

Die Regeln für die Adressierungen in einer SPS ähneln einander und sind in den herstellerspezifischen Programmierumgebungen festgelegt.

4.7 Zusammenfassung

Problematisch für die Verifikation eines SPS-Programms können nur die Sonderfunktionen werden, weil diese entweder eine besondere Behandlung oder zusätzliche Untersuchungen erfordern. Die Frage ist, ob bei der semantischen Programmanalyse der Einfluss des Blocks „program overhead“ gegeben ist oder nicht.

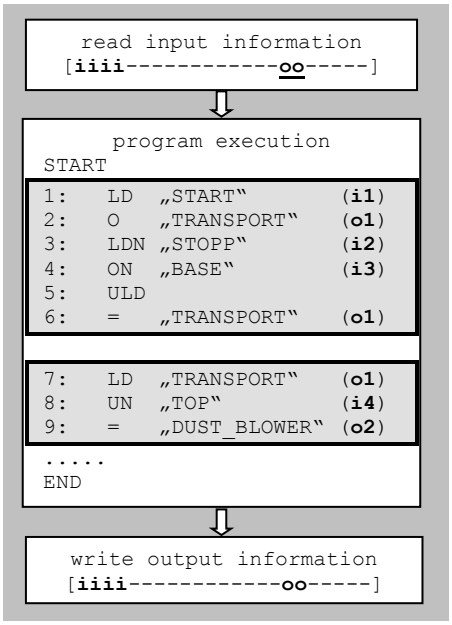
Völlig unproblematisch sind z.B. Zeitfunktionen dann, wenn in der Folge der SPS-Befehle zuerst die Zuweisung der Zeitfunktion erfolgt und die Zeitauswertung selbst erst danach. In einem derartigen Fall ist der Funktionswert „Zeit abgelaufen“ jedenfalls für alle Abfragen identisch. Das Gleiche gilt im umgekehrten Fall, zuerst alle Abfragen und danach die Zuweisung. Im Fall der Betrachtung eines bestimmten SPS-Programmzyklus wird weder der Stand des Zeitzählers noch sein Vorwahlwert benötigt, die Auswertung beschränkt sich wie bei den übrigen Programmelementen auf seinen aktuellen Wert.

Können bei der semantischen Programmanalyse keine formalen Kriterien gefunden werden, mit deren Hilfe die Verifikation vereinfacht werden kann, muss die Anzahl der Verifikationsschritte entsprechend erhöht werden. Umso günstiger für die Einengung des Suchraums bei

der Verifikation wird es auch sein, geeignete SPS-Programmabschnitte voneinander zu isolieren und getrennt zu untersuchen.

4.8 Rückwärtsanalyse

Die Rückwärtsanalyse betrachtet die Ausgangsinformationen. Ausgehend von der Zuweisung wird untersucht, welche Programmelemente eines SPS-Programms ein vordefiniertes Ergebnis bestimmen. Damit wird festgelegt, wie Funktionen oder Ketten von Funktionen ihre Werte erreichen. Da der rein physikalische Anschluss der Eingänge eine Rolle für die Eingangsinformation spielt, wird der Einfachheit halber festgelegt, dass alle Geber als so genannte Schließer ausgeführt sind.[33]



Bohrmaschine: Funktion wie zuvor (Fig. 23). Untersucht wird der Zustand der Ausgänge (**TRANSPORT** und **DUST_BLOWER**). Die Fragestellung lautet, welche Eingangsinformation ist notwendig, um den Ausgang (**DUST_BLOWER**) zu aktivieren.

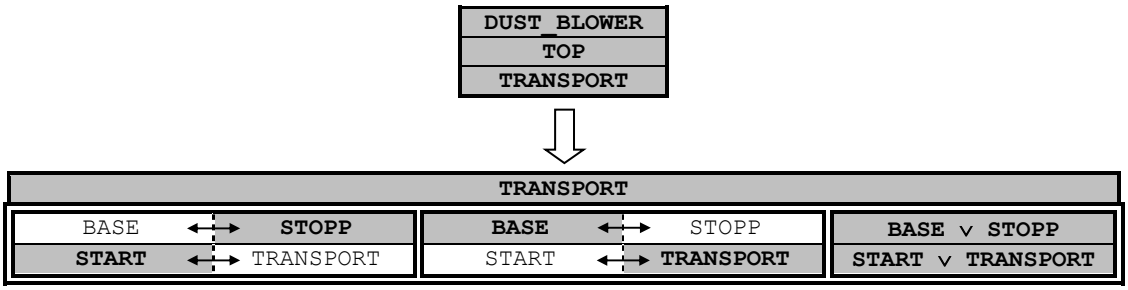
Fig. 32: Analyse der Funktion Bohrmaschine (ähnlich **STEP 7**)

Dazu sind zuerst die Variablenwerte von Interesse. Im Beispiel (Fig. 32) werden Elemente als reine Abfragen also nur lesend (**i** ... input) bzw. schreibend/lesend (**o** ... output) unterschieden. Definitionsgemäß sind die reinen Abfragen im SPS-Programmzyklus konstant und können als Konstante betrachtet werden, während schreibend/lesend deklarierte Elementen Werte

[33] Schließer bedeutet in diesem Zusammenhang, dass, wenn er betätigt wird, „eingeschaltet“ ist und dass sich damit sein zugehöriger Variablenwert als logisch EINS ergibt. Wird dieser Wert durch die Abfrage im SPS-Programm negiert verwendet (gekennzeichnet durch ein N beim Operanden), besitzt er bei Betätigung den Wert logisch NULL.

zugewiesen werden können. Nur Letztere kommen als Variable in Frage und dort auch nur an jenen SPS-Programmstellen, an denen diese zugewiesen werden.

Im folgenden Beispiel werden ausgehend von der Operation **= „DUST_BLOWER"** Schritt für Schritt die weiten Elemente untersucht. Die Operation **UN „TOP"** fragt den negierten Wert von „**TOP**" ab. Diese ist als Eingangsinformation definiert und muss auf Grund der hier vorgesehenen Verknüpfung den Wert „ausgeschaltet" besitzen. Die nächste Operation **LD** „**TRANSPORT**" muss den Wert „eingeschaltet" besitzen. Das Element „**TRANSPORT**" ist entsprechend seiner Deklaration (**o** ... output) ebenfalls als Variable zu betrachten, wenn an anderer Stelle im SPS-Programm sein Wert verändert werden kann. Aus diesem Grund folgt die Untersuchung jenes Bereichs im SPS-Programm ausgehend von der zugehörigen Zuweisungsfunktion von „TRANSPORT". Zu untersuchen sind eigentlich alle möglichen Konstellationen der Werte, im gegebenen Fall werden drei Hauptkonstellationen genauer betrachtet.[34] Fig. 33 zeigt die Analyse in einer graphischen Übersicht. Die Untersuchung ist für diesen Fall abgeschlossen, wenn alle Variablen bestimmt worden sind.



Die Richtung der Untersuchung erfolgt von oben nach unten bzw. von links nach rechts. Im SPS-Programm bedeutet das beginnend von den höheren Adressen zu den niedrigeren. Zuerst wird daher der Zustand des Ausgangs (**DUST_BLOWER**) untersucht. An der Stelle der Untersuchung des Elements mit der Bezeichnung „**TRANSPORT**" muss die Untersuchung verzweigt weitergeführt werden. Hervorgehobene Elemente besitzen in der Darstellung den Funktionswert logisch **EINS**, die anderen „ausgeschaltet" (logisch **NULL**). Achtung: Der Funktionswert „eingeschaltet" bedeutet, dass bei invertierten Funktionswerten der Zustand etwa einer Eingangsinformation den Wert logisch **NULL** besitzen muss. Ausgehend von der Zuweisung **= „TRANSPORT"** folgt der Befehl **ULD**. Mit dieser Anweisung werden zwei im SPS-Programm vorher angeordnete Befehlsketten logisch **UND** verknüpft. Aus diesem Grund muss jede dieser Befehlsketten den Wert logisch **EINS** besitzen, wenn die Anfrage erfüllt sein soll. Die Kombination **BASE** oder **STOPP** ist nur dann logisch **EINS,** wenn entweder **BASE** oder **STOPP** oder beide Werte logisch **NULL** sind. Da sowohl **BASE** bzw. **STOPP** als invertierte Funktionswerte abgefragt werden, muss daher wenigstens einer dieser Werte den Zustand „ausgeschaltet" besitzen. Analog ist die Vorgehensweise bei der zweiten Befehlskette.

Fig. 33: Analyseschritte

In Fig. 33 ist zu erkennen, dass in allen drei Rubriken entsprechend den drei Möglichkeiten passende Werte für die Variablen zugewiesen sind (die hervorgehobene Darstellung symbolisiert den Funktionswert für logisch **EINS** der Abfrage). Im Beispiel musste das Verfahren

[34] Unter Hauptkonstellationen werden hier typische Betriebsfälle verstanden: (1) aus dem Stillstand der Maschine wird die Starttaste betätigt, (2) während des Bohrvorgangs wird die Stopptaste betätigt, (3) die Starttaste wird einige Zeit festgehalten (Abwarten einer Reaktion der gesteuerten Maschine).

zweistufig durchgeführt werden, weil in der Vorstufe Variable (hier das Element „**TRANSPORT**“) vorkommen. Die Anzahl der Untersuchungen ist gegeben durch die Anzahl jener Elemente, die innerhalb des SPS-Programms veränderbar sind. Wird die Fragestellung umgekehrt, etwa weshalb eine Funktionalität des SPS-Programms nicht erfüllt wird, können gezielt Ursachen dafür angegeben werden.

4.9 Rückwärtsanalyse als Verifikationswerkzeug

Die Rückwärtsanalyse liefert Hinweise für den Programmierer bzw. Nutzer der SPS, ob etwa eine gewünschte Funktionalität der Steuerung erfüllt wird oder nicht. Dieser Umstand kann auch zur Überprüfung des SPS Programms vor der Inbetriebsetzung von Geräten oder Anlagen genutzt werden, einerseits um die Erreichbarkeit gewollter Zustände zu verifizieren, andererseits auch, um bestimmte Reaktionen der SPS auszuschließen. Dazu werden Konstellationen von Ausgangsinformationen definiert, die nie eintreten dürfen, weil durch sie eine Fehlreaktion bedingt wäre.

Für diese Art der Vorgehensweise spricht, dass im Wesentlichen die Ausgangszustände alleine das Verhalten einer Steuerung nach außen widerspiegeln, während Eingangszustände und die internen Variablen nur indirekt, auf Grund der Berechnungen beim Programmdurchlauf, beteiligt sind. Damit kann die Anzahl der Untersuchungen stark eingeschränkt werden. Nachteil der Rückwärtsanalyse ist, dass es zu jeder Konfiguration von Ausgangszuständen mehrere Konfigurationen von Eingangszuständen und Zuständen der internen Variablen gibt.

Hier kann unterschieden werden: bei der vollständigen Rückwärtsanalyse müssen alle möglichen Ausgangskombinationen untersucht werden, bei der Gefährdungsanalyse kann man sich auf Zustände beschränken, die zu unerwünschten oder gefährlichen Reaktionen der gesteuerten Einheit führen.

In Bezug auf die beteiligten Programmelemente ergibt sich eine neue Suchstrategie. Ausgangspunkt ist die, vom Ende des SPS-Programms gesehen, erste Zuweisungsoperation. Innerhalb jener Befehlsfolge, die der betrachteten Zuweisung zugeordnet ist, wird unterschieden zwischen bereits fest zugewiesenen Werten, diese entsprechen den im Eingangsabbild festgelegten, und Variablen, deren Werte an anderen Stellen im Programm selbst erst ermittelt werden. Bei den zugewiesenen Variablen muss nochmals unterschieden werden. Bei einer absoluten direkten Zuweisung (z.B. Operator „**=**“) gilt das unmittelbar vorher berechnete Ergebnis, der Wert steht fest und im Programmablauf zeitlich früher ermittelte Werte sind nicht relevant. Bei den Setz- bzw. Rücksetzbefehlen (z.B. Operatoren „**S**“ und „**R**“) muss zusätzlich untersucht werden, ob diese Operationen aktiv Werte liefern, also im betrachteten Programmabschnitt den Wert bestimmen oder nicht.

Die Suche ist beendet, wenn alle beteiligten Elemente identifiziert worden sind. Im folgenden Bearbeitungsschritt wird untersucht, unter welchen Vorbedingungen eine untersuchte Variable den vordefinierten Wert annehmen kann. Ergebnis dieser Untersuchungen sind die möglichen Eingangskonfigurationen.

Bei der anschließenden Gefährdungsanalyse werden Ausschlusskriterien gesucht. Die vorher ermittelten Eingangskonfigurationen werden in Relation gebracht. Es wird dabei beispielsweise für zwei Variable A und B, die unterschiedliche Werte aufweisen müssen, untersucht, ob der geforderte Unterschied bei der Wertzuweisung für alle gefundenen Eingangskonfigurationen sichergestellt werden kann.

Das Ergebnis einer vollständigen Rückwärtsanalyse betrachtet alle Ausgänge einer SPS. Nach der Ermittlung der Eingangsinformation für jede Kombination von Ausgängen sind die für eine Verifikation erforderlichen Vorbedingungen bekannt, die zu den zugrunde gelegten Ausgangsinformationen zuzuordnen sind. An dieser Stelle muss neuerlich mit Hilfe eines Filtermechanismus festgestellt werden, ob damit auch alle weiteren Varianten möglicher Kombinationen der Eingangsinformationen berücksichtigt worden sind. Sind alle erfasst, ist die Vollständigkeit der Überprüfung sichergestellt.

4.10 Schlussbemerkungen

Die klassische, rein syntaktische Programmanalyse ist notwendig, reicht aber nicht aus. Das Zerlegen eines SPS-Programms auf die kleinste, verifizierbare Einheit (ein Netzwerk) gibt zwar Auskunft über ein Detail, ist aber aus dem Zusammenhang gerissen nicht bewertbar. Dazu muss auch die Bedeutung des Programms mit in die Untersuchung einbezogen werden. Die Verifikation erfolgt in der Regel nicht durch den Programmierer einer SPS. Gründe für die Anordnung von Befehlsfolgen im SPS-Programm und der daraus folgenden Bedeutung werden bei der Verifikation selbst nicht untersucht. Die hier diskutierten Ansätze zeigen Wege, wie durch rein formale Untersuchungen Befehlsgruppen identifizierbar sind, die in Interaktion stehen und damit in ihrer Semantik auch zusammenhängende Programmteile darstellen.

Durch die semantische Programmanalyse lassen sich geeignete SPS-Programmabschnitte so voneinander abgrenzen, dass die Verifikation dadurch zumindest erleichtert wird. Ergänzt durch die Rückwärtsanalyse können auch Hinweise für den Programmierer bzw. Nutzer der SPS generiert werden, um die gewünschte Funktionalität der Steuerung sicher zu stellen.

5 Erweiterte Programmanalyse

In diesem Kapitel sind Aufgabenstellungen für die Voruntersuchung von SPS-Programmen definiert. Interaktive Anfragen analysieren SPS-Programme bereits bei der Erstellung und erleichtern die Verifikation.

5.1 Voranalyse

Logische Fehler in einem SPS-Anwenderprogramm ergeben sich aus der unrichtigen Verwendung von Speicheradressen. Eingangsvariablen sind vom Typ *read only*, könnten jedoch auch im Programm zugewiesen worden sein. Vielfach wird dies bereits von den Programmeditoren ausgeschlossen, sollte jedoch überprüft werden. Funktionslos sind jedoch Programmelemente vom Typ *read write*, die im Programm in logischen Verknüpfungen verwendet worden sind aber nicht zugewiesen werden. Außer bei Ausgängen sind auch Elemente, die nur zugewiesen aber sonst im Programm nicht abgefragt werden, in der Regel ohne Bedeutung und können ohne Verlust der vom Programmierer geplanten Funktionalität entfernt werden. Die Bearbeitung erfolgt schrittweise:

Schritt 1: Laden des Quellprogramms

Nach dem Laden des Quellprogramms wird eine Liste erzeugt bzw. das System durchsucht, ob eine bereits vorhandene Liste verwendet werden kann, in der die Ergebnisse der Untersuchung festgehalten werden.

Schritt 2: Durchsuchen des Quellprogramms

Im Durchsuchungsschritt werden die gefundenen Strukturen ausgewertet. Dazu werden Programmelemente entsprechend ihres Typs geprüft. Zu beachten dabei ist, dass es keine feste Reihenfolge bezüglich der Verwendung einzelner Programmelemente in Bezug auf die Beziehung Abfrage – Zuweisung gibt.

Schritt 3: Ergebnisse speichern

Die Ergebnisse der Untersuchung werden dokumentiert. Für die interaktive Nutzung kann eine Bildschirmausgabe vorgesehen werden.

Allgemein gilt, dass ein vorher in den Speicher geladenes SPS-Quellprogramm in einzelne Zeilen zerlegt werden muss und von allfällig vorhandenen Kommentaren befreit wird, da innerhalb von Kommentaren auch syntaktisch richtig bezeichnete Funktionselemente enthalten sein können, die jedoch für das zu untersuchende Programm keine Relevanz besitzen. Gesucht werden „Muster“, die auf Konsistenz zu überprüfen sind. Festgestellte logische Fehler

werden mit der zugehörigen Zeilennummer und der SPS-Anweisung in einer Fehlerdatei festgehalten.

Im typischen Standardablauf werden zuerst die Kommentare aus allen Anweisungszeilen entfernt. Zur weiteren Vereinfachung und Vereinheitlichung werden auch überflüssige Leerzeichen entfernt. Danach wird Zeile für Zeile nach jenen „Mustern“ gesucht, die den zu überprüfenden Funktionen entsprechen. Wird ein Muster erkannt, erfolgt der Abgleich mit dem Speichermodell der SPS. Bei Unstimmigkeit wird die entsprechende Fehlermeldung der Fehlerliste hinzugefügt.

5.2 Aufgaben

Ausgangspunkt der Untersuchung ist ein SPS-Programm, das in der Programmiersprache AWL vorliegt. Dieses soll automatisiert analysiert werden. Der Fokus der Untersuchung liegt bei Zusammenhängen, die aus dem SPS-Programm abgelesen werden können. Die Programmanalyse soll so allgemein gehalten sein, dass sie für SPS-Programme unterschiedlicher Hersteller durchgeführt werden kann.

Bei SPS-Steuerungen werden standardmäßig Funktionen wie das Suchen verwendeter Programmelemente angeboten, ergänzt durch Filtermechanismen zur gezielten Sortierung der Suchergebnisse, die in ihrer Art limitiert sind. Die hier gestellten Anforderungen sind im Wesentlichen auf das Suchen der verwendeten Programmelemente beschränkt, allerdings sind weitere Suchkriterien und andere Randbedingungen zu berücksichtigen.

Gewonnen wird das Quellprogramm für die Untersuchung aus dem AWL-Code eines SPS-Editors. Da der Quellcode in der Regel nicht direkt als Text zur Verfügung gestellt werden kann, muss als Alternative der Code zuerst als Programmdokumentation in ein Textdokument ausgedruckt werden. Die typisch in derartigen Dokumenten vorhandenen Kommentierungen können gegebenenfalls für Kommentierungen im Analyseergebnis oder zur Steuerung des Analysators genutzt werden. [35] Die Form ist eine Liste der Anweisungen, geordnet nach der Reihenfolge der Bearbeitung, z.B.:

Position (im Programm)	Befehl (Operator)	Adresse (Operand)	ev. Kommentar

Die Position einer Anweisung im Programm bestimmt die Reihenfolge der Bearbeitung. Für die Position genügt die aufsteigende Nummerierung. Diese spiegelt die Reihenfolge der Bear-

[35] Als strukturierte Kommentierungselemente bieten manche Programmeditoren etwa die Nummerierung von Netzwerken oder Trennzeichen zwischen diesen an.

beitung der SPS-Befehle wider. Die Bezeichnung der Befehle und der Befehlssatz können von Steuerung zu Steuerung differieren, sind aber wegen der zu Grunde liegenden Norm (IEC 61131-3) ähnlich.[36] Die Adressen sind gekennzeichnet durch Kennbuchstaben und Speicherpositionen. Fakultativ können auch allfällig vorhandene Programmkommentierungen in die Liste mit aufgenommen werden. Diese könnten die spätere Auswertung unterstützen, werden aber für die Analyse nicht benötigt.

Auf Grund differierender Befehlsbezeichnungen und Bezeichnungen der Adressen ist eine Bibliotheksfunktion notwendig, in der die korrekte Syntax für die zu analysierende SPS hinterlegt ist. Vor Beginn der Analyse wird die passende Bibliothek dem SPS-Programm zugeordnet. Als Funktionalität sind dort den Befehlen Bearbeitungsvorschriften zugeordnet, soweit diese für die semantische Untersuchung erforderlich werden. Für die Kalkulation der Programmlaufzeit können dort auch Zeitangaben für die Befehlsausführungszeiten optional enthalten sein. Dieser Teil wird später noch genauer definiert.

Netzwerk Nr.			**ev. Kommentar**
1	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
2	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
...	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
k	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
Netzwerk Nr.			**ev. Kommentar**
k+1	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
k+2	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
...	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
m	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
Netzwerk Nr.			**ev. Kommentar**
m+1	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
m+2	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
...	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
n	**Befehl (Operator)**	**Adresse (Operand)**	**ev. Kommentar**
	END		**Programm Ende**

Fig. 34: Struktur eines SPS-Programms in AWL

[36] Beispiel: der Befehl zum Laden einer Variablen wird mit der Kennung „**LD**“, „**L**“ bzw. „**LAD**“ durch verschiedene Programmeditoren bezeichnet.

Neben der Position im Programm existiert auch bei nahezu allen SPS eine Organisations-Struktur. Typisch ist dabei die Gliederung in Funktionen, Netzwerke und Unterprogramme. Im ersten Ansatz wird hauptsächlich auf das Strukturelement Netzwerk näher eingegangen.[37] In der Regel ist beim Wechsel von einem zum folgenden Netzwerk keine Bearbeitungszeit erforderlich. Der Netzwerkwechsel soll aber als Markierung in die Befehlsfolge mit aufgenommen werden. Der Quellcode eines SPS-Programms soll für die weitere Bearbeitung in folgender Form zur Verfügung stehen oder in diese Form gebracht werden (angenommen sind mehrere Netzwerke mit zusammen n SPS-Befehlen, Fig. 34).

5.3 Rückwärtsanalyse

Ausgehend von der Struktur (Fig. 34) kann ein bestimmtes Netzwerk unmittelbar definiert werden, z.B. das Netzwerk mit der Befehlsfolge k+1 … m. Im Beispiel sei mit m jene Befehlszeile bezeichnet, deren Ergebnis die bis zu dieser Stelle durchgeführten Berechnungen für die zugehörige Adresse jenes Ausgangsfeldes enthält, die einen interessierenden Ausgabewert bestimmt. Für die nachfolgende Logik liegt ab dieser Programmposition der Ausgabewert endgültig fest und kann innerhalb der gerade ablaufenden Verarbeitung nicht mehr verändert werden. Verfolgt man die Entwicklungsschritte wieder in Richtung Programmanfang, also vom zuletzt ausgeführten Befehl der AWL ausgehend gezählt, kann die schrittweise Berechnung des Ausgabewerts rückverfolgt werden. Daher die Namensgebung Rückwärtsanalyse.

Die Rückwärtsanalyse definiert die Entstehungsgeschichte des betrachteten Werts. Liegen in der aktuell betrachteten Befehlsfolge Zwischenergebnisse, die im SPS-Programm zeitlich vorher ermittelt worden sind, müssen die zugehörigen Netzwerke mit niedrigerer Nummerierung ebenfalls auf die gleiche Art untersucht werden. Diese Art der Untersuchung muss solange fortgesetzt werden, bis als Fixpunkte nur mehr die Anfangswerte[38] für die Bestimmung des untersuchten Ausgangswerts zu Grunde liegen.

Fig. 35 verdeutlicht das Prinzip der Rückwärtsanalyse. In der Skizze sind im oberen Bereich die Zustände der Eingangsinformationen als Eingangsabbild dargestellt. Für die Analyse werden die dort festgehaltenen Werte als Fixpunkte (invariant) gesehen. Der betrachtete Programmausschnitt ist vertikal dargestellt. In der Realität müssen Befehle, die sich beeinflussen können, nicht notwendigerweise aufeinander folgen. Programmelemente, die mit den Eingangswerten operieren, sind durch die Pfeile verbunden. Es existieren im Beispiel zwei Zwi-

[37] Als Netzwerk wird eine Programmstruktur bezeichnet, die mit einer Abfrage beginnt, eine Anzahl von Verknüpfungsbefehlen aufweist und mit wenigstens einer Zuweisung endet.

[38] Als Anfangswerte gelten festgelegte Werte für jeden Zyklus. Sie sind im Eingangsabbild als Festwerte für den gerade betrachteten Zyklus invariant abgelegt.

schenergebnisse, deren aktuellen Zustände übertragen werden und in die Bestimmung des Endwerts einfließen. Die Ausgabe des berechneten Werts erfolgt im Ausgangsabbild. Es wurde dabei angenommen, dass der betroffene Ausgangswert im weiteren Programmverlauf innerhalb des gerade betrachteten Zyklus nicht mehr verändert wird. Die Pfeile in Fig. 35 symbolisieren Abhängigkeiten der betrachteten Elemente, die der Untersuchungsrichtung entgegengesetzt gerichtet sind.

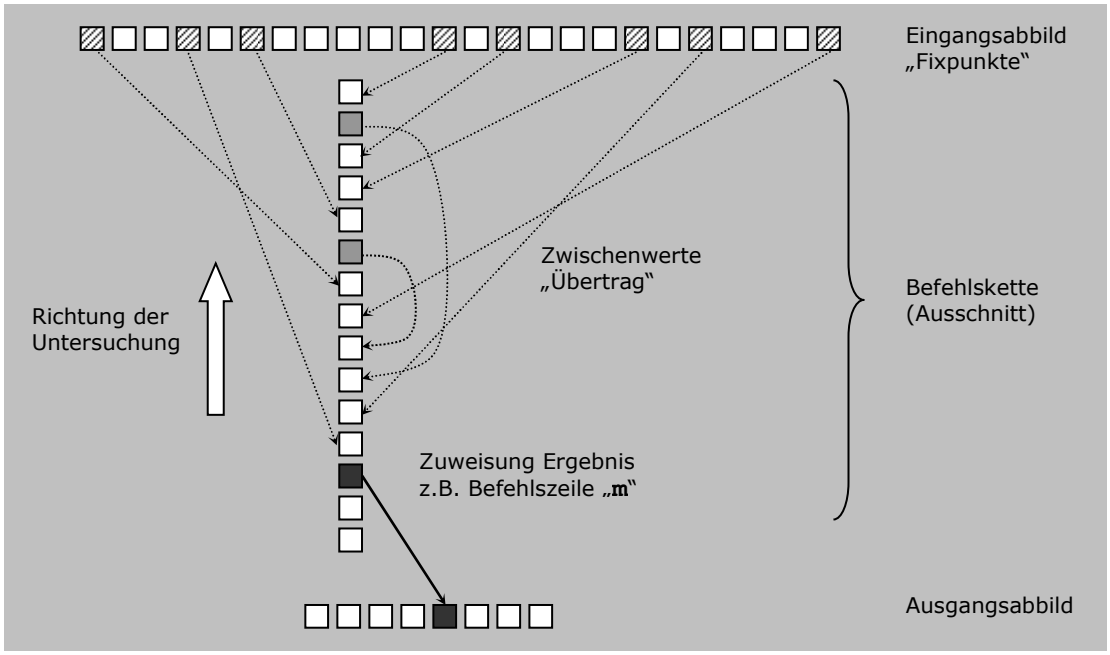


Fig. 35: Prinzip der Rückwärtsanalyse

Betrachtet wird im folgenden Beispiel eine ausgewählte Ausgangsinformation des Ausgangsabbildes. Für die Rückwärtsanalyse ergeben sich damit folgende Anforderungen:

5.3.1 Fall 1: Untersuchung bestimmter Ausgänge

Gesucht wird ein bestimmter Ausgangswert bei einem vorgegebenen Eingangsabbild (Einzelanalyse).

Schritt 0: Festlegen des gesuchten Ausgangswertes

Schritt 1: Lokalisieren der erzeugenden Anweisung.

Ein Ausgangswert kann nur durch die Zuweisungsoperationen im Programm modifiziert werden. Operationen zur Modifikation eines Ausgangswerts können mehrfach im Programm enthalten sein. Es existieren Anweisungen, die unbedingt ausgeführt werden (bei Zuweisungen z.B. =) und solche, die nur bedingt im Sinne einer mehrfach zu-

lässigen Modifikation ausführbar sind (bei Zuweisungen z.B. S, R). Entsprechend der Arbeitsweise einer SPS gilt für die Ausgabe immer das Ergebnis der zuletzt vorgenommen Veränderung.

Schritt 2: Feststellen der beteiligten Operationen.

Typisch für Zuweisungen ist, dass diese am Ende der Struktur „Netzwerk“ anzutreffen sind. Jedenfalls tragen alle vor der Zuweisung angeordneten Operationen zur Wertermittlung bei, Ergebnisse könnten jedoch im Zuge des Programmfortschritts verworfen werden. An dieser Stelle soll nur das letztgültige Ergebnis beeinflussenden Operationen herausgefiltert und der weiteren Analyse zugeführt werden.

Schritt 3: Ermittlung der relevanten Ergebnisse.

Ausgehend von der Anweisung wird die Ermittlung des Ergebnisses rückverfolgt. Grundsätzlich ist es möglich, in einer baumartigen Struktur die einzelnen Blätter (bzw. die dort für die Berechnung vermerkten interessierenden Ausgangselemente) zu erreichen, den entsprechenden Wert abzulesen und den Weg durch das SPS-Programm zurück bis zum Ergebnis, dem ursprünglich untersuchten Ausgabewert, zu verfolgen. Wenigstens für die Dauer eines Prüfzyklus bleiben die Werte im Eingangsabbild unverändert und gelten als Fixpunkte.

Alternative: Verzweigung (Zwischenschritte „z“).
Existieren laut Schritt 3 Zwischenwerte, deren Ergebnisse von Berechnungsvorschriften im SPS-Programm abhängen, wird die gerade laufende Untersuchung unterbrochen, um die benötigten Werte festzustellen.

Zur Ermittlung des Zwischenergebnisses erfolgt im Schritt 1z die Identifikation, im Schritt 2z werden dafür benötigte Operationen gesucht und im Schritt 3z ausgewertet. (Anmerkung: Die Untersuchung von Verzweigungen kann auch mehrfach erforderlich werden.)

Danach wird der unterbrochene Schritt 3 weitergeführt.

Schritt 4: Die Werte für alle beteiligten Operationen sind definiert.

In der Ausgabe kann im Einzelnen begründet werden, warum genau das gefundene Ergebnis erzielt worden ist.

5.3.2 Fall 2: Vollständige Einzelanalyse

Die Einzelanalyse wird auf alle möglichen Variationen des Eingangsabbildes für die dort betroffenen Werte der zugehörigen Variablen ausgeweitet. Die Ausgabe erfolgt sortiert nach den

Ausgabewerten, d.h. Eingangsabbilder, die denselben Ausgabewert erzeugen, sind in Gruppen geordnet.

Bei der vollständigen Einzelanalyse reicht es aus, sich auf die Variation des Eingangsabbilds zu beschränken, wenn in der Kette der untersuchten Befehlsfolge ausschließlich unbedingte Zuweisungsoperationen enthalten sind. Bei bedingten Zuweisungen müssen gegebenenfalls weitere Befehlsketten in die Berechnungen mit einbezogen werden.

5.3.3 Fall 3: Untersuchung mehrerer Ausgänge

Gesucht wird eine Gruppe bestimmter Ausgangswerte bei beliebigen Eingangsabbildern. Es wird gefragt, welche Werte die Ausgänge bei den nacheinander durchgeführten Einzeluntersuchungen erreichen (Mehrfachanalyse). Die Ausgabe erfolgt sortiert nach den Ausgabewerten. Die Vorgehensweise erfolgt wie zuvor bei der Einzelanalyse, erweitert auf mehrere Ausgangswerte.

5.3.4 Fall 4: Verträglichkeit

Gesucht wird innerhalb einer Gruppe bestimmter Ausgangswerte bei beliebigen Eingangsabbildern, ob bestimmte Ausgänge innerhalb eines Untersuchungsschrittes zeitgleich ausgegeben werden (Mehrfachanalyse). Die Ausgabe dieser Untersuchung sind jene Eingangsabbilder, die zu derartigen, für den Untersuchungsfall unerwünschten, Kombinationen von Ausgangswerten geführt hatten, sortiert nach den Ausgabewerten. Die Vorgehensweise erfolgt wie zuvor bei der Einzelanalyse, erweitert auf Ausgangswerte, die nie zeitgleich erzeugt werden dürfen.

5.3.5 Fall 5: Kombinationen

Gesucht wird innerhalb einer Gruppe bestimmter Ausgangswerte, welche Eingangsabbilder zur gefragten Kombination von Ausgabewerten führen. Mit dieser Untersuchung können interessierende Zustände eines SPS-Programms untersucht werden. Die Vorgehensweise erfolgt wie zuvor bei der Einzelanalyse, wobei neben der Verträglichkeitsuntersuchung auch zeitgleich erzeugte Ausgangswerte gesucht werden.

5.3.6 Fall 6: Folgeuntersuchung

Die Folgeuntersuchung nutzt Ergebnisse der Suche nach Kombinationen (Fall 5). Dazu werden die gefundenen Eingangsabbilder variiert, indem der Teilbereich für die Eingänge im Eingangsabbild (read only) als konstant übernommen und mit dem gefundenen Ausgangs-

abbild zusammengesetzt wird. Mit dem so konstruierten Eingangsabbild, es stellt den unmittelbaren Folgezustand im Programmablauf einer SPS dar[39], wird ein weiteres Ausgangsabbild erzeugt. Gefragt wird nach den Unterschieden. Mit dem Ergebnis dieser Untersuchung kann ein „Stabilitätsindikator“ für SPS-Programme definiert werden.

5.3.7 Fall 7: Innere Struktur von Netzwerken

Die innere Struktur eines SPS-Netzwerkes besteht immer aus einer Abfrage am Netzwerkanfang, einer Reihe von Verknüpfungen und einer Zuweisungsoperation am Ende des Netzwerkes. Es ist in verschiedenen Programmiersprachen jedoch auch zulässig, bereits innerhalb der Befehlskette Zuweisungen vorzunehmen und mit den Zuweisungsergebnissen die Befehlskette weiter bis zu einer „finalen“ Zuweisungsoperation weiterzuführen. Bei dieser Analyse sollen derartige Strukturen lokalisiert werden.

5.3.8 Fall 8: Zusammenhänge in Programmstrukturen

Ziel der Untersuchung ist, Programmbereiche zu identifizieren, die nur beschränkt voneinander abhängen. Die Abhängigkeit wird durch eine Kontrollgröße vorgegeben, z.B. als Anzahl, wie viele Variable von verschiedenen Programmteilen gemeinsam genutzt werden dürfen.

Schritt 1: Gemeinsam verwendete Elemente

Ausgangspunkt ist die Struktur Netzwerk. Jedem Netzwerk sind Elemente im Eingangsabbild und im Ausgangsabbild zuordenbar. Gefragt wird nach Zusammenhängen bei der Verwendung von Ein- und Ausgangselementen, inwieweit dieselben Adressen unterschiedlichen Netzwerken zugeordnet sind. Eine voreinstellbare Kontrollgröße „Unterschied“ als Filterfunktion gibt die Anzahl jener Adressen vor, die sich von Netzwerk zu Netzwerk unterscheiden dürfen.

Schritt 2: Bestimmen der Elemente in den Abbildern

Im folgenden Schritt werden die gefundenen Strukturen ausgewertet. Dazu werden gefundene Programmabschnitte und die benutzen Elemente im Eingangs- und im Ausgangsabbild im Analysebericht angezeigt.

[39] Mit der Einschränkung, dass der Zustand der Eingänge innerhalb des betrachteten Ablaufs unverändert geblieben ist. Das ist nahezu immer der Fall.

Zu allen Punkten existieren gemeinsame Anforderungen:

Bei Datenelementen bedeutet „sortiert“ immer, dass alle Eingänge, Ausgänge und Merker zusammengefasst werden und innerhalb der jeweiligen Gruppen nach aufsteigenden Wort-, Byte- und Bitwerten dargestellt werden.

Analog zu den Befehlen, die von Hersteller zu Hersteller leicht variieren können, sind auch verschiedene Formen für die Art und Weise zu berücksichtigen, wie Adressen, also Eingänge, Ausgänge, Merker, Timer usw. im Quellprogramm dargestellt werden können. Diese können auch in der Form von symbolischen Namen vorkommen, für die dann eine entsprechende Assoziationsliste vorliegt.

Alle Ergebnisse sollen interaktiv auf dem Bildschirm dargestellt werden können, aber alternativ auch als Ausdruck auf Papier oder in eine Datei möglich sein.

5.4 Ausschluss von Zuständen

Die Rückwärtsanalyse definiert die Entstehungsgeschichte betrachteter Werte, wobei in nachfolgenden Bearbeitungsschritten untersucht wird, unter welchen Vorbedingungen eine Variable vordefinierte Werte annehmen kann. Ergebnis dieser Untersuchungen sind mögliche Eingangskonfigurationen für die Verifikation. Andererseits können Wertpaare definiert werden, die entsprechend den Anforderungen an das SPS-Programm in Relation gebracht werden, etwa, dass dann und nur dann ein von der SPS gesteuerter Anlagenteil in Betrieb sein darf wenn ein anderer Anlagenteil in einer bestimmten Position verharrt. Aus der Aufgabenstellung lassen sich daher für das Programm Konfigurationen von Zuständen bestimmen, die beispielsweise gemeinsam erreicht werden müssen oder einander ausschließen. Werden derartige Bedingungen verletzt, treten unverträgliche Zustände oder Zustandskombinationen auf.

Derartige Bedingungen lassen sich vielfach einfacher definieren als vollständige Wertepaare der Ein- Ausgangsvektoren, die für die Verifikation benötigt werden.

Für unverträgliche Zustände und Zustandskombinationen werden bei einer anschließenden „Gefährdungsanalyse“ Ausschlusskriterien gesucht, die mit den vorher ermittelten Eingangskonfigurationen in Relation gebracht werden. Das Ergebnis einer vollständigen Rückwärtsanalyse betrachtet alle Ausgänge einer SPS, die Gefährdungsanalyse schränkt den Suchraum ein. Nachteilig ist, dass in Bezug auf die funktionale Sicherheit das SPS-Programm verbessert werden kann, dass jedoch die Untersuchung unvollständig ist.

5.5 Zusammenfassung

Unter Zugrundelegung derzeitiger Rechnersysteme kann eine Verfikation mit etwa 30 Variablen als obere Schranke durchgeführt werden. Eine weitere Grenze bildet die Überschaubarkeit innerer Zusammenhänge der Ein- und Ausgangsvektoren, weil diese vom Programmierer als Prämissen der Verifikation vorherbestimmt werden müssen. Diese Grenze kann durch Segmentierung der SPS-Programme zusammen mit Untersuchung von Programmüberdeckung für gemeinschaftlich verwendete Variable in verschiedenen Programmteilen überschritten werden. Dazu sind jedoch noch weitere Untersuchungen notwendig.

Die Rückwärtsanalyse betrachtet die Ausgangsinformationen. Ausgehend von der Zuweisung wird untersucht, welche Programmelemente eines SPS-Programms ein vordefiniertes Ergebnis beeinflussen. Damit wird bestimmt, wie Funktionen oder Ketten von Funktionen bestimmte Werte ermitteln. Sie liefert damit Hinweise für den Programmierer, ob etwa eine gewünschte Funktionalität der Steuerung erfüllt wird oder nicht.

Damit kann eine Überprüfung eines SPS-Programms bereits vor der Inbetriebsetzung von Geräten oder Anlagen erfolgen, einerseits um die Erreichbarkeit gewollter Zustände zu verifizieren, andererseits auch, um bestimmte Reaktionen der SPS auszuschließen. Konstellationen von Ausgangsinformationen werden vom Programmierer definiert, die laut der Aufgabenstellung nie eintreten dürfen (Ausschlussbedingungen, Fehlreaktionen).

6 Formale Darstellung der SPS

In diesem Kapitel wird dargelegt, welche Eigenschaften von SPS-Programmen für die Verifikation verwendet werden.

6.1 Mathematisches Modell

Ein SPS-Programm ist definiert durch die lineare Folge seiner Kommandos. Die Kommandos bewirken Zustandstransformationen. Entsprechend ihrer linearen Reihenfolge werden die Kommandos auf die als variabel deklarierten Teile des Eingangsvektors angewendet und deren Zustand entsprechend des Programmfortschritts variiert. Nach dem Start des SPS-Programms wird die Kommandofolge ständig wiederholt.

Die Eingänge werden vor der Programmbearbeitung gelesen. Für die Dauer der folgenden Programmbearbeitung werden die Zustände der Eingänge zwischengespeichert und deren Werte sind für diesen Zeitraum konstant. Zusätzlich werden Werte von den genau im vorhergehenden Berechnungsdurchlauf ermittelten Zuständen von Ausgangsinformationen und Zwischenwerte benötigt, die im Fortschritt der Programmbearbeitung entsprechend bearbeitet werden. Zuletzt werden die ermittelten Werte der Zwischenergebnisse und der Ausgänge für den nächsten Berechnungsdurchlauf bereitgestellt und die Ausgangszustände ausgegeben.

6.2 Formale Beschreibung

Ein SPS-Programm beinhaltet eine Reihe von Kommandos. Kommandos bestehen aus einem Befehlsteil und Operanden. Operanden gliedern sich in den Operator mit einer zugehörigen Adresse. Operanden bezeichnen mit Hilfe dieser Adressen Speicherbereiche, in denen die Zustandsinformationen bereitgestellt sind. Diese Informationen können je nach Typ des Operanden für die Verarbeitung nur lesend oder lesend und schreibend genutzt werden. Nur lesend werden Eingänge genutzt. Eingänge erfassen die externen Signalzustände der Steuerung. Ihre Ausgänge schalten entsprechend dem Ergebnis der Programmbearbeitung extern angeschlossene Betriebsmittel. Interne Variable wie die so genannten Merker sind interne Ausgänge und im Verhalten identisch mit den externen Ausgängen. Ihr Zustand kann nicht direkt von der extern angeschlossenen Peripherie abgelesen werden. Das Gleiche gilt für Daten und weitere Variable.

Der typische Aufbau eines Kommandos besteht aus dem Befehl C_{INSTR}, dem Operator C_{OP} und einer Adresse C_{ADR}. Ein Kommando ist die Kombination von Befehl, Operator und Adresse, wobei je nach Kommando auch einzelne Elemente fehlen können:

$$C \underset{def}{=} C_{\text{INSTR}} \times C_{\text{OP}} \times C_{\text{ADR}}$$

Die Bezeichnung der Operatoren bezieht sich in der Regel auf ihre Funktion in der SPS. Typische Bezeichnungen sind E für Eingänge, A für Ausgänge, M für Merker (Memory) usw.

$$C_{\text{OP}} \underset{def}{=} <\texttt{E,A,M,...}>$$

Jeweils am Beginn eines Bearbeitungszyklus wird die Eingangsinformation von den Eingängen der SPS abgelesen und im Wertebereich der Variablen aktualisiert. Dieser Bereich wird konstant gehalten, also für die Dauer der innerhalb eines Programmzyklus folgenden Berechnungsschritte nicht verändert.

Die Eingangs- und Ausgangsinformationen der SPS sind in Variablen abgebildet. Die Ein-Ausgabe-Variablen V sind definiert als $\text{V} \underset{def}{=} < \tilde{S}, S >$. Dabei sind $\tilde{S}$ initiale Variablen.

Es gilt:

$$\tilde{S} \underset{def}{=} < \tilde{s}_0, \tilde{s}_1, \tilde{s}_2, ..., \tilde{s}_{k-1}, \tilde{s}_k >, \quad S \underset{def}{=} < s_0, s_1, s_2, ..., s_{k-1}, s_k >$$

Die Position der Variablen innerhalb von V wird durch ihre Adresse bestimmt (ADR). Die Variablen werden auf Zustände abgebildet. Die Abbildung der Werte auf {0,1} entspricht den logischen Werten für „falsch“ und „wahr“.

Kommandos C entsprechen dem Instruktionssatz für SPS sind in Anlehnung an die IEC 1131-3 definiert. Die in der Aufzählung enthaltenen Identifier entsprechen den Bezeichnungen für Kommandos in Anweisungsliste (AWL).

$$C_{\text{INSTR}} \underset{def}{=} <\texttt{LD,LDN,U,UN,O,ON,U(,O(,),JMP,=,S,R,...}>$$

Tab. 1 beschreibt exemplarisch die Funktion einiger Befehle.

Befehl C_{INSTR}	Funktion
LD	Stellt einen Variablenzustand zur weiteren Verarbeitung bereit.
U	Verknüpft einen Variablenzustand logisch **UND** mit dem vorher berechneten Wert.
U (	Ein vorher berechneter Wert soll logisch **UND** mit dem Klammerausdruck verknüpft werden.
)	Das durch den Klammerausdruck bestimmte Zwischenergebnis wird mit dem vor der Klammer berechneten Wert verknüpft.
=	Das zuletzt bestimmte Zwischenergebnis wird zugewiesen.
JMP	Die folgenden Schritte werden vorwärtsgerichtet übersprungen.

Tab. 1: Zuordnung Befehl/Funktion

Das Programm P ist definiert als eine Folge von Kommandos C:

$$\mathrm{P} \underset{def}{=} < c_0, c_1, c_2, ..., c_{n-1}, c_n >$$

Dabei bezeichnen die Indices die Position des Kommandos im Programm und damit gleichzeitig die Reihenfolge der Bearbeitung.

Jedem Kommando ist ein Zustand Z in der SPS zugeordnet. Zusätzlich werden für die interne Verarbeitung und Zwischenspeicherung von Teilergebnissen Variable benötigt. Für diese sind zusätzliche Zustände Z^+ definiert.

$$\mathrm{Z} \underset{def}{=} < z_0, z_1, z_2, ..., z_{n-1}, z_n, \mathrm{Z}^+ >$$

Tab. 2 definiert die Wirkung der Kommandos auf Zustände. Mit dem Zusatz (ADR) wird die Position einer Variablen in V bezeichnet. Die Kommandos sind Zustandstransformatoren, die einzelne Zustände nach einer Berechnungsvorschrift verändern.

Kommando $C \underset{def}{=} C_{\text{INSTR}} \times \ldots \times C_{\text{ADR}}$	Wirkung
LD (ADR)	Der Variablenzustand von $V_{\texttt{(ADR)}}$ wird in Z übertragen.
U (ADR)	Der Variablenzustand von $V_{\texttt{(ADR)}}$ wird logisch **UND** mit dem vorher berechneten Wert verknüpft und in Z übertragen
U(	Der vorher berechnete Wert und das Kommando **U** werden in Z^+ zwischengespeichert.
)	Das durch den Klammerausdruck bestimmte Zwischenergebnis wird entsprechend dem in Z^+ gespeicherten Kommando C mit dem in Z^+ ebenfalls gespeichertem Wert verknüpft und in Z übertragen.
= (ADR)	Das vorher bestimmte Zwischenergebnis wird in Z gespeichert und in $V_{\texttt{(ADR)}}$ übertragen.
JMP (n)	Die folgenden **(n)** Schritte werden vorwärtsgerichtet übersprungen. Ein bis zu diesem Kommando ermitteltes Zwischenergebnis wird in Z^+ zwischengespeichert und gegebenenfalls mit dem im Sprungziel definierten Kommando verknüpft und in Z übertragen.

Tab. 2: Zuordnung Kommando/Wirkung

Beispiele: Vereinfacht wird in den folgenden Beispielen die Zustandstransformation für einige Kommandos erläutert. Für den Wert einer Variablen aus V ist die Adresse als Zahlenwert angegeben. Da die Position eines Kommandos innerhalb des SPS-Programms definiert durch den Index des Kommandos von Bedeutung für die Bearbeitung ist, wurden diese festgelegt. Die nachfolgenden Kommandos sind frei gewählt und hängen nicht zusammen. Die Bedeutung der Indices liegt daher darin, dass dadurch die Positionen der Veränderungen in den Variablen V und Z festgelegt werden.

- Kommando c_7: **LD (5)** bedeutet: der 5. Wert aus V (s_4) wird in Z an die Stelle z_7 übertragen.
  Es gilt: $s_4 \rightarrow z_7$
- Kommando c_{12}: **U (8)** bedeutet: der 8. Wert aus V (s_7) wird mit dem zuletzt berechneten Wert aus Z (z_{11}) logisch **UND** verknüpft und in Z an die Stelle z_{12} übertragen.
  Es gilt: $z_{11} \wedge s_7 \rightarrow z_{12}$
- Kommando c_{17}: **U(** bedeutet: der zuletzt berechnete Wert aus Z (z_{16}) und der Operator **U** für logisch **UND** werden in Z^+ zwischengespeichert.
  Es gilt: $\texttt{U} \rightarrow \boxed{\wedge^{[Z^+]}}$, $z_{16} \rightarrow \boxed{z_{16}^{[Z^+]}}$
- Kommando c_{20}: **)** (hier ist vorausgesetzt, dass kein weiterer Klammerausdruck dazwischen geschachtelt ist und dass die schließende Klammer dem Kommando c_{17} zugeordnet ist. Das Zwischenergebnis des Klammerausdrucks wird durch die Ausführung des

Kommandos c_{19} bestimmt, das Ergebnis ist daher in z_{19} gespeichert.) Das Kommando bedeutet dann, dass der Wert aus Z (z_{19}) mit dem in Z^+ zwischengespeicherten Wert (entspricht dem Wert von z_{16}) auf Grund des definierten Kommandos (**U**) logisch **UND** verknüpft und in Z an die Stelle z_{20} übertragen wird.

Es gilt: $z_{19}\ \boxed{\wedge^{[Z^+]}}\ \boxed{z_{16}^{[Z^+]}} \rightarrow z_{20}$

- Kommando c_{30} : **JMP (4)** bedeutet, dass das Programm mit dem Kommando c_{34} fortgesetzt wird.

Als Konfiguration K wird das Wertepaar bezeichnet, die das noch abzuarbeitende Programm mit seinen Zuständen beschreibt. Die Abarbeitung des Programms erfolgt nach der Abarbeitungsrelation, die Konfigurationen schrittweise verknüpft. Eine Konfiguration ist definiert:

$\mathrm{K} \underset{def}{=} (< \mathrm{P} >\ ,\ [\mathrm{P'}] < \mathrm{Z,V} >)$, mit P' als den bereits abgearbeiteten Kommandos.

Für die Startkonfiguration gilt dann:

$$\mathrm{K} \underset{def}{=} (< c_0, c_1, c_2, ..., c_{n-1}, c_n >\ ,\ [\ ] < \mathrm{Z,V} >)$$

Wurde ein Kommando abgearbeitet, sind auch die Zustände entsprechend verändert worden. Für das Kommando c_i wird die Zustandstransformation im Folgenden mit $< \mathrm{Z}^{i-1}, \mathrm{V}^{i-1} > \xrightarrow{c_i} < \mathrm{Z}^{i}, \mathrm{V}^{i} >$ dargestellt.[40]

Für die Abarbeitung eines Zyklus des SPS-Programms gilt, dass beginnend mit der Startkonfiguration die Abarbeitungsrelation auf die jeweils zuletzt bearbeitete Konfiguration angewendet wird, solange, bis alle Kommandos ausgeführt worden sind.

$$\begin{aligned}
&(< c_0, c_1, .., c_{n-1}, c_n >\ ,\ [\ ] < \mathrm{Z,V} >) && \rightarrow (< c_1, ..., c_{n-1}, c_n >\ ,\ [c_0] < \mathrm{Z}^0, \mathrm{V}^0 >),\\
&(< c_1, .., c_{n-1}, c_n >\ ,\ [c_0] < \mathrm{Z}^0, \mathrm{V}^0 >) && \rightarrow (< c_2, .., c_{n-1}, c_n >\ ,\ [c_0, c_1] < \mathrm{Z}^1, \mathrm{V}^1 >),\\
&... &&\\
&(< c_n >\ ,\ [c_0, c_1, ..., c_{n-1}] < \mathrm{Z}^{n-1}, \mathrm{V}^{n-1} >) && \rightarrow (< >\ ,\ [c_0, c_1, .., c_{n-1}, c_n] < \mathrm{Z}^n, \mathrm{V}^n >).
\end{aligned}$$

Durch die Abarbeitungsrelation wird der Zustand schrittweise weiterentwickelt. Die Ausführung eines Kommandos bewirkt eine Zustandsänderung. Nach der Bearbeitung des Letzen

---

[40] Die Indices wurden hochgestellt. Tiefgestellte Indices wurden reserviert für die Darstellung möglicher Ein- Ausgangsinformationen des PLC, die jeweils zu unterschiedlichen Zustandstransformationen führen.

Kommandos c_n wird der Endzustand Z^n erreicht. Gleichzeitig wird auch der Endzustand der Variablen V^n hergestellt und als Ausgangsinformation ausgegeben.

Bei einem Sprungbefehl wird die gegebene Anzahl von Kommandos (Bearbeitungsschritte) übersprungen. Zur Illustration wird im Beispiel angenommen, dass der Sprungbefehl am Anfang des Programms steht. Einzige Voraussetzung ist, dass das Ende des Programms erreicht werden muss und daher nicht übersprungen werden darf ($x < n$). Dann gilt:

$$(< \mathbf{JMP\ (x)}, c_1, .., c_{x-1}, c_x, .., c_n > \ , \ [\,] < \mathrm{Z,V} >) \rightarrow (< c_x, .., c_n > \ , \ [\mathbf{JMP\ (x)}, c_1, .., c_{x-1}] < \mathrm{Z}^{\mathrm{x\text{-}1}}, \mathrm{V}^{\mathrm{x\text{-}1}} >)$$

Kommandos können in Gruppen eingeteilt werden:

- **Abfragen**. Abfragen sind Kommandos, die aus den Variablen V Werte übertragen. Die Adressinformation **(adr)** gibt dabei die Position des Wertes in V an. Bei einer Abfrage wird der Zustandswerte der Variablen in die Konfiguration K übertragen: $\mathrm{V}_{\mathbf{(ADR)}} \rightarrow \mathrm{K}$. Beispiele für diese Kommandos sind **LD** und **LDN**. Ist das Kommando an der Stelle x eine Abfrage, werden die Ein- Ausgabe-Variablen V nicht verändert. Dann gilt immer: $\mathrm{V}^{x-1} \equiv \mathrm{V}^{x}$.
- **Verknüpfungen**. Verknüpfungen sind Kommandos, die Werte von Variablen berechnen. Dabei wird der Zustandswerte einer Variablen aus K mit dem Wert einer Variablen aus V logisch verknüpft und das Ergebnis in die Konfiguration K übertragen: $\mathrm{K} \otimes \mathrm{V}_{\mathbf{(ADR)}} \rightarrow \mathrm{K}$. Beispiele für diese Kommandos sind **U**, **UN**, **O** und **ON**. Bei Klammerausdrücken wie bei **U(** oder **O(** werden zunächst das Kommando und der zuletzt berechnete Zustandswert als Wertepaar in Z^+ zwischengespeichert $\{\mathrm{K'}, \mathrm{C}\} \rightarrow \mathrm{K}^+$. Folgt in der Bearbeitungsfolge die schließende Klammer, werden K' und K entsprechend des Kommandos C logisch verknüpft und das Ergebnis in die Konfiguration K übertragen: $\mathrm{K'} \otimes \mathrm{K} \rightarrow \mathrm{K}$. Ist das Kommando an der Stelle x eine Verknüpfung, werden die Ein- Ausgabe-Variablen V nicht verändert. Dann gilt immer: $\mathrm{V}^{x-1} \equiv \mathrm{V}^{x}$.
- **Zuweisungen**. Zuweisungen sind Kommandos, die berechnete Zustandswerte den Variablen V entsprechend ihrer Adressinformation zuweisen: $\mathrm{K} \rightarrow \mathrm{V}_{\mathbf{(ADR)}}$. Beispiele für diese Kommandos sind **=**, **S** und **R**. Ist das Kommando an der Stelle x eine Zuweisung, können die Ein- Ausgabe-Variablen V entsprechend dem Berechnungsergebnis verändert werden.

Diskussion

Invariant ist das Programm P, definiert als die Folge von Kommandos $c \in C$. Die Zustandsänderung, die ein Kommando bewirkt, wird einerseits in Z gespeichert, andererseits in V ausgegeben. Dabei werden Ausgabeinformationen und bestimmte, für die weiteren Berechnungen notwendige Zwischeninformationen aus Z in V übertragen. Die Anzahl der Variablen V

ist durch die Peripherie, das sind die als Hardware zur Verfügung gestellten Eingänge und Ausgänge der SPS und durch das Programm bestimmt. Da die Variablen V genau zwei Zustände annehmen können, existieren $2^{\|V\|}$ verschiedene Zustände [41], gleichzusetzen mit verschiedenen, zumindest theoretisch möglichen Startkonfigurationen K. Anders ausgedrückt: es sind im ungünstigsten Fall $m = 2^{\|V\|}$ verschiedene Zyklen eines SPS-zu untersuchen. In der verkürzten Schreibweise gilt:

$$\begin{array}{ccccccccccc}
K_0 & \rightarrow & K_0^0 & \rightarrow & K_0^1 & \rightarrow & ..... & \rightarrow & K_0^{n-1} & \rightarrow & K_0^n \\
K_1 & \rightarrow & K_1^0 & \rightarrow & K_1^1 & \rightarrow & ..... & \rightarrow & K_1^{n-1} & \rightarrow & K_1^n \\
\vdots & & \vdots & & \vdots & & \vdots\vdots & & \vdots & & \vdots \\
K_m & \rightarrow & K_m^0 & \rightarrow & K_m^1 & \rightarrow & ..... & \rightarrow & K_m^{n-1} & \rightarrow & K_m^n
\end{array}$$

Für die späteren Betrachtungen sind dabei die Startkonfigurationen und die zugehörigen Endkonfigurationen von besonderem Interesse.

Diese Methode eignet sich deshalb besonders, weil im Programmfortschritt schrittweise die Verifikationsbedingung entwickelt wird und unmittelbar im Anschluss der Verifikation zugeführt werden kann.

Anmerkung

Es genügt eine einzige nicht verifizierbare Konfiguration, um eine Verifikation zu falsifizieren. In diesem Fall ist es in der Regel notwendig, Veränderungen im SPS-Programm vorzunehmen. Hier stellt sich die Frage, welche Arbeitsschritte zur Verifikation nach der Korrektur zu wiederholen sind oder genauer, welche Auswirkungen eine notwendig gewordene Änderung auf bis zu diesem Zeitpunkt ermittelte Ergebnisse hat.

Durch die Segmentierung des SPS-Programms wird das Programm in Teile zerlegt, die entweder vollständig autark genutzt werden oder derart gruppierbar sind, dass sie zur Erfüllung ihrer Funktion nur eine geringe Anzahl von Informationen untereinander austauschen müssen.

Durch die Kenntnis von Abhängigkeiten zwischen Programmteilen wird einerseits erreicht, dass die Komplexität der als Teilverifikation bezeichneten Arbeitsschritte reduzierbar wird und dass anderseits bei geeigneter Wahl der Programmabschnitte in sich abgeschlossene und damit verifizierte Teilergebnisse ermittelt werden können, die bei allfälligen Korrekturen im Programm nicht wiederholt verifiziert werden müssen.

[41] mit ||V|| wird die Anzahl der Variablen in V bezeichnet

6.3 Konzept zur Teilverifikation

In diesem Abschnitt werden Möglichkeiten untersucht, die dazu dienen um den Aufwand zu verringern. Dazu werden Methoden zum Modularisieren eines SPS-Programms diskutiert und dazu passende Hilfsstrukturen festgelegt. Unter dem Begriff „Teilverifikation“ wird hier verstanden, einzelne SPS-Programmmodule zu verifizieren.

Verifikationsbedingung ist die Beendigung eines Zyklus, also die schrittweise Entwicklung der Konfigurationen $K_i \rightarrow K_i^n$. Angenommen wird nun, dass ein Programm für ein technisches System aus einigen mehr oder weniger abhängigen Programmmodulen besteht, etwa für die Anlagenteile „Maschine 1“ bis „Maschine n“, die voneinander unabhängig arbeiten und zum Austausch von Informationen wenige Signale wie „Start“ und als Rückmeldung zum Arbeitsfortschritt „Fertig“ melden, um etwa in einem Arbeitsablauf den nächsten Bearbeitungsschritt zu starten. Weiters wird angenommen, dass die Reihenfolge der Programmteile der Benennung der Anlagenteile folgt, also, dass das Programm für Maschine 1 zuerst im SPS-Programm bearbeitet wird und dass α Kommandos für diesen Programmteil benötigt worden sind usw.[42] In diesem Fall gilt für die Bearbeitung der Kommandos und der zugehörigen Konfigurationen:

$$K_i \rightarrow K_i^{\alpha} \rightarrow K_i^{\alpha+\beta} \rightarrow K_i^{\alpha+\beta+\gamma} \rightarrow K_i^{\alpha+\beta+\gamma+\delta} \rightarrow ... \rightarrow K_i^n$$

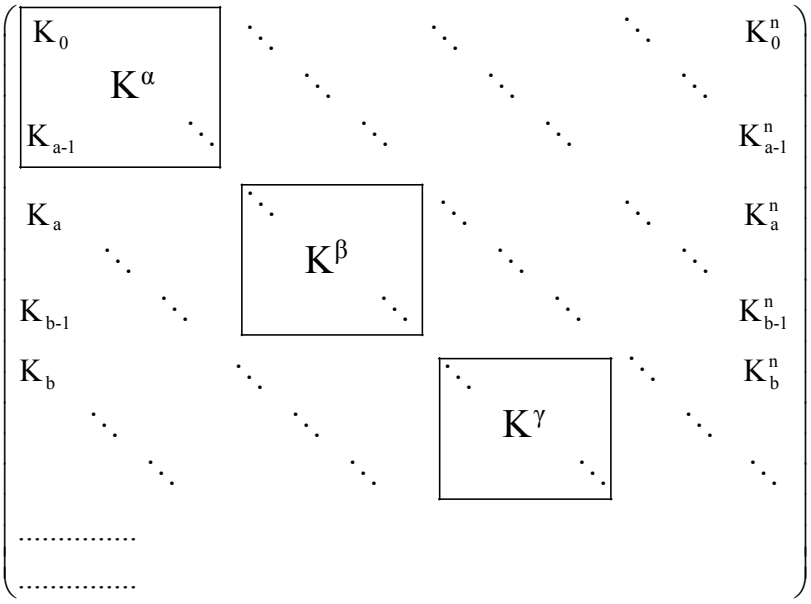
Das SPS-Programm ist strukturiert aufgebaut und unterteilt in Kommandofolgen. Die den Kommandofolgen zugeordneten Konfigurationen werden wie folgt bezeichnet:

$$K^{\alpha} :< c_0, c_1, ..., c_{\alpha-2}, c_{\alpha-1} >,\ K^{\beta} :< c_{\alpha}, c_{\alpha+1}, ..., c_{\beta-2}, c_{\beta-1} >,\ ...$$

Zum Zeitpunkt, wenn die Konfiguration K_i^{α} erreicht ist, ist die Kommandofolge für die Maschine 1 abgearbeitet und alle für diesen Programmabschnitt relevanten Informationen stehen fest. Die zu diesem Programmabschnitt gehörigen Ausgangsinformationen bleiben für die Bearbeitung der weiteren Kommandos konstant, das heißt, sie werden nicht mehr verändert. Das bedeutet auch, dass für die Verifikation des Programms von Maschine 1 unter den gegebenen Voraussetzungen die restliche Bearbeitung der Kommandos irrelevant ist. Der Idealfall ist gegeben, wenn alle Programmteile voneinander unabhängig sind.

---

[42] Ein SPS-Programm kann, von Spezialfällen abgesehen wo die Eigenschaft der zyklischen Bearbeitung für den Programmentwurf genutzt wird, in beliebiger Reihenfolge programmiert sein. Die Gruppierung in funktionell zusammenhängende Programmabschnitte dient in der Regel zur besseren Übersichtlichkeit.



Die markierten Konfigurationen $K^{\alpha}, K^{\beta}, K^{\gamma}, \ldots$ bewirken die Steuerung der Maschinen 1, 2, 3 usw. und die dadurch veränderten Variablen sind damit relevant für die Verifikation. Die nicht markierten Bereiche repräsentieren weitere Befehlsfolgen, die jeweils keinen Einfluss auf die Funktion dieser haben und aus diesem Grund auch nicht weiter untersucht werden müssen.

Aussage: Voneinander völlig unabhängige Programmteile können ohne Einschränkung auf das Gesamtergebnis der Verifikation einzeln verifiziert werden.
Begründung: Die Begründung ergibt sich aus der Forderung, dass Variable, die einem bestimmten Programmteil zugeordnet worden sind, von den davon unabhängigen Programmteilen nicht genutzt oder verändert werden. □

Diskussion

Seien $V(\alpha)$ jene Variablen, die dem Programm von Maschine 1 zugeordnet sind und $a = 2^{\|V(\alpha)\|}$ die Anzahl ihrer Zustände, dann gilt für die Verifikation von Maschine 1, dass die folgende Bedingung erfüllt sein muss.

$$\mathbf{VER} = \bigwedge_{i=0}^{a-1} \{V(\alpha)_i\} \left(K_i \rightarrow K_i^{\alpha}\right) \left\{V(\alpha)_i^{\alpha}\right\}$$

Für die Verifikation des gesamten SPS-Programms müssen alle Teilprogramme verifiziert sein. Die Bedingung dafür lautet:

$$\mathbf{VER} = \left(\bigwedge_{i=0}^{a-1} \{ \mathrm{V}(\alpha)_\mathrm{i} \} \left(\mathrm{K}_\mathrm{i} \rightarrow \mathrm{K}_\mathrm{i}^{\alpha} \right) \{ \mathrm{V}(\alpha)_\mathrm{i}^{\alpha} \} \right) \wedge \left(\bigwedge_{i=a}^{b-1} \{ \mathrm{V}(\beta)_\mathrm{i} \} \left(\mathrm{K}_\mathrm{i}^{\alpha} \rightarrow \mathrm{K}_\mathrm{i}^{\beta} \right) \{ \mathrm{V}(\beta)_\mathrm{i}^{\alpha} \} \right) \wedge \ldots$$

Die Elemente der Operationen können im gegebenen Fall beliebig vertauscht werden, ohne dass sich das Ergebnis verändert. Daraus folgt, dass die einzelnen Programmstücke aber auch Konfigurationen in beliebiger Reihenfolge verifiziert werden können. Liegt eine teilweise Überdeckung von Variablen vor, die sowohl in dem einen und zugleich in weiteren Programmstücken genutzt werden (so genannte Übergabevariablen), dann wird die Überprüfung entsprechend ausgedehnt. Seien x Variable in zwei Programmbereichen wechselweise genutzt, gilt: [43]

$$\mathbf{VER} = \left(\bigwedge_{i=0}^{a-1+2^x} \{ \mathrm{V}(\alpha + x)_\mathrm{i} \} \left(\mathrm{K}_\mathrm{i} \rightarrow \mathrm{K}_\mathrm{i}^{\alpha} \right) \{ \mathrm{V}(\alpha + x)_\mathrm{i}^{\alpha} \} \right) \wedge \left(\bigwedge_{i=a-2^x}^{b-1} \{ \mathrm{V}(\beta + x)_\mathrm{i} \} \left(\mathrm{K}_\mathrm{i}^{\alpha} \rightarrow \mathrm{K}_\mathrm{i}^{\beta} \right) \{ \mathrm{V}(\beta + x)_\mathrm{i}^{\alpha} \} \right) \wedge \ldots$$

Durch das Zerlegen des SPS-Programms in kleinere Einheiten wird der Aufwand für das Verifizieren deutlich geringer.

Die Programmanalyse liefert voneinander unabhängige Programmteile. Diese wurden definiert als eine Teil-Folge von Kommandos. Diese Teil-Folgen sind Abschnitte in einen SPS-Programm, die sich genau eingrenzen lassen.

Bei den Programmabhängigkeiten wurde dabei unterschieden:

- Abhängigkeiten, die Eingangsinformationen betreffen. Zugehörige Informationen werden nur lesend verarbeitet. Aus diesem Grund verändert sich auch die Anzahl der Konfigurationen der untersuchten Teil-Aufgabenstellung nicht.
- Abhängigkeiten, die Ausgangsinformationen betreffen oder intern verwendete Zwischenergebnisse, die so genannten Merker. Zugehörige Informationen werden lesend und schreibend verarbeitet. Werden Programme in Module zerlegt, erfolgt der Informationsaustausch über diese Variablen, die in mehreren Modulen genutzt werden können. Durch die Nutzung der Informationen in mehreren Teil-Programmfolgen verändert sich auch die Anzahl der Konfigurationen der untersuchten Teil-Aufgabenstellungen. Dabei sind als Kennzeichen von Programmabhängigkeiten das Vorhandensein von Variablen Bereichen definiert worden, die von verschieden Programmteilen gemeinsam genutzt werden.
- Als logische Formeln dienen die kleinsten, in sich geschlossenen Programmstrukturen in einem SPS-Programm. Diese werden üblicherweise als Netzwerke bezeichnet. Ein Netz-

[43] unter der Annahme, dass die Bereiche für die Kommunikation von Programmteilen und damit die zugehörigen Kommandos jeweils an die untersuchten Programmbereiche angrenzen, jedoch ohne Einschränkung auf die Allgemeinheit. Damit lässt sich die Formel vereinfacht darstellen.

werk ist für sich verifizierbar, aber erst die Kombination mehrerer Netzwerke zu einer Programmfunktion berücksichtigt auch Abhängigkeiten, die zwischen den aus mehreren Netzwerken bestehenden Programmabschnitten existieren. Netzwerke sind als logische Formeln darstellbar. In der Anfangszeit der Entwicklungsgeschichte von SPS gab es auch Ansätze, Steuerungsprogramme ausschließlich als die Aneinanderreihung von Formeln zu definieren.[44]

6.4 Eigenschaften von Konfigurationen

Prinzipiell sind die Konfigurationen die Abbilder eines Programmzustandes. Ein- bzw. Ausgangskonfigurationen beschreiben die Start- und Endzustände eines Programmdurchlaufs. Eine beliebige Konfiguration K_i^x stellt jenen Zustand dar, der aus der Eingangskonfiguration K_i^0 folgt und nach der Bearbeitung des Kommandos x erreicht worden ist. Damit sind Konfigurationen eindeutig zuordenbar.

Die Menge aller Konfigurationen beschreiben daher alle Zustände, die das untersuchte Programm einnehmen kann. Jeder Konfiguration K_i^x ist genau eine Endkonfiguration nämlich die zugehörige Ausgangskonfiguration K_i^n zugeordnet. Anders ausgedrückt: können von den genannten Bedingungen abweichende Konfigurationen identifiziert werden, muss ein Fehler vorliegen.

Grundsätzlich ist es jedoch zulässig, dass Konfigurationen existieren für die gilt: $K_i^x \equiv K_j^x$ für $i \neq j$. Existieren Konfigurationen, denen im gleichen Bearbeitungsschritt des SPS-Programms identische Inhalte zuordenbar sind, dann gilt auch $K_i^n \equiv K_j^n$. Dieser Zusammenhang ist bereits durch die Definition der Befehlsfolge des SPS-Programms sichergestellt. Das bedeutet, dass die zugeordneten Ausgangskonfigurationen bei der weiteren Abarbeitung zum gleichen Ergebnis geführt haben und dass damit die Ausgangreaktion der SPS identisch ist.

Es wurde eingangs festgelegt, dass in der Eingangskonfiguration bestimmte Bereiche als Festwerte, die als konstant gehaltenen Eingänge der SPS mit der Gültigkeit für die Dauer eines Programmdurchlaufes (Programmzyklus), gesehen werden kann, ein weiterer Bereich als variabel, im Wesentlichen der Bereich interner und externer Ausgänge einer SPS.

---

[44] Programmierkonzept von VEB Robotron, Leipzig in den frühen 80ziger Jahren. Zuerst musste die Zuweisung definiert werden, danach die benötigten Rechenoperationen. Die Programme erforderten einen hohen Speicheraufwand und einen auch für damalige Verhältnisse unverhältnismäßig hohen Zeitaufwand bei der Befehlsbearbeitung. Damit war das Reaktionsvermögen der Steuerung insbesondere bei komplexeren Programmen für viele Anwendungen nicht ausreichend.

Weiters gilt, dass Informationen aus den Eingangskonfigurationen das Programmergebnis nur so lange beeinflussen können, als diese Informationen in der weiteren Programmbearbeitung genutzt werden. Es gilt also, dass ab einem bestimmten Arbeitsschritt in der Abarbeitung der Befehlskette einer SPS nur mehr ein Teil der in der Eingangskonfiguration definierten Informationen genutzt werden. Es wird daher nur mehr ein Teilbereich der in der Eingangskonfiguration gespeicherten Zustände genutzt, deren Anzahl tendenziell abnimmt. Die im weiteren Programmablauf nicht mehr genutzten Informationen können daher das weitere Ergebnis nicht mehr beeinflussen. Bereits auf Grund von Analyseergebnisse kann der entsprechende Programmschritt und damit die zugehörige Konfiguration festgestellt werden.

Im Programmfortschritt eines SPS-Programms ist das immer dann der Fall, wenn die beteiligten Variablen in einer betrachteten Konfiguration durch die weitere Bearbeitung im SPS-Programm nicht mehr verändert werden können, oder wenn die relevanten Teile der Konfiguration bereits an anderer Stelle untersucht worden sind. Für diesen Fall gilt, dass ab diesem Zeitpunkt ein in einem vorher ermittelten Programmdurchlauf bestimmtes Ergebnis in Form der zugehörigen Ausgangskonfiguration bereits bestimmt worden ist und dass in diesem Fall die neuerliche schrittweise Berechnung nicht mehr erfolgen müsste.

Beispiel: Seien $\xrightarrow{Command\ Stucture\ A}$, $\xrightarrow{Command\ Structure\ B}$ und $\xrightarrow{Command\ Structure\ C}$ beliebige Befehlsketten in einen SPS-Programm, wobei $\xrightarrow{Command\ Stucture\ A}$ und $\xrightarrow{Command\ Structure\ B}$ disjunktiv verknüpft sind. Das Ergebnis dieser Verknüpfung soll in weiterer Folge mit $\xrightarrow{Command\ Structure\ C}$ konjunktiv verknüpft werden. Die Ergebnisse der Befehlskette $\xrightarrow{Command\ Structure\ C}$ führen zur wertmäßig exakt gleichen Teilbelegungen von Informationen in den Konfigurationen nach Bearbeitung der Befehlskette $\xrightarrow{Command\ Structure\ C}$, wenn als weitere Voraussetzung die in den Konfigurationen bestimmenden Variablenwerte für die Befehlskette $\xrightarrow{Command\ Structure\ C}$, also die dort definierten Prämissen, übereinstimmen.[45]

Interessant sind für diese Fälle bestimmte Wertebereiche innerhalb von Konfigurationen, die bereichsweise gleich sind. Sie werden als partielle Übereinstimmung bezeichnet, mit Δ als Bereichsbezeichnung. Für den Programmschritt x gilt dann $K_i^x \triangleq K_j^x$. Die Eingrenzung auf interessierende Programmschritte und zugehörige Bereiche kann mittels der weiter oben beschriebenen Methode der Rückwärtsanalyse gefunden werden.[46]

---

45 Bei der standardisierten Abarbeitung der SPS-Programme wird auf derartige Umstände keine Rücksicht genommen und die benötigten Ausgabeergebnisse jedes Mal neu berechnet. Kann jedoch bei der Verifikation auf Teilauswertungen bereits erbrachter Beweisketten zurückgegriffen werden, bedeutet das eine Reduzierung der Komplexität und damit des Aufwandes.

46 Insbesondere bei Konjunktionen von mehr als zwei Programmketten ist eine Einsparung von Beweisschritten beim Verfikationsprozess zu erwarten. Dem steht der Aufwand bei der Voruntersuchung gegenüber.

Insgesamt wird damit beschrieben, dass unterschiedliche Eingangskonfigurationen identische Ausgangsabbilder erzeugen können. Solche Ausgangskonfigurationen werden in Klassen zusammengefasst.

Aussage: Der Begriff Klasse impliziert, dass voneinander verschiedene Konfigurationen existieren, die bezogen auf das Verhalten der SPS identische Ausgangszustände erzeugen können. Das bedeutet, dass, ohne dass das SPS Programm vollständig analysiert werden muss, es Zwischenzustände also Konfigurationen K_i^x gibt, die ab dem Programmschritt x bereits das Analyseergebnis vorwegnehmen.

Begründung: Allgemein gesehen sind Konfigurationen in der in dieser Arbeit verwendeten Sprechweise als der Informationsgehalt bestimmter Speicherbereiche zu sehen. Im Programmfortschritt werden diese Speicherbereiche manipuliert. Aus dem eingangs festgelegten Programmzyklus $\{V_i\}(K_i \rightarrow K_i^n)\{V_i^n\}$ sei K_i^x eine beliebige Konfiguration *i* im Programmschritt x und bildet einen Zustand des SPS Programms ab. Es ist bekannt, dass bezogen auf die Menge verschiedener Eingangszustände einer SPS die Anzahl der korrespondierenden Ausgangszustände deutlich geringer ist. Aus der Definition des Begriffes Klasse folgt, dass die in den Klassen zusammengefassten Ausgangskonfigurationen sich dadurch auszeichnen, dass ihre Ausgangszustände identisch sind. Im Umkehrschluss muss daher auch gelten, dass gleichwertige Konfigurationen existieren, die identische Ausgangzustände erzeugen können. Ab einem bestimmten Programmschritt x ist ein Teilbereich der Ausgangszustände bestimmt, ab dem Programmschritt x+1 werden die restlichen Ausgangszustände bestimmt. Wenn daher ab einem Programmschritt x+1 die in den Konfigurationen für den Programmablauf benötigten Variablen Abschnittsweise übereinstimmen, werden die Beweisschritte übereinstimmende Ergebnisse erzielen. Welche Zustände und Zustandskombinationen davon betroffen sind, ist Ergebnis der Programmanalyse.

Diskussion

Die praktische Bedeutung gleichwertiger Konfigurationen liegt darin, dass mit einer derartigen Vorgehensweise die Anzahl der Arbeitsschritte für die Beweiskette bei der Verifikation stark eingeschränkt werden kann. Ist der Beweis bei der zuerst untersuchten Konfiguration bereits erbracht, bringt der neuerliche Beweis keine neuen Erkenntnisse.

Seien also K_i^x und K_j^x mit $i \neq j$ unterschiedliche Konfigurationen im gleichen Programmschritt des Programmablaufes eines SPS-Programms und wurden die zu verifizierenden Ausgangzustände im finalen Schritt n einer dieser Konfigurationen z.B. der Konfiguration i an der Stelle K_j^n bereits bestimmt und verifiziert, sind Beweisschritte ab der Stelle x+1 für die Kon-

figuration K_j^x nicht mehr notwendig. Mit dieser Methode wird das *state explosion problem* gemildert.

Diese Überlegung zeigt, dass mit Hilfe der Programmanalyse auch Programmabschnitte isoliert werden können. Dadurch wird die Argumentation unterstützt, dass die Verifikation ohne Verlust der Aussagekraft für das Ergebnis über Teilbereiche zulässig ist. Derartige Teilbereiche werden durch die Analyse bestimmt und sind funktionell zusammengehörige Programmteile, also die weiter oben als Teilprogramme identifizierten Programmfunktionen. Darüber hinaus können Unterprogramme aber auch Funktionsblöcke als Teilfunktionen eines SPS-Programms gesehen werden.[47] Wesentliche Merkmale von Unterprogrammen sind, dass in der Regel nur vergleichsweise wenige Parameter mit den übrigen Programmteilen ausgetauscht werden müssen.

[47] Funktionsblöcke sind parametrierbare Bausteine, bei denen als Eingangsgrößen bestimmte Ein- und Ausgänge zugeordnet sind, um z.B. die Ansteuerung einer mehrfach im SPS-Programm benötigten Funktionsgruppe zu realisieren. Dem Funktionsblock werden entsprechende Ein- und Ausgänge zugeordnet. Funktionsblöcke sind als Teilprogramme in sich abgeschlossen und können in der Regel auch einzeln verifiziert werden.

7 Vorbereitende Arbeiten

In diesem Abschnitt wird zunächst untersucht, wie die Zuweisung von Werten z.B. Wahrheitszuständen für die Menge von Ein- und Ausgangsvektoren erfolgen kann. Anders ausgedrückt: es wird untersucht, wie Anfangs- und Endwerte also die Vor- und Nachbedingungen für die Verifikationsschritte bestimmt werden. Intuitiv erscheint dabei die Möglichkeit, diese Bedingungen unmittelbar bei der Konzeption des SPS-Programms direkt aus der Aufgabenstellung abzulesen und sofort festzulegen, als eine vergleichsweise einfache Vorgehensweise.

Um die Komplexität soweit wie möglich in Grenzen zu halten, wird zusätzlich untersucht, wie durch das Aufteilen eines SPS-Programms Programmabschnitte voneinander derart separiert werden können, dass die Dimension von Ein- und Ausgangsvektoren, also die Anzahl der darin vereinigten Elemente, so klein wie möglich gehalten werden kann.

7.1 Programmsegmentierung

Es interessiert zuerst die Beantwortung der Frage, wie zusammengehörige Programmteile, Funktionen oder Funktionsgruppen bestimmt werden können, um diese bei einer Verifikation aus den Gesamtprogramm herauszulösen und isoliert zu betrachten. Im Idealfall kann die Zusammengehörigkeit bereits aus einer Aufgabenstellung hervorgehen. Ein solcher Programmteil „Schere“ wird weiter unten beschrieben und als Beispiel für eine funktionale Einheit einer Maschine herangezogen. Zulässig seien die Abhängigkeit einer solchen Funktion zu anderen Funktionen, die z.B. durch Übergabeparameter wie der Befehl „abschneiden“ und die Meldung „abgeschnitten“ mit weiteren Funktionalitäten einer Maschine zu Funktionsgruppen vereinigt werden.

7.1.1 Ablesen aus der Aufgabenstellung

Ist die Aufgabenstellung ausreichen gut dokumentiert und hat es nicht zu viele Anpassungen oder Veränderungen zu einem ursprünglich geplanten Ablauf gegeben, können zumindest die interessierenden Teile gut voneinander separiert werden. Welche Informationen für die Programmbearbeitung relevant sind, kann mit Hilfe der Rückwärtsanalyse bestimmt werden.

Darüber hinaus wird zusätzlich festgestellt, welche Elemente exklusiv einer derart identifizierten Funktion zur Verfügung stehen und welche auch in weiteren Funktionen oder Programmteilen bearbeitet werden. In der Regel wird eine Liste der Referenzen ausreichend dafür sein.

7.1.2 Nachträgliche Bestimmung

Wenn Aufgabenstellungen nicht exakt bestimmbar sind, etwa bei unzureichenden Programmbeschreibungen oder bei bereits mehrfach überarbeiteten SPS-Programmen, ist die Vorgehensweise aufwendiger. Hier wird eine aus mehreren Arbeitsschritten bestehende Methode vorgeschlagen. In der Folge werden die einzelnen Elemente im Eingangsvektor zur besseren Unterscheidung als Referenzen bezeichnet.

Schritt 1: Rückwärtsanalyse

Alle Aktuatoren oder sonstigen Ausgabeelemente werden mit Hilfe der Rückwärtsanalyse untersucht, um die den Ausgaben der SPS zugeordneten Referenzen in der Konfiguration (am besten in der Eingangskonfiguration bzw. im Eingangsvektor) zuzuordnen. Es entstehen Teilkonfigurationen, deren Referenzen auch von anderen Teilkonfigurationen genutzt werden können.

Schritt 2: Feststellen eines „Überdeckungsgrades“

Rein analytisch wird festgestellt, inwieweit verschiedenen Ausgaben die gleichen Referenzen in der Konfiguration zugeordnet worden sind. Es wird also gefragt, ob verschiedenen Ausgaben der SPS die gleichen Referenzen zugeordnet worden sind. Die Unterscheidung soll ergeben, wie hoch ein „Überdeckungsgrad“ in Bezug auf die Verwendung der genutzten Referenzen ist. Eindeutig ist dabei der Fall, wenn es gar keine Überdeckung gibt.

Schritt 3: Erweiterung des Untersuchungsraumes

Für den Fall einer Überdeckung ab einem bestimmten (parametrierbaren) Schwellwert für die Anzahl der gemeinsam genutzten Referenzen wird durch Hinzunahme von zusätzlichen Referenzen die Basis der Untersuchung für die weitere Überprüfung erweitert. Diese Untersuchung ist abgeschlossen, wenn alle Ausgabeelemente überprüft worden sind.

7.1.3 Anwendung

Zur Erläuterung soll die vorgestellte Schrittfolge beispielhaft auf ein SPS-Programm angewendet werden.

Das weiter unten herangezogene Beispiel beschreibt die Funktion „Schere“. Laut der festgelegten Aufgabenstellung sind dazu drei Ausgänge deklariert. Die Deklaration ist üblicherweise abhängig von der Konzeption, welche Bewegungsabläufe notwendig sind, um die geplante Funktion etwas abzuschneiden durchführbar zu machen. Als minimale Vorgabe, um

die zugehörige Funktion im SPS-Programm zu isolieren, werden die zugeordneten Ausgänge herangezogen, die durch die Verdrahtung der Anlage gegeben ist und damit als bekannt vorausgesetzt werden kann. Typisch für das SPS-Programm könnte sein, dass zusätzlich interne Variable im untersuchten Programmabschnitt benötigt werden. Diese sollen als Ergebnis der Analyse bestimmt werden.

Im Schritt 1 muss zunächst festgestellt werden, welche Elemente aus der Konfiguration zur Bestimmung des Ausgabewerts für den zuerst untersuchten Ausgang benötigt werden (Ausgang 1 in Fig. 36). Daraus ergibt sich ein Set von Referenzen im Eingangsvektor (bzw. der Eingangskonfiguration). Danach wird jeweils der nächste Ausgang auf die gleiche Art untersucht (Ausgang 2 bis Ausgang n in Fig. 36). Im Schritt 2 und 3 wird jeweils zusätzlich festgestellt, ob Referenzen mehrfach genutzt werden, also bestimmend für die Funktionalität der untersuchten Ausgänge sind. Bei mehreren Ausgängen wird verglichen, welche Referenzen mehrfach genutzt sind und welche Referenzen sich unterscheiden. Existieren mehrfach genutzte Referenten und liegt ihre Anzahl unterhalb eines vorgegebenen Schwellwerts für die Mehrfachnutzung, dann werden noch nicht zugeordnete Referenzen, die den weiteren Ausgängen neu zugeordnet worden sind, in das Set der Referenzen mit aufgenommen (z.B. erste Erweiterung in Fig. 36). Dieser Vorgang wird fortgesetzt, wenn in das Set der Referenzen weitere Ausgabereferenzen (z.B. interne Ausgänge, Merker) hinzugenommen worden sind. Abbruchbedingung für den Vorgang ist das Überschreiten des oben erwähnten Schwellwerts.

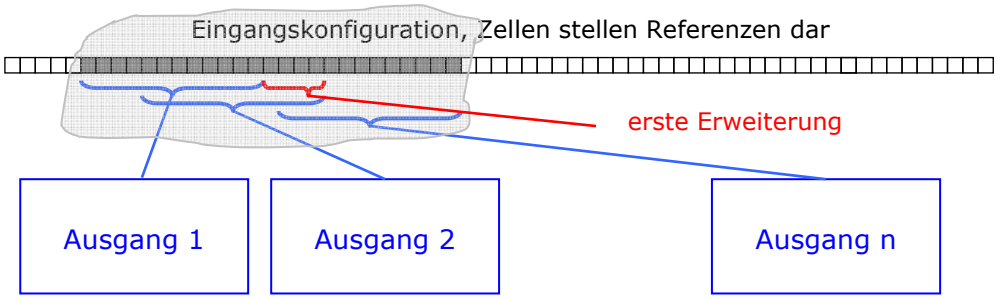


Fig. 36: Modell eines Programmsegments (Funktion)

Schritt 4: Reduktion des Untersuchungsraumes

Wurden alle Ausgaben eines Bereiches bearbeitet, beschreibt die Menge dieser Ausgaben die Mächtigkeit (Dimension) des Ausgabevektors zur untersuchten Funktion. Um weitere Funktionen im SPS-Programm zu identifizieren, können die bereit zugeordneten Ausgaben aus der weiteren Untersuchung ausgeschlossen werden.

Diese gesamte Analyse ist abgeschlossen, wenn allen Ausgabeelemente Funktionen zugeordnet worden sind.

7.2 Methoden zur Bestimmung von Vor- und Nachbedingungen

Die aktuell abgelesenen Eingänge und die zuletzt ausgegebenen Ausgänge bilden zusammengesetzt die Eingangskonfiguration. In der Aufgabenstellung, nach der das Programm einer SPS erstellt worden ist, werden durch die Beschreibung der Abläufe die Reaktionen eines Systems beschrieben. Die Reaktion des durch die SPS gesteuerten Systems auf in der Systembeschreibung enthaltenen Ereignisse bildet den Zusammenhang zwischen einer Anfangssituation und dazu korrespondierenden Endzuständen ab. Klassisch gesehen entsprechen die Anfangssituationen den Vorbedingungen und die zu erzielenden Endzustände die Nachbedingungen für eine Verifikation. In „Schritte zur Verifikation von SPS-Programmen“ [KP12] sind Beispiele dazu enthalten.

Abhängig von der Aufgabenstellung werden unterschiedliche Programmbeschreibungen als Vorgabe für die Programmierung erstellt. Allen ist gemeinsam, dass entweder ein Schaltplan, aus dem die genaue Zuordnung der Ein- und Ausgänge abzulesen ist, oder Belegungslisten, in denen zu den Adressen eine Kurzbeschreibung zur Funktion der betreffenden Anschlüsse an die Hardware beschrieben ist. In der Folge werden zwei Möglichkeiten einer Funktionsbeschreibung diskutiert, die eine in sich abgeschlossene Funktionalität einer Maschine beschreiben, um Ein- bzw. Ausgangsvektoren zu definieren. Anschließend wird festgestellt, wie Ein- und Ausgangsvektoren aus der Aufgabenstellung ablesbar sind.

Beschreibung Variante 1

Gegeben sei die Beschreibung einer Maschinenfunktion. Programmiert werden soll der automatisierte Ablauf für eine Schere, die ein Kabel ablängt. Fig. 37 zeigt die Anordnung zur Verdeutlichung des Prinzips.

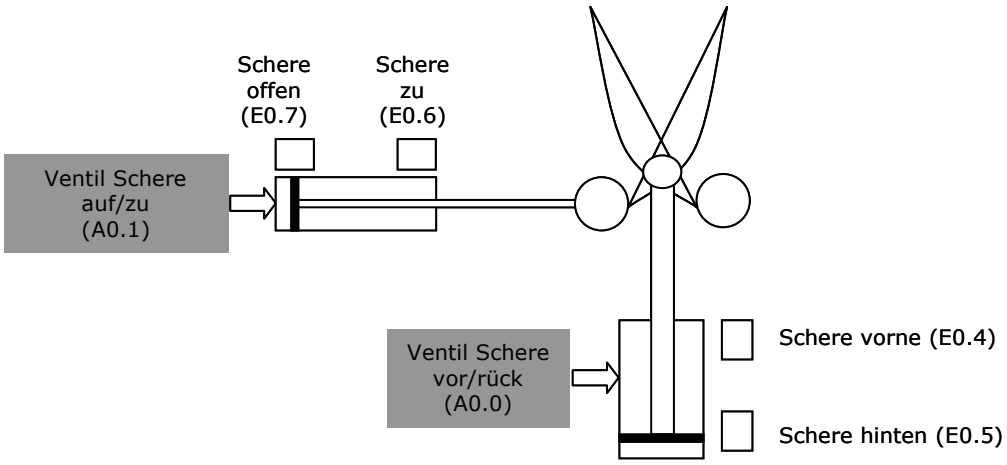


Fig. 37: Prinzip Steuerung der Schere

Zwei Magnetventile (Ausgänge A0.0 und A0.1) steuern jeweils einen Zylinder, die für die Bewegung der Schere bzw. für das Schneiden zuständig sind. Die zugehörigen Stellungen werden mittels Endschaltern überwacht (Eingänge E0.4 bis E0.7). Zwischenstellungen der linearen Bewegungen sind nicht vorgesehen, d.h. die Bewegungen erfolgen jeweils von der einen Endlage zur anderen. Über die Endlagen hinaus ist die Bewegungsmöglichkeit mechanisch begrenzt. Zusätzlich steht eine Auflistung der Signale zur Verfügung (Tab. 3).

Eingang/Ausgang	Bezeichnung	Bemerkung
E0.0	Taster „Abschneiden“	
E0.1	Taster „Zurück“	
E0.2	Schalter „Automatik“	nicht eingeschaltet bedeutet Handbetrieb
E0.3	Taster „Schere“	
E0.4	Endschalter „Schere vorne“	
E0.5	Endschalter „Schere hinten“	(Grundstellung)
E0.6	Endschalter „Schere zu“	
E0.7	Endschalter „Schere offen“	(Grundstellung)
A0.0	Ventil „Schere vor/rück“	nicht eingeschaltet: Hubzylinder ist hinten
A0.1	Ventil „Schere auf/zu“	nicht eingeschaltet: Zylinder ist hinten (Schere ist offen)
A0.2	Meldeleuchte „Anlage fertig“	Wenn der Ablauf gestartet ist, soll die Meldeleuchte blinken

Tab. 3: Signale für das Programm „Schere“

Die Funktionen sind textuell beschrieben.

```
Betriebsart „Handbetrieb“:
Wird der Taster „Abschneiden“ kurz betätigt, fährt die Schere in ihre
vordere Endlage und, wenn diese erreicht ist, wird die Schere geschlossen
und damit das Kabel abgeschnitten.
Wird der Taster „Zurück“ betätigt öffnet zuerst die Schere und fährt danach
in die Ausgangsstellung zurück.

Betriebsart „Automatikbetrieb“:
Mit dem Taster „Schere“ wird jeweils ein (und nur ein) kompletter Ablauf
durchgeführt. Im Programmablauf entspricht der Automatikablauf exakt dem
Handablauf, wie wenn die Taster zuerst „Abschneiden“ und danach „Zurück“
nacheinander betätigt werden.

Um einen weiteren kompletten Ablauf zu starten, muss der Taster „Schere“
neuerlich betätigt werden.

Meldeleuchte:
In der Grundstellung leuchtet die Meldeleuchte. Wenn der Ablauf gestartet
ist, wird die Meldeleuchte ausgeschaltet.
```

Beschreibung Variante 2

Die Maschinenfunktion ist identisch mit jener aus Beschreibung 1. Als Grundlage für die Programmierung wird hier ein Ablaufdiagramm zur Verfügung gestellt (Fig. 38).

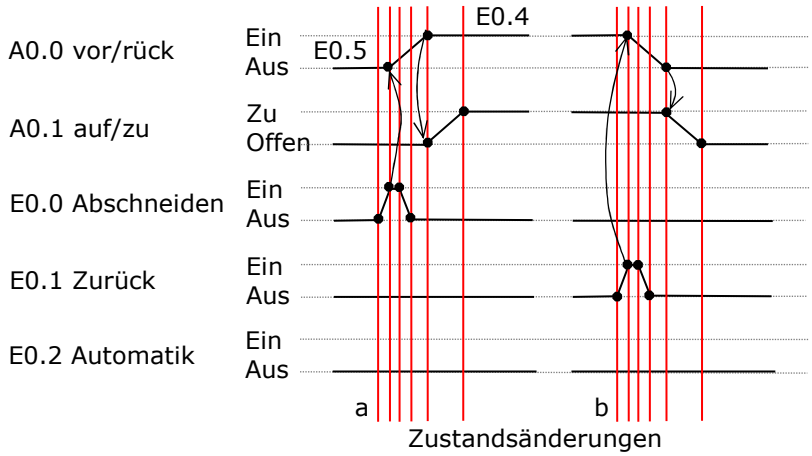


Fig. 38: Steuerung der Schere, Ablaufdiagramm (Ausschnitt)

Die in Fig. 38 dargestellten senkrechten Markierungen stellen unterschiedliche Zustände dar. Wie weiter oben definiert sind z.B. dem Ausgang A0.0 (Bewegung der Schere vor/rück) die Eingänge E0.4 (Schere vorne) und E0.5 (Schere hinten) zugeordnet. Damit lässt sich für den mit „a“ gekennzeichneten Zustand ablesen: A0.0 ist ausgeschaltet, E0.5 ist eingeschaltet, weil die Schere hinten ist, E0.4 ist ausgeschaltet usw. Bei Zustand „b“ wird abgelesen: A0.0 ist eingeschaltet, E0.4 ist eingeschaltet, weil die Schere vorne ist, E0.4 ist ausgeschaltet usw.

Es stellt sich dazu die Frage, wie die Vollständigkeit sichergestellt werden kann, alle möglichen Ein- bzw. Ausgangskonfigurationen zu definieren. Für Eingangskonfigurationen genügt es z.B. mit einer Zählfunktion alle Kombinationen der Wahrheitswerte binärer Eingangswerte darzustellen. Darin sind auch solche Kombinationen enthalten, die z.B. auf eine Fehlbedienung zurückzuführen sind, wenn etwa sich widersprechende Kommandos zeitgleich vorliegen.

Es zeigt sich, dass offenbar das Bestimmen von Ein- und Ausgangsvektoren und damit das Ablesen aus der Aufgabenstellung ausgehend von einem Ablaufdiagramm einfacher sind, als aus der verbalen Beschreibung. Die Zusammenhänge treten klarer hervor, weil mehrere Abhängigkeiten der beteiligten Elemente auf einen Blick erfassbar sind. Der benötigte Aufwand wird daher sehr stark davon abhängen, in welcher Form eine Aufgabenstellung zur Verfügung steht.

Unabhängig davon, in welcher Form eine Aufgabenstellung vorliegt, erscheint es als die günstigste Art Ein- und Ausgangskonfigurationen zu bestimmen, wenn die Bestimmung zusammen mit der Programmierung Schritt für Schritt erfolgt. Problematisch ist dabei jedoch, wenn im Zuge der Inbetriebnahme Veränderungen am Programm notwendig werden. In diesen Fällen müssen parallel dazu auch die betroffenen Bereiche der Konfigurationen überarbeitet werden.

7.3 Vordefinierte Funktionen

Wie bereits weiter oben diskutiert, ist es für die Verifikation günstig, möglichst kleine Programmbereiche in Gruppen zusammengefasst zu betrachten, und derartige Bereiche einzeln zu untersuchen. In diesem Zusammenhang soll als vordefinierte Funktion ein in sich abgeschlossener Programmbereich bezeichnet werden, der eine möglichst geringe Anzahl von Elementen der Ein- und Ausgangsvektoren enthält, die auch anderen Programmbereichen zugeordnet sind. Die dazu benötigten Bereichsgrenzen liefert die Programmanalyse. Elemente, die mehreren Funktionen zuzuordnen sind, müssen jeder Funktion zugeordnet werden. Eingangskonfigurationen sind in der Form des Eingangsvektors Speicherbereichen der SPS Hardware zugeordnet und können in einem Denkmodell als global bezeichnet werden, entsprechend existieren in den oben genannten Funktionen Teilabbilder, die in der Skizze als lokal bezeichnet sind.

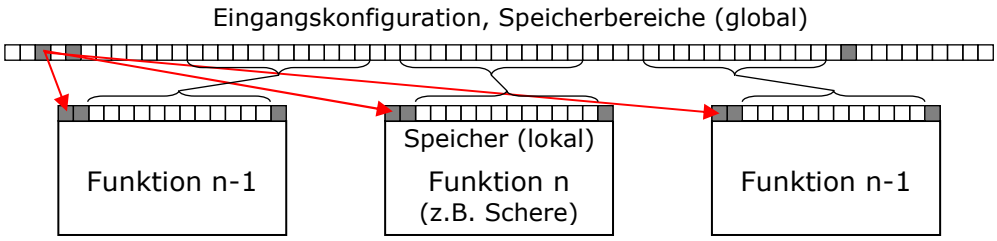


Fig. 39: Modell der Konfiguration mit Bereichszuordnung

Fig. 39 zeigt Bereichszuordnungen als Modell. Gemeinsam genutzte Informationen sind den Funktionen auch mehrfach zugeordnet. Die Mehrfachzuordnung ist bei übergeordneten Funktionalitäten in einem Programm z.B. die Wahl einer Betriebsart (Hand, Automatik etc.), Funktionen für Start und Stopp und weitere.

Bezogen auf die Funktion n (z.B. Schere) lässt sich festlegen, dass alle der Funktion zugeordneten Variablen als eine lokale Teilkonfiguration betrachtet werden kann, die exklusiv genau der betrachteten Funktion zugeordnet ist. Die Eingangskonfiguration wird in virtuelle Teilkonfigurationen geteilt (Fig. 40).

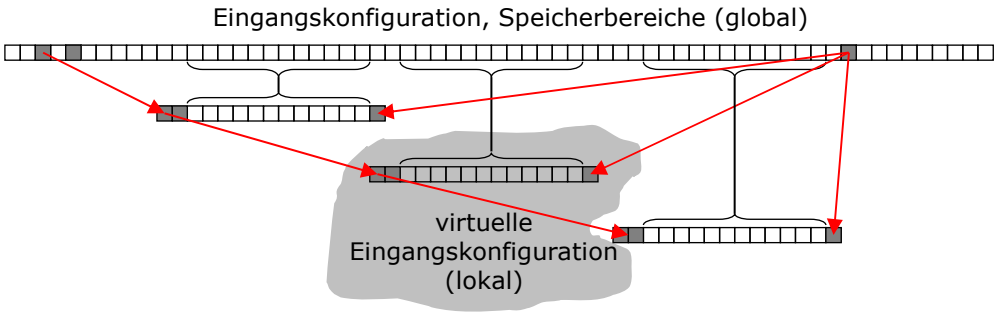


Fig. 40: Lokale Konfigurationen

In Fig. 40 sind jene Elemente der Konfigurationen hervorgehoben, die mehreren virtuellen Konfigurationen zugeordnet sind. Alle Elemente dieser Konfigurationen sind exklusiv einem bestimmten Programmteil zugeordnet. Bei den Elementen kann ein Unterscheidungsmerkmal festgelegt werden: Elemente, die im Zuge der Programmbearbeitung verändert werden (also auch schreibend bearbeitet werden und solche, die nur gelesen werden. Im Beispiel „Schere“ kann z.B. die Meldung „Schere hat abgeschnitten“ ein derartiges Element sein, das für einen Folgeprozess von Bedeutung ist. In diesem Fall existieren Querverbindungen zwischen den Funktionsblöcken, die über den Bereich der lokalen Elemente der Konfigurationen abgebildet werden.

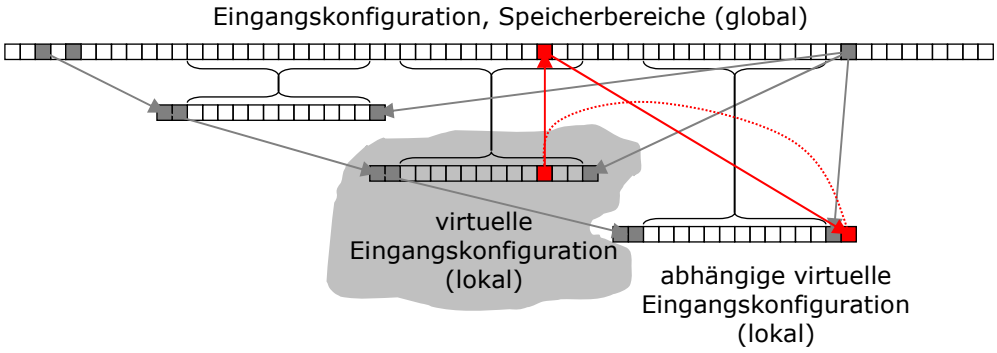


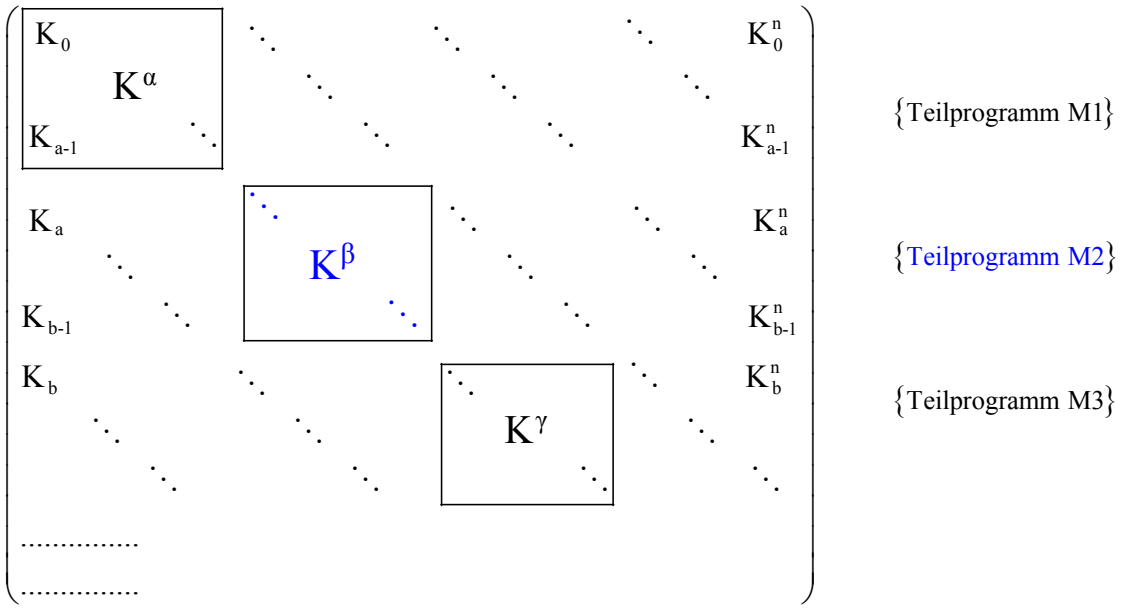
Fig. 41: Verbindung lokaler Konfigurationen

Fig. 41 zeigt die Verbindung von Konfigurationen und deutet damit ihre Abhängigkeit an. Auf Grund der Analyse ist ergänzend festzustellen, ob ein derart genutztes Signal nur genau einmal zugewiesen wird (im Beispiel in der Konfiguration für die Schere). In der Regel wird dies der Fall sein.[48]

[48] Liegt eine Mehrfachzuweisung vor, muss zuerst untersucht werden, ob ein Programmfehler vorliegt. Mehrfachzuweisungen in SPS-Programmen deuten in der Regel auf fehlerhafte Eingaben beim Programmieren hin.

7.4 Einordnung von Funktionen in die Verifikation

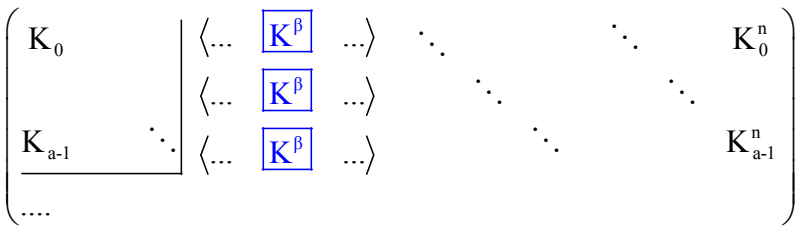
Im Abschnitt 6.3 (Konzept zur Teilverifikation) wurde eine Übersicht der Konfigurationen dargestellt. Darin ist etwa das Teilprogramm **M2** durch eine Matrix von Konfigurationen $\{K^{\beta}\}$ dargestellt. Eine derartige Matrix ist mit den vorher beschriebenen „virtuellen" Konfigurationen gleichzusetzen.



Für die Verifikation bedeutet jede Zeile von $\{K^{\beta}\}$ einen Teilverifikationsschritt des Programmteils **M2.**

Generell muss gelten, dass die Reihenfolge, wie ein SPS-Programm verifiziert werden soll, völlig freigestellt ist. Die wesentliche Bedingung ist die Forderung nach Vollständigkeit, das heißt, dass jedenfalls alle Konfigurationen untersucht werden müssen. Daher ist es zulässig, die Reihenfolge der Arbeitsschritte frei zu wählen.

Seien $\underline{K_0....|}$ eine Konfiguration der Beweiskette **M1**, und $\langle ... \boxed{K^{\beta}} ... \rangle$ alle Konfigurationen der Beweisketten **M2.** Kombiniert man die notwendigen Arbeitsschritte beispielsweise mit den Arbeitsschritten des Teilprogramms **M1**, ergibt sich folgende Matrix:



Diskussion

Die Matrix der Beweisketten $\langle ... \boxed{K^{\beta}} ... \rangle$ beschreibt die Teilbeweise der Beweisketten **M2.** Diese sind für alle Konfigurationen $\underline{K_0 ....}|$ mehrfach gleichartig zu führen. Sind jedoch in vorhergehenden Beweisschritten alle einzelnen Beweise bereits gelungen, resultieren aus den Wiederholungen keine neuen Erkenntnisse.

Weil bei einer Verifikation Vollständigkeit bedingt ist, andererseits bereits Teile, z.B. die Teilverifikation $\langle ... \boxed{K^{\beta}} ... \rangle$ vollständig in vorhergehenden Arbeitsschritte erfolgt ist, können ohne Einschränkung der allgemeinen Gültigkeit solche Ergebnisse für die weitere Beweisführung übernommen werden.

Solange sichergestellt ist, dass Teilverifikationen sich nicht gegenseitig beeinflussen, dass also keine Abhängigkeiten zwischen verschiedenen Programmbereichen eines SPS-Programms existieren, reichen bereits die Teilverifikationsergebnisse aus.

7.5 Zusammenfassung

Die Komplexität zur Bestimmung von Eingangskonfigurationen wird verringert, wenn das zu untersuchende Programm in Teilprogramme gegliedert werden kann. Welche Programmteile strukturell in sich weitgehend geschlossen sind, kann aus der Programmanalyse abgeleitet werden. Werden z.B. Eingangskonfigurationen betrachtet, ergeben sich Zuordnungen von Elementen innerhalb der Konfigurationen, die exklusiv nur in bestimmten Abschnitten eines SPS-Programms verwendet werden.

Im Abschnitt 6.3 „Konzept zur Teilverifikation“ sind die Teilfunktionen einer mittels SPS gesteuerten Maschine als Blöcke von Konfigurationen hervorgehoben dargestellt worden. Elemente der Konfiguration außerhalb dieser Blöcke beeinflussen das Ergebnis der Verifikation nicht und können daher unberücksichtigt bleiben. Als Konsequenz ist mit der Vereinfachung der Verifikation zu rechnen, ohne dass das Ergebnis der Verifikation beeinträchtigt wird. Eine zutreffende Bezeichnung könnte sein: Ein SPS- (Teil-) Programm ist in ein anderes SPS-Programm eingefügt bzw. dort eingekapselt. Herausgelöst nutzt das SPS- (Teil-) Programm eine (Teil-) Konfiguration des zu untersuchenden SPS-Programms. (Teil-) Konfigurationen zusammengesetzt ergeben als Vereinigungsmenge die gesamte Konfiguration.

Prinzipiell können in den Konfigurationen der oben betrachteten Funktionen Elemente enthalten sein, die nur lesend bearbeitet werden. Diese Elemente sind in allen Programmschritten innerhalb eines betrachteten Verifikationsschritts statisch, verändern daher ihren Zustand

nicht. Alle Variablen innerhalb einer Funktion können jedoch verändert werden. Grundsätzlich ist dieser Umstand für die untersuchte Funktion selbst ohne Bedeutung, da ausschließlich das Endergebnis für die weitere Untersuchung relevant ist.

Gesondert untersucht werden muss jedoch, ob Rückmeldungen z.B. andere Funktionen in der Form von im SPS-Programm festgelegten Abhängigkeiten vorliegen. Zur Erläuterung ein einfaches Beispiel: Die Funktion Schere liefert als Ergebnis die Meldung „abgeschnitten“. Eine Folgefunktion kann nach Erreichen dieses Zustands weitere Manipulationen vornehmen. In der Regel wird es nicht eine einzelne Meldung sein, die einen besonderen Zustand der Funktion darstellt, sondern die Verknüpfung mehrerer Zustände, die in Summe weitere Programmteile bzw. Funktionen beeinflussen.

7.6 SPS-Programmbeschreibung für sequentielle Abläufe

In diesem Abschnitt wird der Frage nachgegangen, wie ein SPS-Programm beschrieben werden kann und ob es möglich ist, ohne besondere Zwischenschritte ein lauffähiges SPS-Programm zu gewinnen.

Das vorher herangezogene Beispiel, mit einer Schere ein Kabel abzuschneiden, wird weiter diskutiert. In der Skizze ist eine Anordnung der gerätetechnischen Ausrüstung symbolisch dargestellt (Fig. 46).

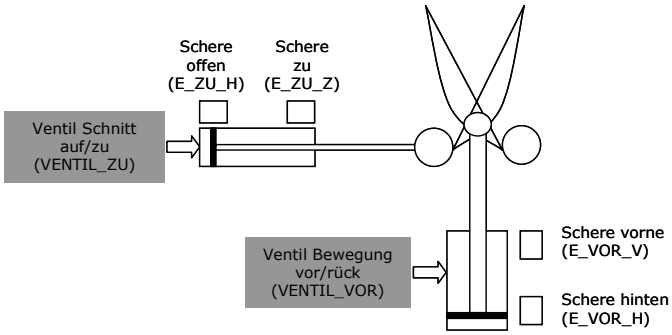


Fig. 42: Skizze „Schere“

Sequenzielle Abläufe sind insbesondere bei Maschinenprogrammen vorherrschend. Unter Berücksichtigung der Eigenheiten von SPS-Programmen kann die Programmkette für die Beispiel Funktion „Abschneiden“ wie folgt in der Form „wenn – dann – sonst“ (if – then – else) verbal beschrieben werden:

Schritt 1:
Wenn die Grundstellung gegeben ist, gilt: VENTIL_VOR ausgeschaltet und VENTIL_ZU ausgeschaltet und die Rückmeldungen E_VOR_H und E_ZU_H sind vorhanden. Wenn in der Betriebsart Automatik der Taster „SCHERE“ betätigt wird oder wenn in der Betriebsart HAND (gleichzusetzen mit nicht im Automatikbetrieb) der Taster „ABSCHNEIDEN“, dann wird das Ventil Schere Bewegung eingeschaltet (VENTIL_VOR) sonst prüfe nächstes Statement

Schritt 2:
Wenn die Schere die vordere Endlage erreicht hat, gilt: VENTIL_VOR eingeschaltet und VENTIL_ZU ausgeschaltet und die Rückmeldungen E_VOR_V und E_ZU_H vorhanden sind, dann wird das Ventil Schere Schneidbewegung eingeschaltet (VENTIL_ZU) sonst prüfe nächstes Statement

Schritt 3:
Wenn die Schnittbewegung die vordere Endlage erreicht hat, gilt: VENTIL_VOR eingeschaltet und VENTIL_ZU eingeschaltet und die Rückmeldungen E_VOR_V und E_ZU_Z sind vorhanden. In der Betriebsart Automatik soll der Ablauf fortgesetzt werden, in der Betriebsart HAND (gleichzusetzen mit nicht im Automatikbetrieb) erst nach Betätigung der Taste „ZURÜCK“. Dann wird das Ventil Schere Schneidbewegung ausgeschaltet (VENTIL_ZU) sonst prüfe nächsten Schritt

Schritt 4:
Wenn die Schere geöffnet ist und die hintere Endlage erreicht hat, gilt: VENTIL_VOR eingeschaltet und VENTIL_ZU ausgeschaltet und die Rückmeldungen E_VOR_V und E_ZU_H vorhanden sind, dann wird das Ventil Schere Bewegung ausgeschaltet (VENTIL_VOR) sonst prüfe nächstes Statement

Schritt 0:
wenn die Schere Bewegung die hintere Endlage erreicht hat, gilt: VENTIL_VOR ausgeschaltet und VENTIL_ZU ausgeschaltet und die Rückmeldungen E_VOR_V und E_ZU_H vorhanden sind, dann ist die Grundstellung wieder erreicht und alle Schritte werden zurückgesetzt, sonst prüfe nächstes Statement

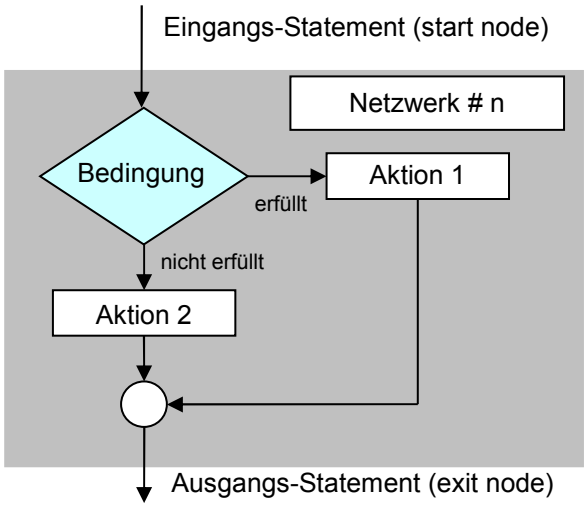
Auswertung Ventil Schere Bewegung:
wenn Schritt 1 aktiv und nicht Schritt 4 aktiv
dann VENTIL_VOR eingeschaltet
sonst VENTIL_VOR ausgeschaltet
prüfe nächstes Statement

Auswertung Ventil Schere Bewegung:
wenn Schritt 2 aktiv und nicht Schritt 3 aktiv
dann VENTIL_ZU eingeschaltet
sonst VENTIL_ZU ausgeschaltet
prüfe nächstes Statement

Auswertung Meldeleuchte:
wenn Schalter „Automatik“ eingeschaltet ist und der Ablauf nicht gestartet worden ist
dann ANLAGE eingeschaltet
sonst ANLAGE ausgeschaltet
prüfe nächstes Statement

Bei der Codierung des Programms stehen dem Programmierer einige Freiheiten zu. In der Folge wird auch wegen der besseren Übersicht eine Schrittkette aufgebaut. Bei der im Beispiel angenommenen Schrittkette wird vorausgesetzt, dass jeder Folgeschritt nur dann ausgeführt werden kann, wenn der logisch vorhergehende Schritt abgeschlossen ist. Entsprechend dem bereits eingangs eingeführten Schema (Fig. 43) erfolgt die Bearbeitung des SPS-Programms.

Fig. 43: Befehlsbearbeitung (vgl. [NF92])



Tab. 4 zeigt das Programm in Tabellenform. Zugeordnet sind den zusammengefassten Bedingungen (if) die zugehörigen Aktionen bei erfüllter Bedingung (then) bzw. bei nicht erfüllter Bedingung (else).

Nr.	if (Bedingungen)	then (dann)	else (sonst)
1	(SCHERE (ein) AUTOMATIK (ein) ODER SCHNEIDEN (ein) /AUTOMATIK (aus)) VENTIL_VOR (aus) VENTIL_ZU (aus) E_VOR_H (hinten) /E_VOR_V (nicht vorne) E_ZU_H (hinten, offen) /E_ZU_Z (nicht geschlossen)	Schritt_1 einschalten (bedeutet VENTIL_VOR einschalten)	next statement
2	Schritt_1 (ein) VENTIL_VOR (ein) VENTIL_ZU (aus) /E_VOR_H (nicht hinten) E_VOR_V (vorne) E_ZU_H (hinten, offen) /E_ZU_Z (nicht geschlossen)	Schritt_2 einschalten (bedeutet VENTIL_ZU einschalten)	next statement
3	(AUTOMATIK (ein) ODER ZURÜCK (ein) /AUTOMATIK (aus)) Schritt_2 (ein) VENTIL_VOR (ein) VENTIL_ZU (ein) /E_VOR_H (nicht hinten) E_VOR_V (vorne) /E_ZU_H (nicht hinten, offen) E_ZU_Z (geschlossen)	Schritt_3 einschalten (bedeutet VENTIL_ZU ausschalten)	next statement
4	Schritt_3 (ein) VENTIL_VOR (ein) VENTIL_ZU (aus) /E_VOR_H (nicht hinten) E_VOR_V (vorne) E_ZU_H (hinten, offen) /E_ZU_Z (nicht geschlossen)	Schritt_4 einschalten (VENTIL_VOR ausschalten)	next statement
5	Schritt_4 (ein) VENTIL_VOR (aus) VENTIL_ZU (aus) E_VOR_H (hinten) /E_VOR_V (nicht vorne) E_ZU_H (hinten, offen) /E_ZU_Z (nicht geschlossen)	Ablauf beendet Schritt_1, Schritt_2, Schritt_3, Schritt_4 ausschalten	next statement
6	Schritt_1 (ein) Schritt_4 (aus)	VENTIL_VOR ein- bzw. ausschalten	ausschalten + next statement
7	Schritt_2 (ein) Schritt_3 (aus)	VENTIL_ZU ein- bzw. ausschalten	ausschalten + next statement
8	AUTOMATIK Schritt_2 (ein)	ANLAGE ein- bzw. ausschalten	ausschalten + next statement

Tab. 4: Programm Schere

Die verwendeten Datentypen sind als BOOL Variable definiert. Weiters wird angenommen, dass die SPS-Adressierung symbolisch mit den oben gewählten Abkürzungen erfolgt. Wird die nach IEC 61131-3 definierte Programmiersprache AWL zu Grunde gelegt, kann nach der in Fig. 43 festgelegten Struktur das SPS-Programm unmittelbar aus der Tab. 4 abgelesen werden.

Nissim Frances nennt in seinem Werk „Program Verification“ [NF92] diese Art der Programmbeschreibung Programmiersprache PLF (programming language as flow diagrams). Das dort vorgestellte Sprachkonzept sieht gerichtete Graphen vor, die in Fig. 43 als Richtungspfeile dargestellt sind. Das Ausgangs-Statement (exit node) am Ende grenzt unmittelbar an den Beginn für die folgende Netzwerkbearbeitung, das folgende Eingangs-Statement (start node, entry node). Im Unterschied zu dem dort beschriebenen Sprachkonzept werden hier die Alternativen nach der Entscheidung als alternative Ergebnisse zusammengeführt und sind für den weiteren Programmablauf zur Verfügung gestellt.

SPS-Programm Schere

Netzwerk 3: Schritt 3		Ablauf starten, Schere in Schnittposition	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	SCHERE	Taster "Schere"	1
U	AUTOMATIK	Schalter "Automatik"	2
LD	SCHNEIDEN	Taster "Abschneiden"	3
UN	AUTOMATIK	Schalter "Automatik" (nicht eingeschaltet, bedeutet Handbetrieb)	4
OLD		Klammerbefehl	5
UN	VENTIL_VOR	Ventil "Schere vor/rück" (nicht eingeschaltet)	6
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	7
UN	E_VOR_V	Endschalter "Schere nicht vorne"	8
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	9
UN	E_ZU_Z	Endschalter "Schere nicht zu"	10
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	11
S	Schritt_1, 1	Schrittmerker 0.1 Ablauf 1. Schritt	12
Netzwerk 2: Schritt 2		Schnittposition erreicht, Kabel abschneiden	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
U	E_VOR_V	Endschalter "Schere vorne"	4
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_2, 1	Schrittmerker 0.2 Ablauf 2. Schritt	8

Netzwerk 3: Schritt 3		Kabel ist abgeschnitten, Schere öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	AUTOMATIK	Schalter "Automatik"	1
LD	ZURÜCK	Taster "Zurück"	2
UN	AUTOMATIK	Schalter "Automatik" (nicht eingeschaltet, bedeutet Handbetrieb)	3
OLD		Klammerbefehl	4
U	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	5
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	6
U	VENTIL_ZU	Ventil "Schere auf/zu" (eingeschaltet)	7
U	E_VOR_V	Endschalter "Schere vorne"	8
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	9
U	E_ZU_Z	Endschalter "Schere nicht zu"	10
UN	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	11
S	Schritt_3, 1	Schrittmerker 0.3 Ablauf 3. Schritt	12
Netzwerk 4: Schritt 4		Schere in die Ausgangsposition zurückziehen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	1
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
U	E_VOR_V	Endschalter "Schere vorne"	4
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_4,1	Schrittmerker 0.4 Ablauf 4. Schritt	8
Netzwerk 5 Reset		Ablauf beendet, Schrittkette zurücksetzen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	1
UN	VENTIL_VOR	Ventil "Schere vor/rück" (nicht eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
UN	E_VOR_V	Endschalter "Schere nicht vorne"	4
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
R	Schritt_1,1	Schrittmerker 0.1 zurücksetzten	8
R	Schritt_2,1	Schrittmerker 0.2 zurücksetzten	9
R	Schritt_3,1	Schrittmerker 0.3 zurücksetzten	10
R	Schritt_4,1	Schrittmerker 0.4 zurücksetzten	11
Netzwerk 6: Ventilansteuerung		Ventil Schere vor/rück	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
UN	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	2
=	VENTIL_VOR	Ventil "Schere vor/rück"	3

Netzwerk 7: Ventilansteuerung		Ventil Schneidblätter schließen/öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	1
UN	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	2
=	VENTIL_ZU	Ventil "Schere auf/zu"	3
Netzwerk 8: Ventilansteuerung		Meldeleuchte "Anlage bereit" ein/ausschalten	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	AUTOMATIK	Schalter "Automatik"	1
UN	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	2
=	ANLAGE	Meldeleuchte "Anlage bereit"	3

Symbol	Adresse	Kommentar
SCHNEIDEN	E0.0	Taster "Zurück"
ZURÜCK	E0.1	Starttaster (Ablauf Start)
AUTOMATIK	E0.2	Schalter "Automatik"
SCHERE	E0.3	Taster "Schere"
E_VOR_H	E0.4	Endlage Schere Bewegung hintere Stellung
E_VOR_V	E0.5	Endlage Schere Bewegung vordere Stellung
E_ZU_H	E0.6	Endlage Schere Schnittbewegung hintere Stellung (offen)
E_ZU_V	E0.7	Endlage Schere Schnittbewegung vordere Stellung (geschlossen)
Schritt_1	M0.1	Ablauf 1. Schritt
Schritt_2	M0.2	Ablauf 2. Schritt
Schritt_3	M0.3	Ablauf 3. Schritt
Schritt_4	M0.4	Ablauf 4. Schritt
VENTIL_VOR	A0.0	Ventil "Schere vor/rück"
VENTIL_ZU	A0.1	Ventil "Schere auf/zu"
ANLAGE	A0.2	Meldeleuchte "Anlage bereit"

Diskussion

Insgesamt ist die Programmbeschreibung vereinfacht, weil Funktionalitäten fehlen. So kann z.B. der beschriebene Ablauf im Fehlerfall nicht angehalten werden. Darüber hinaus gibt es noch mehrere andere Möglichkeiten, die genannte Aufgabenstellung zu lösen. Abgesehen davon, deckt die tabellarische Beschreibung bereits den daraus ableitbaren Programmcode so gut ab, dass die beschriebene Funktion getestet werden kann. Die symbolische Darstellung als Flussdiagramm entspricht der formalen Darstellung des Ablaufes. Die im Flussdiagramm genutzten Gruppen entsprechen im Programm jeweils einer als Netzwerk bezeichneten abgeschlossenen Funktionalität. Weil die Anzahl der Variablen vergleichsweise gering ist, kann die in der Aufgabenstellung beschriebene Funktion einfach simuliert werden.

Das SPS-Programm ist in AWL in STEP7 für eine Kompaktsteuerung codiert. Werden symbolische Adressen benutzt, kann der Ablauf unmittelbar abgelesen werden, ähnlich wie dieser in der tabellarischen Übersicht dargestellt worden ist.

Vereinfachtes SPS-Programm Schere

Netzwerk 1: Schritt 1		Ablauf starten, Schere in Schnittposition	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	START	Taster "Start"	1
UN	VENTIL_VOR	Ventil "Schere vor/rück" (nicht eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
UN	E_VOR_V	Endschalter "Schere nicht vorne"	4
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_1, 1	Schrittmerker 0.1 Ablauf 1. Schritt	8
Netzwerk 2: Schritt 2		Schnittposition erreicht, Kabel abschneiden	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
U	E_VOR_V	Endschalter "Schere vorne"	4
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_2, 1	Schrittmerker 0.2 Ablauf 2. Schritt	8
Netzwerk 3: Schritt 3		Kabel ist abgeschnitten, Schere öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	1
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	2
U	VENTIL_ZU	Ventil "Schere auf/zu" (eingeschaltet)	3
U	E_VOR_V	Endschalter "Schere vorne"	4
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	5
U	E_ZU_Z	Endschalter "Schere zu"	6
UN	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_3, 1	Schrittmerker 0.3 Ablauf 3. Schritt	8
Netzwerk 4: Schritt 4		Schere in die Ausgangsposition zurückziehen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	1
U	VENTIL_VOR	Ventil "Schere vor/rück" (eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
U	E_VOR_V	Endschalter "Schere vorne"	4
UN	E_VOR_H	Endschalter "Schere nicht mehr hinten" (Grundstellung verlassen)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_4,1	Schrittmerker 0.4 Ablauf 4. Schritt	8

Netzwerk 5 Reset		Ablauf beendet, Schrittkette zurücksetzen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	1
UN	VENTIL_VOR	Ventil "Schere vor/rück" (nicht eingeschaltet)	2
UN	VENTIL_ZU	Ventil "Schere auf/zu" (nicht eingeschaltet)	3
UN	E_VOR_V	Endschalter "Schere nicht vorne"	4
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
UN	E_ZU_Z	Endschalter "Schere nicht zu"	6
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
R	Schritt_1,1	Schrittmerker 0.1 zurücksetzten	8
R	Schritt_2,1	Schrittmerker 0.2 zurücksetzten	9
R	Schritt_3,1	Schrittmerker 0.3 zurücksetzten	10
R	Schritt_4,1	Schrittmerker 0.4 zurücksetzten	11
Netzwerk 6: Ventilansteuerung		Ventil Schere vor/rück	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
UN	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	2
=	VENTIL_VOR	Ventil "Schere vor/rück"	3
Netzwerk 7: Ventilansteuerung		Ventil Schneidblätter schließen/öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	1
UN	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	2
=	VENTIL_ZU	Ventil "Schere auf/zu"	3

Darstellung der Zustände

Die Bestimmung der Zustände (der Konfigurationen) ist in Tabellenform dargestellt. In der Folge werden jedoch nur die nach außen wirkenden Signale betrachtet, die als „sichtbare Reaktionen“ der Anlage direkt bei der Funktionsüberprüfung beobachtet werden können. Nach außen wirkende Signale sind z.B. die Rückmeldungen von Positionen der Zylinder oder die Ansteuerungssignale für die Ventile, also die Ein- bzw. Ausgänge der SPS. Programminterne Zustände wie die im Beispiel genutzten Schrittmerker sind in der Regel nur mit Hilfe einer geeigneten Programmentwicklungsumgebung verfolgbar und im rechten Bereich der Tabelle dargestellt. Zur Vereinfachung der Darstellung der möglichen Zustände wird der Vorgang des Abschneidens durch die Funktion „START“ ausgelöst. START bildet dabei die Zusammenfassung der unterschiedlichen Möglichkeiten den Programmablauf in Gang zu setzten.

Konfigurationen

START	E_VOR_H	E_VOR_V	E_ZU_H	E_ZU_Z	VENTIL_VOR	VENTI_ZU		Schritt 1	Schritt 2	Schritt 3	Schritt 4
0	**1**	**0**	**1**	**0**	**0**	**0**	Grundstellung	**0**	**0**	**0**	**0**
1	**1**	**0**	**1**	**0**	**1**	**0**	Start des Ablaufes (Schritt 1)	**1**	**0**	**0**	**0**
_	**0**	**1**	**1**	**0**	**1**	**1**	Schritt 2	**1**	**1**	**0**	**0**
_	**0**	**1**	**0**	**1**	**1**	**0**	Schritt 3	**1**	**1**	**0**	**0**
_	**0**	**1**	**1**	**0**	**0**	**0**	Schritt 4	**1**	**1**	**1**	**1**
0	**1**	**0**	**1**	**0**	**0**	**0**	Ende (Grundstellung)	**0**	**0**	**0**	**0**

Legende: **0** … ausgeschaltet, **1**… eingeschaltet, **_**… nicht relevant, 'DONT CARE'

In der Darstellung sind jene Eingangssignale hervorgehoben, die Auslöser für eine Reaktion der SPS auf Grund der Zustandsänderungen sind. So wird beispielsweise nach dem Betätigen der Funktion „Start" der erste Schritt gesetzt und als Reaktion davon das zugehörige Ventil für die Bewegung der Schere eingeschaltet. Mittels der als Hilfsenergie genutzten Druckluft wird die Bewegung gestartet usw. Während des Ablaufes muss spätestens vor Beendigung der Starttaster losgelassen werden, da sonst der Ablauf neuerlich gestartet würde.

Für die weiteren Überlegungen ist es notwendig, Bewegungsabläufe gedanklich zu analysieren. Die Eigenheit der Hardware SPS ist, dass sie Signalwechsel innerhalb von wenigen Millisekunden registriert und das SPS-Programm entsprechend reagiert. Ein Bewegungsablauf wird jedoch deutlich länger andauern. Ein weiterer Umstand ist, dass die Überwachung einer Endlage nahezu nie sich auf einen bestimmten Punkt beschränken lässt, sondern in der Regel ein Bereich existiert, innerhalb dessen die tatsächlich eingenommene Position liegen wird. Derartige als Schalthysterese bezeichnete Bereiche werden bei der Bewegung passiert und solange der Bereich nicht verlassen worden ist wird in der SPS auch das zugehörige Eingangssignal als „aktiv" erkannt werden. Während der Bewegung kann es also bei sehr kurzen Hub-Bewegungen dazu kommen, dass etwa eine hintere Endlage noch nicht vollständig verlassen worden ist wenn die vordere Endlage bereits „Position erreicht" meldet. Hingegen wird der Regelfall bei längeren Hub-Bewegungen sein, dass während des Ablaufes weder die hintere noch die vordere Endlage anspricht. Exemplarisch sie diese Zwischenzustände (mit und ohne Überschneidung der Zustände) weiter unten dargestellt, wobei die interessierenden Bereiche hervorgehoben sind.

START	E_VOR_H	E_VOR_V	E_ZU_H	E_ZU_Z	VENTIL_VOR	VENTI_ZU		Schritt 1	Schritt 2	Schritt 3	Schritt 4
0	**1**	**0**	**1**	**0**	**0**	**0**	Grundstellung	**0**	**0**	**0**	**0**
1	**1**	**0**	**1**	**0**	**1**	**0**	Start des Ablaufes (Schritt 1)	**1**	**0**	**0**	**0**
_	**1**	**1**	**1**	**0**	**1**	**0**	Überschneidung (Schritt 1)	**1**	**0**	**0**	**0**
_	**0**	**1**	**1**	**0**	**1**	**1**	Schritt 2	**1**	**1**	**0**	**0**
_	**0**	**1**	**1**	**1**	**1**	**1**	Überschneidung (Schritt 2)	**1**	**1**	**0**	**0**
_	**0**	**1**	**0**	**1**	**1**	**0**	Schritt 3	**1**	**1**	**1**	**0**
_	**0**	**1**	**1**	**1**	**1**	**0**	Überschneidung (Schritt 3)	**1**	**1**	**1**	**0**
_	**0**	**1**	**1**	**0**	**0**	**0**	Schritt 4	**1**	**1**	**1**	**1**
_	**1**	**1**	**1**	**0**	**0**	**0**	Überschneidung (Schritt 4)	**1**	**1**	**1**	**1**
0	**1**	**0**	**1**	**0**	**0**	**0**	Ende (Grundstellung)	**0**	**0**	**0**	**0**

Legende: **0** … ausgeschaltet, **1**… eingeschaltet, **_**… nicht relevant, 'DONT CARE'

START	E_VOR_H	E_VOR_V	E_ZU_H	E_ZU_Z	VENTIL_VOR	VENTI_ZU		Schritt 1	Schritt 2	Schritt 3	Schritt 4
0	**1**	**0**	**1**	**0**	**0**	**0**	Grundstellung	**0**	**0**	**0**	**0**
1	**1**	**0**	**1**	**0**	**1**	**0**	Start des Ablaufes (Schritt 1)	**1**	**0**	**0**	**0**
_	**0**	**0**	**1**	**0**	**1**	**0**	ohne Überschneidung (Schritt 1)	**1**	**0**	**0**	**0**
_	**0**	**1**	**1**	**0**	**1**	**1**	Schritt 2	**1**	**1**	**0**	**0**
_	**0**	**1**	**0**	**0**	**1**	**1**	ohne Überschneidung (Schritt 2)	**1**	**1**	**0**	**0**
_	**0**	**1**	**0**	**1**	**1**	**0**	Schritt 3	**1**	**1**	**1**	**0**
_	**0**	**1**	**0**	**0**	**1**	**0**	ohne Überschneidung (Schritt 3)	**1**	**1**	**1**	**0**
_	**0**	**1**	**1**	**0**	**0**	**0**	Schritt 4	**1**	**1**	**1**	**1**
_	**0**	**0**	**1**	**0**	**0**	**0**	ohne Überschneidung (Schritt 4)	**1**	**1**	**1**	**1**
0	**1**	**0**	**1**	**0**	**0**	**0**	Ende (Grundstellung)	**0**	**0**	**0**	**0**

Legende: **0** … ausgeschaltet, **1**… eingeschaltet, **_**… nicht relevant, 'DONT CARE'

Verifikationsschritt

Am Beispiel des Netzwerks 1 wird untersucht, ob dieser Programmteil verifiziert werden kann, ob es also bei Vorliegen bestimmter Eingangsbedingungen im Ergebnis die geforderten Ausgangsbedingungen erzeugt. Definitionsgemäß ist ein Programmschritt verifiziert, wenn nach der Programmbearbeitung unter Zugrundelegung der Vorbedingungen zugesichert werden kann, dass die Nachbedingungen in allen Fällen der Programmbearbeitung erfüllt werden.

vor Aktion 1: Vorbedingung für den Beginn des Ablaufes ist die Grundstellung des Systems. Die Zustände der einzelnen Elemente sind aus der tabellarischen Übersicht ablesbar. Zusätzlich muss der Start aktiv ausgelöst werden.	durch Aktion 1: Nachbedingung ist, dass ein Zustandswechsel für den Schrittmerker (und in weiterer Folge das zugehörige Ventil) ausgelöst werden.
vor Aktion 2: Vorbedingung für den Beginn des Ablaufes ist die Grundstellung des Systems. Der Start wurde jedoch (noch) nicht aktiv ausgelöst	durch Aktion 2: Es erfolgt kein Zustandswechsel

Das Wertepaar [Vorbedingung, Nachbedingung; $\{\mathcal{P}\}$, $\{\mathcal{Q}\}$] beschreibt die Zusicherung über das korrekte Programmverhalten im Sinne der Programmausführung (Verifikation). Ob damit jedoch auch das System richtig reagiert, kann erst die Validation zeigen.

Zum Zustandswechsel stellt sich die Frage, ob programminterne Zustände bei Definition des Zustandsraums zum Teil oder zur Gänze ausgespart werden können. Wird nämlich anstelle der Betrachtung eines einzelnen Netzwerkes das gesamte Programm als funktionale Gruppe betrachtet, sind die nach außen wirkenden Systemreaktionen entscheiden, denn genau diese sollen verifiziert und beim Systemtest auch validiert werden.

Für die Bestimmung der Wertepaare [$\{\mathcal{P}\}$, $\{\mathcal{Q}\}$] stellt sich die Frage, ob die nur intern berechneten Zustände, das sind die im Beispiel SCHERE definierten Schrittmerker, in die Verifikationsbedingungen aufgenommen werden müssen oder nicht. Folgende Konfigurationen der Zustände ohne Berücksichtigung der Schrittmerker sind aus der Aufstellung ablesbar (Tab. 5),

Im Schritt 1a (Tab. 5) ist zum Zeitpunkt der Auslösung das System SCHERE noch in der Grundstellung. Ausgelöst durch den START Befehl erfolgt der Übergang in den Zwischenschritt 1x-a. Durch das Ventil angetrieben setzt sich der Kolben in Bewegung und die Abfragen der zugehörigen Endlagen nehmen Zwischenzustände ein (Zwischenschritt 1x). Der dem Schritt zugeordnete Endzustand Schritt 1x-e bzw. Schritt 1e wird mit dem Erreichen der Endposition des Kolbens erreicht und der zugehörige Endschalter zeigt die Position an.

Stellung	Eingangskonfig.(E/A)	Ergebnis	Anmerkung	
Grundstellung	[0, 1, 0, 1, 0, 0, 0]	→ [0, 0]		✓
Schritt 1a	[1, 1, 0, 1, 0, 0, 0]	→ [1, 0]	Schritt ausgelöst	✓
Schritt 1x-a	[1, 1, 0, 1, 0, 1, 0]	→ [1, 0]	Übergang	✓
Schritt 1x	[_, _, _, 1, 0, 1, 0]	→ [1, 0]	Übergang	✓
Schritt 1x-e	[_, 0, 1, 1, 0, 1, 0]	→ [1, 0]	Übergang	
Schritt 1e	[_, 0, 1, 1, 0, 1, 0]	→ [1, 0]	durchgeführt	✓
Schritt 2a	[_, 0, 1, 1, 0, 1, 0]	→ [1, 1]	Schritt ausgelöst	✓
Schritt 2x-a	[_, 0, 1, 1, 0, 1, 1]	→ [1, 1]	Übergang	✓
Schritt 2x	[_, 0, 1, _, _, 1, 1]	→ [1, 1]	Übergang	✓
Schritt 2x-e	[_, 0, 1, 0, 1, 1, 1]	→ [1, 1]	Übergang	
Schritt 2e	[_, 0, 1, 0, 1, 1, 1]	→ [1, 1]	durchgeführt	✓
Schritt 3a	[_, 0, 1, 0, 1, 1, 1]	→ [1, 0]	Schritt ausgelöst	✓
Schritt 3x-a	[_, 0, 1, 0, 1, 1, 0]	→ [1, 0]	Übergang	✓
Schritt 3x	[_, 0, 1, _, _, 1, 0]	→ [1, 0]	Übergang	✓
Schritt 3x-e	[_, 0, 1, 1, 0, 1, 0]	→ [1, 0]	Übergang	
Schritt 3e	[_, 0, 1, 1, 0, 1, 0]	→ [1, 0]	durchgeführt	✓
Schritt 4a	[_, 0, 1, 1, 0, 1, 0]	→ [0, 0]	Schritt ausgelöst	✓
Schritt 4x-a	[_, 0, 1, 1, 0, 0, 0]	→ [0, 0]	Übergang	✓
Schritt 4x	[_, _, _, 1, 0, 0, 0]	→ [0, 0]	Übergang	✓
Schritt 4x-e	[0, 1, 0, 1, 0, 0, 0]	→ [0, 0]	Übergang	
Schritt 4e	[0, 1, 0, 1, 0, 0, 0]	→ [0, 0]	durchgeführt	✓
Grundstellung	[0, 1, 0, 1, 0, 0, 0]	→ [0, 0]		✓

Tab. 5: Zustandsübergänge Schere

Im realen Programmablauf verharrt das System in der Grundstellung also eine unbestimmte Anzahl von SPS-Programmzyklen, solange, bis der Start ausgelöst wird. Der Übergang von Schritt 1a zum Schritt 1x-a benötigt genau einen Programmzyklus, danach erfolgen wieder mehrere Programmzyklen, die im Schritt 1x symbolisiert sind. Der erreichte Endzustand ist als Schritt 1x-e bzw. als Schritt 1e bezeichnet. Die Folgeschritte erfolgen analog.

7.7 Zusammenfassung

Ausgangspunkt für ein SPS-Programm ist eine Aufgabenstellung. Die genannten Steuerungsaufgaben sind mit geringfügigen Anpassungen aus einer realen Anwendung beispielhaft herangezogen worden. Dazu wurde z.B. der Ablauf „Schere“ als Teil eines Programmbausteins betrachtet.

Es ist davon auszugehen, dass in Fertigungsautomaten nur wenige Funktionen existieren, die in ihrer Größe so einfach überblickbar und behandelbar sind. Im Anhang sind Programmbeschreibungen einer umfangreicheren Fertigungseinrichtung vorgestellt. Im Wesentlichen sind

auch diese Beschreibungen als informell zu bezeichnen, die erst auf Gund zusätzlicher Informationen Basis für ein entsprechendes SPS-Programm bieten (Anhang B).

Darüber hinaus müssen diese Programme erst in das vom Verifikator genutzte „Sprachkonzept“ übertragen werden (die bereits erwähnten unterschliedlichen „Dialekte“ der SPS-Programmiersprache AWL). Die Umwandlung für AWL von Step 7 Steuerungsfamilie S7-300 ist im Anhang C vorgenommen worden. Das dort genutzte Hilfsprogramm erzeugt dazu eine als „cross reference“ bezeichnete Aufstellung, in der die verwendeten Elemente der betreffenden Programmteile zusammengestellt sind. Diese bilden die Basis, aus denen die zugehörigen Verifikationsbedingungen definiert werden.

8 Definition von Verifikationsbedingungen

Neben der Formalisierung von SPS-Programmen ist das Bestimmen der Ein- und Ausgangskonfigurationen der Zustände eine Voraussetzung. Gefragt wird, inwieweit direkt aus der Definition einer Aufgabenstellung Verifikationsbedingungen gewonnen werden können.

8.1 Beispiel: Sequentieller Ablauf „Schere“

Zur Planung eines Testlaufs mit dem Verifikator wird das etwas vereinfachte SPS-Programm herangezogen, Neben der Funktionstaste „Start“ werden die Endlagen der beiden Pneumatik Zylinder (4 Stk. Eingänge, E) und die beiden Ausgänge (A) für die Ventile genutzt. Für die Schrittmerker (M) sind Platzhalter vorgesehen. In Tab. 6 sind die Konfigurationen dargestellt.

Eingangskonfiguration (E/M/A)	Ergebnis (M/A)	Anmerkung	K
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [0, 0, 0, 0, 0, 0]	Grundstellung	1
[1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [1, 0, 0, 0, 1, 0]	Schritt ausgelöst	2
[1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	Übergang	3
[_, _, _, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	Übergang	4
[_, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	durchgeführt	5
[_, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 1, 0, 0, 1, 1]	Schritt ausgelöst	6
[_, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	Übergang	7
[_, 0, 1, _, _, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	Übergang	8
[_, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	durchgeführt	9
[_, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1]	→ [1, 1, 1, 0, 1, 0]	Schritt ausgelöst	10
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 0, 1, 0]	Übergang	11
[_, 0, 1, _, _, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 0, 1, 0]	Übergang	12
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 1, 1, 0]	durchgeführt	13
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 1, 0, 0]	Schritt ausgelöst	14
[_, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	Übergang	15
[_, _, _, 0, 1, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	Übergang	16
[0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	durchgeführt	17
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [0, 0, 0, 0, 0, 0]	Grundstellung	1

Tab. 6: Zustandsübergänge Schere für Verifikation

Die Eingangskonfiguration (Tab. 6) ist gegliedert in fünf Zustände für die Eingänge (E) als Teilbereich der Eingangsinformationen, weitere vier Zustände für interne Informationen hier die Schrittmerker (M) und zusätzlich zwei Zustände für die Ausgänge (A). Die Ausgänge und die Merker in den Eingangsinformationen bilden die Historie des Programmfortschritts ab,

d.h. sie verändern ihren Informationsgehalt mit dem Programmfortschritt. Schrittmerker (M) und Ausgänge (A) werden im Ergebnis abgebildet. Welche Elemente dem untersuchten Programmteil zugeordnet werden, ist durch die Programmanalyse zu ermitteln. Entscheidend ist, dass speziell die Ausgänge exklusiv einem Teilprogramm zugeordnet werden können, während Eingänge und Merker auch in weiteren Teilprogrammen auftreten können. Der Übergang von der Auslösung eines Schrittes in den Folgezustand ($2 \rightarrow 3$, $6 \rightarrow 7$, $10 \rightarrow 11$, $14 \rightarrow 15$) findet immer in einer unmittelbaren Abfolge statt und ist damit von besonderem Interesse.

Rein rechnerisch ergibt sich, dass es bei 11 Variablen 2^{11} unterschiedliche Eingangskonfigurationen geben kann. Untersucht müssen jedoch nur 17 verschiedene Konfigurationen werden (vgl. Tab. 6). Weil jedoch bei dem als Übergang (von einem Zustand in den unmittelbaren Folgezustand) z.B. der Zustand 3 als Variante des Zustands 4 darstellbar ist, können die mit 4, 8, 12 und 16 bezeichneten Zustände sogar aus den Verifikationsbedingungen ohne Beeinträchtigung des Gesamtergebnisses weggelassen werden.

Damit sind Vor- und Nachbedingungen festgelegt. Die Verifikation selbst erfolgt durch die in „Schritte zur Verifikation von SPS-Programmen“ [KP12] beschriebene Methodik.

Diskussion (1)

Zur Diskussion steht zunächst die Frage nach der Deutung der im Beispiel angesprochenen Vorgehensweise. Vordergründig ist sofort klar, dass eine Reduktion von Überprüfungsschritten, die ohne Beeinträchtigung der Gültigkeit der Verifikation stattfinden können, Zeit, Komplexität und Ressourcen einspart. Ein weiterer Punkt ist, dass die beteiligten Teilbereiche der Ausgangsvektoren bereits vollständig verifiziert worden sind, solange alle Einflussgrößen in der Form der Eingangskonfigurationen berücksichtigt worden sind, was definitionsgemäß stattgefunden hat.

Damit wurde das Beispielprogramm SCHERE ähnlich dem Teilprogramm **M2** durch eine Matrix von „virtuellen“ Konfigurationen $\left\{K^{\beta}\right\}$, wie diese im 7.4 Abschnitt beschrieben worden sind, bereits vollständig dargestellt. Damit ist aber auch der zugehörige SPS-Programmabschnitt mit Hilfe des Verifikators verifizierbar.

Als weitere Frage ist zu untersuchen, welche Querbeeinflussung von anderen Programmteilen des untersuchten SPS-Programms gegeben sein könnte. Offenbar muss das Kabel, dass durch die Ansteuerung der SCHERE abgeschnitten worden ist, einerseits entnommen werden und andererseits durch ein neues Kabel ersetzt werden, um etwa den Vorgang des Abschneidens zu wiederholen. Es ist leicht einzusehen, dass das Hantieren mit dem Kabel nur stattfinden

kann, wenn die SCHERE solange in ihrer Grundstellung verharrt. Daraus folgt, dass bestimmte Rückmeldungen von Eingängen (z.B. Scherenzylinder befinden sich in der Grundstellung) in einem anderen Programmteil des SPS-Programms ebenfalls berücksichtigt sein müssen, also bestimmte Informationen aus den Konfigurationen auch für andere Programmteile relevant sind.

Als Lösung bietet sich an, solche Informationen in die zugehörige Eingangskonfiguration mit einzubinden, also bestimmte Elemente wie z.B. die zugehörigen Eingänge in mehrere Teilkonfigurationen mit aufzunehmen, die dann für sich ein weiteres vollständig beschreibbares Teilprogramm der SPS abbilden.

Diskussion (2)

Ein weiterer Diskussionspunkt ist, ob nur intern genutzte Werte von Variablen wie z.B. Merker in den Konfigurationen berücksichtigt werden müssen oder nicht. Für das hier diskutierte Beispielprogramm Schere wird gefragt, ob die Schrittmerker für die Fallunterscheidung der Zustände erforderlich sind oder nicht.

Eingangskonfiguration (E/M/A)	Ergebnis (M/A)	Anmerkung	K
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [0, 0, 0, 0, 0, 0]	Grundstellung	1
[1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [1, 0, 0, 0, 1, 0]	Schritt ausgelöst	2
[1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	Übergang	3
[_, _, _, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	Übergang	4
[_, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 0, 0, 0, 1, 0]	durchgeführt	5
[_, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0]	→ [1, 1, 0, 0, 1, 1]	Schritt ausgelöst	6
[_, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	Übergang	7
[_, 0, 1, _, _, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	Übergang	8
[_, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1]	→ [1, 1, 0, 0, 1, 1]	durchgeführt	9
[_, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1]	→ [1, 1, 1, 0, 1, 0]	Schritt ausgelöst	10
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 0, 1, 0]	Übergang	11
[_, 0, 1, _, _, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 0, 1, 0]	Übergang	12
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 1, 1, 0]	durchgeführt	13
[_, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0]	→ [1, 1, 1, 1, 0, 0]	Schritt ausgelöst	14
[_, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	Übergang	15
[_, _, _, 0, 1, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	Übergang	16
[0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]	→ [1, 1, 1, 1, 0, 0]	durchgeführt	17
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]	→ [0, 0, 0, 0, 0, 0]	Grundstellung	1

Tab. 7: Zustandsübergänge Schere ohne Schrittmerker

In Tab. 7 sind die Schrittmerker hervorgehoben. Untersucht werden die möglichen Zustände der Eingänge (expandiert durch Elimination der 'Dont Care') ohne Berücksichtigung das den Programmablauf auslösenden Start und die Zustände der Ausgänge, die zur einfacheren Unterscheidbarkeit Hexadezimal codiert worden sind.[49]

START	E_VOR_H	E_VOR_V	E_ZU_H	E_ZU_Z	VENTIL_VOR	VENTI_ZU		Code E	Code A
0	**1**	**0**	**1**	**0**	**0**	**0**	Grundstellung	**A**	**0**
1	**1**	**0**	**1**	**0**	**1**	**0**	Start des Ablaufes (Schritt 1)	**A**	**2**
_	**0**	**0**	**1**	**0**	**1**	**0**	Überschneidung (Schritt 1)	**2**	**2**
_	**1**	**1**	**1**	**0**	**1**	**0**	Überschneidung (Schritt 1)	**1**	**2**
_	**0**	**1**	**1**	**0**	**1**	**1**	Schritt 2	**E**	**3**
_	**0**	**1**	**0**	**0**	**1**	**1**	Überschneidung (Schritt 2)	**4**	**3**
_	**0**	**1**	**1**	**1**	**1**	**1**	Überschneidung (Schritt 2)	**7**	**3**
_	**0**	**1**	**0**	**1**	**1**	**0**	Schritt 3	**5**	**2**
_	**0**	**1**	**0**	**0**	**1**	**0**	Überschneidung (Schritt 3)	**4**	**2**
_	**0**	**1**	**1**	**1**	**1**	**0**	Überschneidung (Schritt 3)	**7**	**2**
_	**0**	**1**	**1**	**0**	**0**	**0**	Schritt 4	**6**	**0**
_	**0**	**0**	**1**	**0**	**0**	**0**	Überschneidung (Schritt 4)	**2**	**0**
_	**1**	**1**	**1**	**0**	**0**	**0**	Überschneidung (Schritt 4)	**7**	**0**
0	**1**	**0**	**1**	**0**	**0**	**0**	Ende (Grundstellung)	**A**	**0**

Legende: **0** … ausgeschaltet, **1**… eingeschaltet

Codierung (aufsteigend geordnet): 12H, 20 H, 22 H, 42 H, 43 H, 52 H, 70 H, 72 H, 73 H, A0 H, A2 H, E3 H

Der Code stellt in hexadezimaler Darstellung die Zustände der Ein- bzw. Ausgänge dar. Aus der Betrachtung der Codetabelle ist direkt ersichtlich, das im gegebenen Fall alle codierten Werte verschieden sind. Das bedeutet, dass auf die Berücksichtigung der Zustände von Schrittmerkern verzichtet werden kann, ohne das Ergebnis der Verifikation zu beeinflussen. Auf diese Art kann auch der zu betrachtende Zustandsraum signifikant verkleinert werden, wodurch das state explosion Problem weiter gemildert wird.

8.2 Programmiervarianten im Ablauf „Schere“

Die Art und Weise, wie ein Programmierer bei der Programmerstellung vorgeht, ist individuell verschieden. Daher wird auch die gleiche Aufgabenstellung in unterschiedlichen Varianten gelöst. Formal gesehen sind solche Lösungen unvollständig, weil zwar den Vorgaben entspre-

[49] Bei der Expansion der Zustände im Bereich der Überschneidungen muss nur zwischen mit und ohne Überschneidung unterschieden werden, wie Beginn und Ende der Zylinderbewegung als Hauptbedingungen bereits in der Werteaufstellung berücksichtigt sind.

chend ein SPS-Programm erstellt worden ist aber bestimmte Zustände unberücksichtigt geblieben sind.

Wird beispielsweise im Programm SCHERE die wenn – dann Beziehung „wenn die Endlage erreicht ist, dann erfolgt der nächste Programmschritt“ wortwörtlich umgesetzt, könnte im Fehlerfall z.B. bei einer defekten Endlageüberwachung ein Folgeschritt bei einer als problematisch zu wertenden Kombination von Zuständen eingeleitet werden.

In der in Folge betrachteten Programmvariante der Funktion SCHERE sind eine Reihe von Programmzeilen entfernt worden. Es entsteht dadurch folgender Programmcode:

Netzwerk 1: Schritt 1		Ablauf starten, Schere in Schnittposition	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	START	Taster "Start"	1
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_1, 1	Schrittmerker 0.1 Ablauf 1. Schritt	8
Netzwerk 2: Schritt 2		Schnittposition erreicht, Kabel abschneiden	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
U	E_VOR_V	Endschalter "Schere vorne"	4
S	Schritt_2, 1	Schrittmerker 0.2 Ablauf 2. Schritt	8
Netzwerk 3: Schritt 3		Kabel ist abgeschnitten, Schere öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	1
U	E_ZU_Z	Endschalter "Schere zu"	6
S	Schritt_3, 1	Schrittmerker 0.3 Ablauf 3. Schritt	8
Netzwerk 4: Schritt 4		Schere in die Ausgangsposition zurückziehen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	1
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	Schritt_4,1	Schrittmerker 0.4 Ablauf 4. Schritt	8
Netzwerk 5 Reset		Ablauf beendet, Schrittkette zurücksetzen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	1
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
R	Schritt_1,1	Schrittmerker 0.1 zurücksetzten	8
R	Schritt_2,1	Schrittmerker 0.2 zurücksetzten	9
R	Schritt_3,1	Schrittmerker 0.3 zurücksetzten	10
R	Schritt_4,1	Schrittmerker 0.4 zurücksetzten	11

Netzwerk 6: Ventilansteuerung		Ventil Schere vor/rück	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_1	Schrittmerker 0.1 Ablauf 1. Schritt	1
UN	Schritt_4	Schrittmerker 0.4 Ablauf 4. Schritt	2
=	VENTIL_VOR	Ventil "Schere vor/rück"	3
Netzwerk 7: Ventilansteuerung		Ventil Schneidblätter schließen/öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	Schritt_2	Schrittmerker 0.2 Ablauf 2. Schritt	1
UN	Schritt_3	Schrittmerker 0.3 Ablauf 3. Schritt	2
=	VENTIL_ZU	Ventil "Schere auf/zu"	3

In einer weiteren Vereinfachung des Programms werden die Schrittmerker entfernt und die Ventile direkt angesprochen. Dann ergibt sich folgende Variante:

Netzwerk 1: Schritt 1		Ablauf starten, Schere in Schnittposition	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	START	Taster "Start"	1
U	E_VOR_H	Endschalter "Schere hinten" (Grundstellung)	5
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
S	VENTIL_VOR	Ventil "Schere vor/rück"	x
Netzwerk 2: Schritt 2		Schnittposition erreicht, Kabel abschneiden	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	VENTIL_VOR	Ventil "Schere vor/rück"	1
U	E_VOR_V	Endschalter "Schere vorne"	4
S	VENTIL_ZU	Ventil "Schere auf/zu"	x
Netzwerk 3: Schritt 3		Kabel ist abgeschnitten, Schere öffnen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LD	VENTIL_ZU	Ventil "Schere auf/zu"	1
U	E_ZU_Z	Endschalter "Schere zu"	6
R	VENTIL_ZU	Ventil "Schere auf/zu"	x
Netzwerk 4: Schritt 4		Schere in die Ausgangsposition zurückziehen	
Bef.	Operand (Symb.)	Kommentar (Funktion)	Nr.
LDN	VENTIL_ZU	Ventil "Schere auf/zu"	1
U	E_ZU_H	Endschalter "Schere offen" (Grundstellung)	7
R	VENTIL_VOR	Ventil "Schere vor/rück"	x

Diskussion

Beide oben vorgestellten Programmvarianten erfüllen die von der Schere erwartete Funktionalität. Alle definierten Ein- bzw. Ausgangssignale werden im Programmcode verknüpft.

Vorausgesetzt muss jedoch dabei werden, dass seitens der Eingangs- bzw. Ausgangssignale die fehlerfreie Funktion sichergestellt ist. Unterstellt wird, dass z.B. der Endschalter **E_VOR_H** dauernd den Zustand vorne meldet. Gefragt ist, wie der Ablauf in einem solchen Fall stattfinden würde:

- Vor dem Startsignal: es ist keine Fehlfunktion feststellbar
- Startsignal: der erste Schritt, Bewegung der geöffneten Schneidblätter in die Schnittposition erfolgt gemäß der Aufgabenstellung. Auch die übrigen Schritte verlaufen wie geplant.
- In der Programmvariante mit Verwendung von Schrittmerkern werden die Schrittmerker nicht zurückgesetzt, das heißt, dass der Ablauf nur ein Mal stattfindet. Wurde auf die Verwendung von Schrittmerkern verzichtet, ist zunächst kein Unterschied zum Normalablauf festzustellen. Allerdings könnte ein Startsignal den Ablauf auch aus jeder beliebigen Stellung des Zylinders für die Scherenbewegung auslösen.
- Weil nur jene Rückmeldungen des Systems die einen Schrittwechsel auslösen berücksichtigt worden sind, könnte der Programmablauf bei defekten Signalgebern (hier die Endlagenüberwachungen) fehlerhaft ablaufen.

8.3 Gegenbedingungen

Die ursprünliche Ausgangsüberlegung zu diesem Kapitel war das Bestimmen der Ein- und Ausgangskonfigurationen der Zustände als Verifikationsbedingungen. Gegenbedingungen sind solche Ein- und Ausgangskonfigurationen von Zuständen, die das durch die SPS gesteuerte System in einen programmtechnisch nicht gewollten und damit unzulässigen Zustand überführen.

In Abschnitt 8.1 sind als Konfigurationen Zustände codiert wie folgt dargestellt worden: 12_H, 20_H, 22_H, 42_H, 43_H, 52_H, 70_H, 72_H, 73_H, $A0_H$, $A2_H$ und $E3_H$. Nicht in dieser Auflistung genannte Konfigurationen dürften theoretisch nicht auftreten bzw. können bei funktionell einwandfrei aufgebauten Systemen nicht existieren und deuten damit auf fehlerhafte Systeme hin. Als Beispiele seien die Konfigurationen 00_H und $F3_H$ herangezogen: laut Funktion der Schere müssen einerseits wenigstens bestimmte Endlagenschalter bei ausgeschalteten Ventilen den Zustand „geschaltet“ rückmelden (außer es wurde die Hilfsenergie Druckluftversorgung für die Zylinderantriebe nicht angeschlossen). Andererseits dürfen keinesfalls alle Endlagenschalter „geschaltet“ melden, wenn beide Ventile eingeschaltet sind.

Die Definition von Gegenbedingungen ist daher geeignet, insbesondere Reaktionen der SPS für sicherheitskritische Zustände mit Hilfe des Verifikators zu untersuchen.

Als „neutrale“ Bedingungen können solche Konfigurationen festgelegt werden, die im Zuge des Programmablaufs durchaus vorkommen können, die jedoch keinen Einfluss auf die geplante Funktion der Aufgabenstellung haben. Im Beispielprogramm „Schere“ wäre dies zum Beispiel das zeitgleiche Schalten der Endlagen „vorne“ und „hinten“ während des Bewegungsablaufes eines Zylinders.

8.4 Überlegungen zur Vollständigkeit des Untersuchungsraums

Weitere Überlegungen betreffen die Frage, in wie weit der Untersuchungsraum der Ein- und Ausgangskonfigurationen der Zustände als Verifikationsbedingungen vollständig abgebildet wird oder nicht. Das vorher herangezogene Beispielprogramm „Schere“ konnte in der vereinfachten Form vollständig verifiziert werden. In realen Anwendungen werden typisch zusätzliche Ein- bzw. Ausgänge benötigt, etwa um einzelne Zustandsmeldungen für die Bedienerführung bereitzustellen. Dazu sind in der Regel eigene Funktionalitäten im Programm vorgesehen, die separat wie weitere SPS-Teilprogramme behandelt werden können. Beispiele dafür sind die Meldungen zu Betriebszuständen, die das gesteuerte System gerade eingenommen hat.

Im Gegensatz zu den Zustandsinformationen in den Eingangskonfigurationen werden jene der Ausgangskonfigurationen genau einmal mit dem Ende des SPS-Programmdurchlaufes final zugewiesen. Das bedeutet für die Programmfunktion „Schere“, dass zu diesem Zeitpunkt ein Zustandsübergang in Abhängigkeit der Eingangszustände stattfindet, unabhängig davon, in welchem Abschnitt des SPS-Programms der zugehörige Programmcode situiert gewesen ist. Daraus folgt, dass die Bearbeitung des übrigen Programmcodes keinen Einfluss auf die Ergebnisse der Berechnungen des Programmabschnitts „Schere“ nimmt.

Analog gilt diese Aussage für alle Teilfunktionen eines SPS-Programms. Der Untersuchungsraum ist dann und nur dann vollständig, wenn alle Ausgangszustände erfasst sind.

Das Erfassen aller Ausgangzustände und die Zuordnung in funktionale Gruppen ist die Aufgabe der Programmanalyse. Die Frage zur Vollständigkeit wird dadurch überprüft, dass alle im Programm vorhandenen Ausgangsreferenzen mit der Liste der Ausgänge laut einer gegebenen Aufgabenstellung verglichen werden.

8.5 Arbeitsschritte

Für die Verifikation von Programmabschnitten werden die zugehörigen Ein- und Ausgangskonfigurationen der Zustände definiert. Dieser Arbeitsschritt sollte parallel zur Programmerstellung erfolgen. Damit kann die Teil-Verifikation für den jeweiligen Programmabschnitt

durchgeführt werden. Als Zusatznutzen sind derartige Konfigurationen gleichzeitig auch als Testbedingungen bei der Inbetriebsetzung der Steuerungen geeignet.

Die Teilbeweise der Beweisketten $\langle ... \boxed{K^{\beta}} ... \rangle$ könnten mit weiteren Teilbeweisen kombiniert werden, beispielsweise $\langle ... \boxed{K^{\beta}}, \boxed{K^{\chi}}, \boxed{K^{\delta}} ... \rangle$. Per Definition sind die SPS-Ausgänge exklusiv einzelnen Programmfunktionalitäten zugeordnet.

Daraus ergibt sich als Schlussfolgerung: Auf Grund des Umstandes, dass Eingangskonfigurationen temporär im Zyklus konstant gehalten werden, dass ausschließlich die Ausgangskonfigurationen beeinflussbar sind und dass auf Grund der Programmanalyse die Unabhängigkeit der betroffenen Ausgänge durch die Zuordnung zu Funktionsgruppen sichergestellt worden ist, gilt, dass jeder Teilbeweis für sich bereits das Verifikationsergebnis für die untersuchten Programmabschnitte zeigt.

Zuletzt muss noch sichergestellt sein, dass tatsächlich alle SPS-Ausgänge zugeordnet worden sind. In der Regel ist dieser Umstand durch Standardfunktionen der Programmeditoren für SPS-Programme einfach zu verifizieren.

8.6 Beitrag der Programmanalyse

Prinzipiell arbeitet eine SPS als Zustandsübergangs-Maschine, die entsprechend ihres Anwenderprogramms in typisch nahezu äquidistanten Zeitabschnitten Zustandsübergänge generiert. Innerhalb eines Zeitabschnitts läuft ein SPS-Zyklus ab, der völlig unabhängig vom aktuellen Systemzustand nur durch den Umstand gestartet wird, dass genau vor einem Zyklus der zeitlich davorliegende Zyklus beendet worden ist. Die Dauer eines Zyklus schwankt programmabhängig nur dann geringfügig, wenn im SPS-Programm bedingte Aufrufe und Sprungfunktionen vorgesehen sind. Das vereinfachte Modell der Zustandsübergangsmaschine zeigt Fig. 44.

Sichtbar für den Programmablauf sind nur jene Zustandsübergänge, die das Prozessabbild verändern, in denen z.B. Ausgangsreaktionen ausgelöst werden. Solange es keine Veränderungen im Prozessabbild gibt, werden gerade vorherrschende Zustände durch die im Zyklusbetrieb durchgeführte neue Berechnung nur bestätigt.

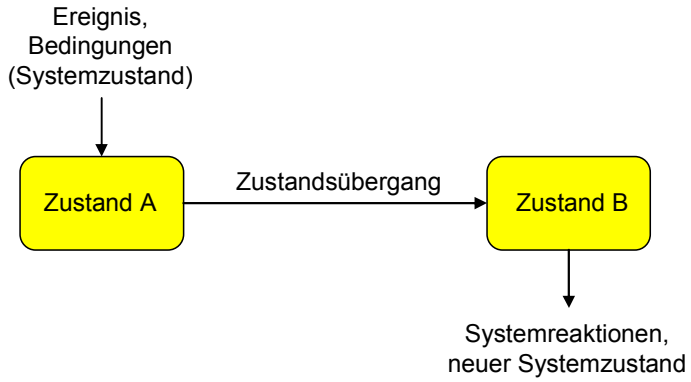


Fig. 44: SPS als Zustandsübergangsmaschine

Die Verifikation des SPS-Programms ist auf das Anwenderprogramm beschränkt. Ein Beispiel für eine Abgrenzung ist die Triggerung des Programmablaufs. Diese wird durch eine interne „Watch-Dog“ Funktion überwacht, um die Reaktionsfähigkeit des Systems sicher zu stellen. Im Fehlerfall würde der Programmablauf unterbrochen und das System in einen vordefinierten Sicherheitszustand überführt werden. Solche und weitere Systemfunktionen bleiben bei der Verifikation des SPS-Programms unberücksichtigt.

Das auslösende Ereignis für den Programmstart ist der oben genannte Zyklusbeginn (Fig. 44). Der genau zu diesem Zeitpunkt eingenommene Systemzustand sind die Anfangsbedingungen, die gleichzeitig die Anfangsbedingungen für die Verifikation bilden. Das SPS-Programm entwickelt aus den Anfangsbedingungen jenen Zustand, der als neuer Systemzustand die Systemreaktionen der SPS bewirkt. Genau dieser Endzustand ist die zu erfüllende Verifikationsbedingung.

Die Definition geeigneter Zustandsübergänge ist aus informellen Aufgabenstellungen bei der Programmerstellung vergleichsweise einfach, solange Teilproblemstellungen des SPS-Programms in den Programmcode umzusetzen sind, insbesondere dann, wenn diese Definitionen unmittelbar bei der Programmierung erfolgen. Die Begründung dafür ist einfach: genau die Summe derartiger informell beschriebener Zustandsübergänge bildet das Fundament für die vom Programmierer der SPS umzusetzende Aufgabenstellung. Hingegen ist eine rein formale Beschreibung von Aufgabenstellungen für SPS-Programme in der Praxis kaum anzutreffen, einfach auch deshalb, weil der Aufwand für eine Formalisierung im Vergleich zur Programmerstellung typischer Steuerungsabläufe unverhältnismäßig groß ist.

Das Handhaben eines exponentiell mit der Anzahl der Zustände wachsenden Zustandsraums bringt es mit sich, dass sehr schnell Grenzen der Berechenbarkeit in Verifikationssystemen erreicht werden. Ein logischer Schritt wäre daher die Segmentierung durch das Festlegen von

Grenzen, also in eine Kette von Zustandsübergängen. Prinzipiell können solche Grenzen beliebig festgelegt werden. Allerdings ist bei willkürlich festgelegten Programmsegmenten der Aufwand unverhältnismäßig hoch, die Systembedingungen zusammen mit den Systemreaktionen für solche Segmente festzulegen. Daher ist die Segmentierung von Teilsystemen vorteilhaft und der in dieser Arbeit eingeschlagene Weg, weil auch die Aufgabenstellungen für die Steuerungen in der Regel entsprechend gegliedert sind.

Zweck der Analyse des SPS-Programms ist das Bestimmen von funktional zusammenhängenden Programmteilen. Üblicherweise ist bei den meisten SPS-Programmen der funktionale Zusammenhang bereits dadurch gegeben, weil bei der Programmierung strukturiert eine Teilaufgabe nach der anderen bearbeitet wird. Dieser Umstand ist oft auch durch die räumliche Nähe von Programmsequenzen innerhalb eines SPS-Programms gegeben, ist jedoch bei der Programmerstellung nicht zwingend vorgeschrieben und muss daher untersucht werden. Diese Untersuchung wird speziell durch die Programmanalyse und besonders durch die Rückwärtsanalyse unterstützt. Damit lassen sich auch gegebenenfalls in wenig oder unstrukturierten SPS-Programmen voneinander unabhängige „Teilsysteme“ bzw. deren entsprechenden Programmteile abgrenzen. Auf diese Art kann jeder Zustandsübergang des Gesamtsystems auf geeignete Übergänge von Teilsystemen reduziert werden (Fig. 44).

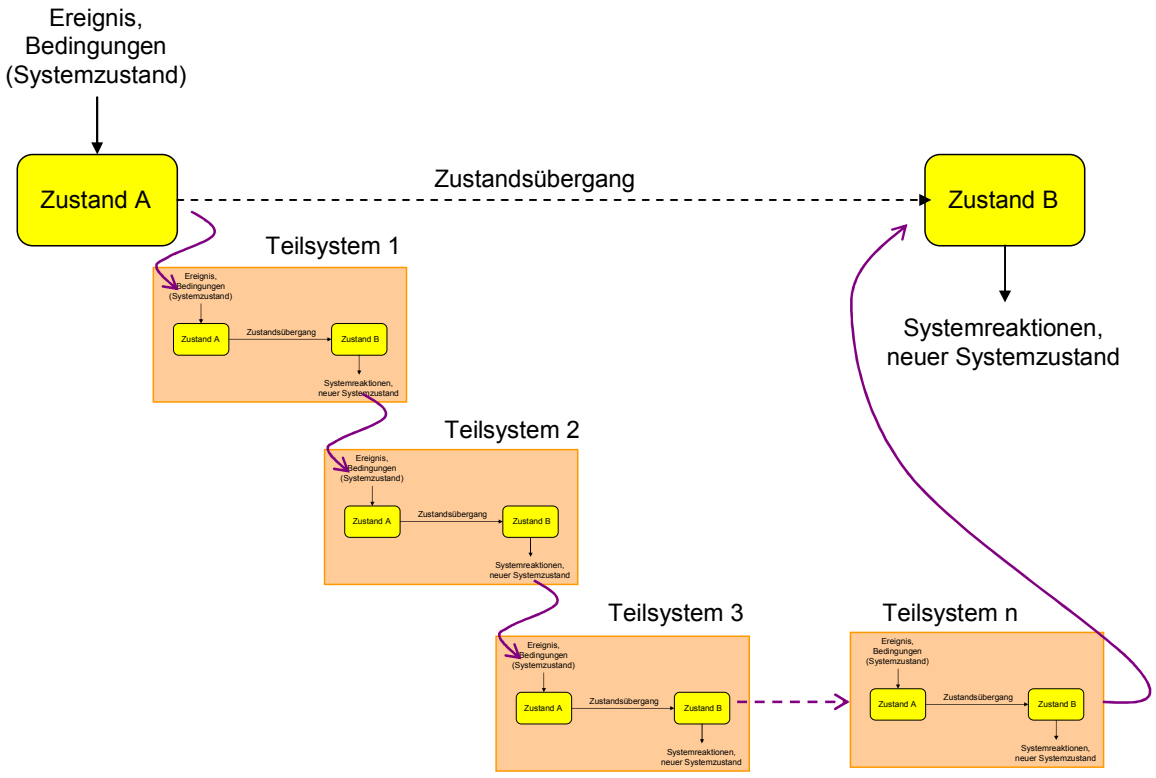


Fig. 45: Zustandsübergangsmaschine aus Teilübergängen

Allgemein gilt: die Mächtigkeit der Menge der Bedingungen $\mathcal{M}$ für jeden Zustandsübergang ist konstant und ergibt sich aus der Menge der Ein- und Ausgänge einer SPS zuzüglich der intern gespeicherten Variablen wie z.B. der Merker. Die Menge der Bedingungen für ein Teilsystem $\mathcal{M}^{Tx}$ ist eine Teilmenge aus $\mathcal{M}$. Überschneidungen in den Teilmengen sind zulässig. Die Mächtigkeit der Menge der Systemreaktionen $\mathcal{S}$ für jeden Zustandsübergang ist ebenfalls konstant. Die in der Menge der Teilsystemen $\mathcal{S}^{Tx}$ enthalten Elemente überschneiden sich nicht, sind also echte Teilmengen. Begründet ist das mit dem Verbot einer Doppelzuweisung von z.B. Ausgängen. Auf der anderen Seite ist damit auch die Vollständigkeit der Systemreaktion sichergestellt, wenn die Vereinigung aller Mengen der Teilsysteme $\mathcal{S}^{Tx}$ genau die Gesamtmenge der Menge der Systemreaktionen $\mathcal{S}$ ergibt, also die Vollständigkeitsbedingung für die Verifikation sicherstellt. Auch dieser Umstand kann durch die Programmanalyse des SPS-Programms festgestellt werden.

Ein Teilsystem könnte laut einer gegebenen Aufgabenstellung für das SPS-Programm auch mit anderen Teilsystemen interagieren. Ist dies der Fall, existieren Abhängigkeiten, die andere Teilsysteme beeinflussen. Für das SPS-Programm bedeutet das z.B., dass Berechnungsergebnisse, die durch den Programmfortschritt bei der SPS-Programmbearbeitung in den Konfigurationen repräsentiert sind, als Systemzustand des beeinflussbaren Teilsystems berücksichtigt werden müssen. Dieser Umstand ist die weiter oben genannte Überschneidung in den Teilmengen. Damit können diese Elemente als jene Referenzen von Zuständen identifiziert werden, die in mehreren Teilsystemen genutzt werden. Existiert eine derartige Abhängigkeit, bedeutet das, dass die Menge der Bedingungen für das betroffene Teilsystem $\mathcal{M}^{Tx}$ um die zusätzlich notwendigen Variablen zu erweitern ist.

Das Ziel der Analyse ist, alle Abhängigkeiten aufzudecken und sicher zu stellen, dass bei der Systemreaktion nach einem Programmdurchlauf alle Ausgangsgrößen berücksichtigt worden sind. Damit legen die Ergebnisse der Analyse den Rahmen fest, welche Referenzen für den Zustandsübergang jedes Teilsystems und damit des Gesamtsystems berücksichtigt werden müssen. Im Idealfall existieren keine Überschneidungen. In diesem Fall kann das SPS-Programm für das Teilsystem völlig unabhängig vom Gesamtsystem verifiziert werden. Existieren Überschneidungen, wird dadurch nur der Aufwand bei der Erstellung der Bedingungen erhöht, sonst läuft der Verifikationsvorgang gleichartig wie zuvor ab.

Die Technik des Teilens ist die Umkehrung der Entwicklung zu immer leistungsfähigen Steuerungssystemen. Beschränkten in der Anfangszeit Systemgrenzen wie die Anzahl der Ein- und Ausgangsignale den Leistungsumfang von SPS Steuerungssystemen, wurden mehrere SPS-Systeme zu einem Verbund zusammengeschlossen. In solchen Einzelsystemen ist die Anzahl der Zustände beschränkt und daher auch einfacher zu überblicken. Die Segmentierung eines umfangreicheren SPS-Programms in Teilsysteme kann daher vereinfacht so verstanden

werden, dass mehrere „virtuelle“ SPS eine Aufgabenstellung gemeinsam lösen. Diese virtuellen SPS könnten theoretisch sogar voneinander getrennt aufgebaut werden. Der Datenaustausch zwischen den Teilsystemen erfolgt durch die Zuordnung von gemeinsam genutzten Informationen.

9 Ergebnisse

Wenn ein SPS-Programm umfangreiche Aufgabenstellungen zu erfüllen hat, wird die Verifikation durch die Anzahl der zu beachtenden Zustände erschwert. Grund dafür ist die mit der Anzahl der Signalzustände exponentiell anwachsende Dimension des Eingangsvektors.[50] Daher existiert eine Schranke, bis zu der die Rechenleistung des Untersuchungssystems die vorgestellten Algorithmen nutzen kann. Um auch umfangreichere SPS-Programme zu verifizieren, sind zusätzliche Maßnahmen zu treffen.

9.1 Beitrag der Analyse

Hauptzweck der Programmanalyse ist das Aufsuchen von Bereichen, in denen Variable einer SPS innerhalb des SPS-Programms vermehrt genutzt werden. Solche Bereiche lassen darauf schließen, dass die dort realisierte Funktion eine abgrenzbare Teilfunktion innerhalb des SPS-Programms abbildet. Im Ergebnis betrachtet werden durch die Rückwärtsanalyse die den solchen Bereichen zugeordneten Ein- und Ausgangsinformationen festgestellt. Auf diese Art und Weise sind Teilbereiche eines SPS-Programms abgrenzbar, die in der Regel eine in sich geschlossene Teilaufgabe des SPS-Programms abdecken. Derartige Abgrenzungen sind deshalb von besonderem Interesse, weil sie vom Rest des SPS-Programms weitgehend unabhängig betrachtet werden können und damit auch einzeln einer Verifikation zugänglich sind.

Das Zerlegen eines SPS-Programms auf die kleinsten, verifizierbaren Einheiten wie z.B. ein Netzwerk gibt zwar Auskunft über ein Detail, ist aber aus dem Zusammenhang gerissen. Erst mit der Bedeutung der Programmfunktion sind auch Netzwerke bewertbar. Die semantische Programmanalyse erleichtert dem Programmierer geeignete SPS-Programmabschnitte voneinander abzugrenzen. Diese Abgrenzung kann zum Teil durch die informelle Beschreibung einer Aufgabenstellung erleichtert sein. Fehlen solche Beschreibungen insbesondere bei schlecht dokumentierten Programmen, muss zusätzlich vor einer Verifikation eine ausführlichere und damit oft recht zeitaufwendige Analyse einer Maschinen- oder Anlagenfunktion erfolgen.

9.2 Teilverifikation

In einem Gedankenexperiment wird eine Steuerungsaufgabe betrachtet, bei der einige Anlagenteile existieren, die voneinander unabhängig Teilaufgaben in einer Fertigungskette zu bearbeiten haben. Historisch gesehen wurden zu Beginn der Einführung von SPS derartige Auf-

[50] das *state explosion* Problem

gaben auf mehrere Steuerungen aufgeteilt, weil die damals zur Verfügung stehenden Systemressourcen für die Lösung der Gesamtaufgabe nicht ausgereicht hatten. Mittlerweile sind durch den Fortschritt der technischen Entwicklung derartige Einschränkungen nicht mehr gegeben. Wenn nicht aus anderen Gründen autarke Einheiten benötigt werden, können auch umfangreiche Aufgaben von einer einzigen SPS in ausreichender Performance gelöst werden.

Die Motivation wird daher in erster Linie durch die Verringerung der Komplexität begründet sein. Außerdem ist der Informationsaustausch zwischen Programmsegmenten wesentlich vereinfacht, wenn eine zentrale SPS die Lösung der kompletten Aufgabenstellung übernimmt. Um dennoch die Identifikation von unabhängigen oder bis zu einen definierten Grad abhängigen Programmsegmenten zu ermöglichen, wurden Werkzeuge für die Programmanalyse und deren Arbeitsweise diskutiert. Theoretisch ist die Segmentierung der SPS-Programme dem Programmierer bereits durch die Programmerstellung bekannt. Dennoch kommt es immer wieder vor, dass im Zuge der Überarbeitung und später bei der Inbetriebsetzung oder Programmoptimierung Ergänzungen oder Veränderungen an bestehenden Programmen vorgenommen werden, die zusätzliche Programmbeziehungen nicht ausschließen.

Zweck der Programmanalyse war es unter anderem, zusammenhängende bzw. auf einander aufbauende Programm-Konstrukte zu isolieren. Es ist davon auszugehen, dass eine in sich geschlossene Teilaufgabe einer SPS durch Programm-Konstrukte zu lösen ist, die aufeinander aufbauen und daher auch eng miteinander verknüpft sind. Dadurch werden bestimmte Bereiche in den Eingangs- und Ausgangsvektoren häufig benutzt, während der Rest nicht oder nur punktuell angesprochen wird. Allgemein gilt, dass eine feste Zuordnung zwischen SPS-Programm und Teilen der Eingangs- und Ausgangsvektoren existiert.

Eingangs- und Ausgangsvektoren bilden in der SPS Speicherbereiche ab. Die gegenseitige Beeinflussung von Programmsegmenten in einer SPS kann nur über diese Speicherbereiche erfolgen. Kritische Bereiche existieren nur dort, wo mehrere Programmsegmente den gleichen Speicherbereich nutzen.

Die Programmanalyse kann als Werkzeug Zuordnungen von Programmelementen zum Speicherbereich liefern. Als Nebenergebnis erhält der Nutzer auch Information darüber, ob es Überschneidungen zu anderen Speicherbereichen gibt. Ein Teil derartiger Überschneidungen könnte auch auf fehlerhafte Adressierung von Programmelementen zurückgeführt werden. Auch die geübte Praxis, ähnliche Programm-Konstrukte zu kopieren und erst im Zuge der folgenden Nachbearbeitung die Adressierung richtig zu stellen, führt fallweise zur nicht beabsichtigten Doppeladressierung. Ein entsprechender Hinweis hilft mögliche Fehler aufzudecken.

Betrachtet werden abgeschlossene Intervalle. Als Herausforderung an den Verifkator erweist sich, dass, wenn nur Teile der Informationen von den Eingangs- und Ausgangsvektoren genutzt werden, dann die Abfrage oder die Zuweisung im **input_vector** abgeändert werden sollte. Zwingend ist diese Forderung nicht, jedoch zweckmäßig, weil dann alle für den betrachteten Programmabschnitt nicht relevanten Zustandsinformationen auch in den Eingangs- und Ausgangsvektoren weggelassen werden können. In einer abgeänderten Formel **line(5, u, 'E 2.1', VE4, VE5)** mit einer ergänzenden Hilfsdefinition einer Adressrelation **adr(3, 'E 2.1')** zeigt dieser auf den Wert des abgefragten Elements im **input_vector**, wobei Adressinformationen an die jeweiligen Befehlszeilen gekoppelt sind.[51] Eine Definition etwa der im Beispiel vorgesehenen Logisch-**UND**-Verknüpfung im Verifikator müsste dann folgendermaßen aussehen:

```
line(_, u, Adr, VEalt, VEneu) :-
    temp_vector(Z),
    adr(Opi, Adr),
    nth1(Opi, Z, Optemp),
    and(Optemp, VEalt, VEneu).
```

Die Bezeichnungen **Adr, VEalt, VEneu** entsprechen der Interpretation der Adresse des aktuellen SPS-Befehls,[52] dem Verknüpfungsergebnis der vorhergehenden Berechnung bzw. dem Verknüpfungsergebnis der durchgeführten Operation.

Der Eingabeoperand **Opi** zeigt in den Zustandsvektor, der Index des dritten Elements in diesem Vektor hat den Wert des Eingangs **E 2.1** auf Grund der oben eingeführten Adressrelation **adr(3, 'E 2.1')**. **Optemp** zeigt in den temporären Zustandsvektor.

Für die Interpretation des Befehls wird zuerst der temporäre Zustandsvektor ermittelt (dies gilt für alle Befehle mit einem Operanden). **nth1** ist eine PROLOG-Funktion, die das Element mit dem Index **Opi** aus der Liste **Z** (diese ist die Repräsentation des Zustandsvektors) holt und mit **VEneu** unifiziert (vgl. [KP12]).

---

[51] Automatisch werden vom Hilfsprogramm die Verknüpfungsergebnisse **VEalt** und **VEneu** nummeriert. Der Zahlenwert für „**neu**“ entspricht der Nummer der aktuellen Befehlszeile, „**alt**“ der unmittelbar davor liegenden. Der **input_vector** ist Teil der Datenbank und wird während des Verifikations-Durchlaufes nicht verändert. Für diesen Zweck wurde der **temp_vector** eingeführt, der eine Kopie des **input_vectors** darstellt und zur Laufzeit des Verifikators bearbeitet wird. An Ende des Durchlaufs wird der **temp_vector** mit dem Ausgangsvektor verglichen (der eigentliche Verifikationsschritt) (vgl. KP12).

[52] Bei einer realen SPS sind SPS-Befehle sehr ähnlich aufgebaut. Ein SPS-Befehl besteht aus einem Operanden und seiner Adresse, wobei die Adresse durch einen Kennbuchstaben (dieser bezeichnet einen bestimmten Speicherbereich) und der Position im Speicherbereich gegeben ist.

Die Eingangs- und Ausgangsvektoren enthalten damit eine verdichtete Form der Zustandsinformationen, die der im untersuchten Programmabschnitt verwendeten entspricht und nicht mehr notwendigerweise die vollständige in Verwendung stehende Hardware abbilden. Der Verifikator muss um eine Liste von Fakten ergänzt werden, die von der Bauart

```
adr(1, 'E 1.0').
adr(2, 'E 1.1').
adr(3, 'E 2.1').
adr(4, 'M 0.0').
adr(5, 'A 0.5').
```

sind und den jeweiligen Abbildungen entsprechen. Zweckmäßigerweise wird diese Liste im Zuge der automatisierten Umwandlung des SPS-Programms in das entsprechende Prologprogramm erzeugt, um Fehler durch falsche Assoziationen (z. B. weist ein Eingang auf zwei unterschiedliche Positionen oder zwei Eingänge weisen auf dieselbe Position) oder durch Inkonsistenzen in der Schreibweise der Adressen zu vermeiden (z. B. `'E 1.0'` vs. `'E1.0'`).[53]

Diese „echten“ Hardware-Adressen müssen dann wegen der Syntax von PROLOG als Hochkomma-Literale dargestellt werden. Dadurch ist aber auch ihre Darstellung vollkommen frei wählbar. Ein nicht zu unterschätzender Nachteil in einer realen Anwendung nicht-trivialer Steuerungsprogramme ist, dass dadurch die Ausführungsgeschwindigkeit des Verifikators leidet, weshalb in der vorliegenden Arbeit auf diese Modifikation verzichtet wird und die in [KP12] vorgestellte Form des Verifikators zum Einsatz kommt. Dort ist eine vergleichbare Transformation vorab zwar im Rahmen der Umwandlung durch Hilfsprogramme schon vorgenommen worden, allerdings ohne irgendeine Art der Komprimierung, also linear über den gesamten verwendeten Adressraum der SPS.

Diskussion

Im Wesentlichen kann die Verifikation einer Teilfunktion innerhalb eines untersuchten SPS-Programms auch so verstanden werden, dass für jedes Teilprogramm eine eigene Eingangs- und Ausgangskonfiguration bestimmt wird, die genau die Speicherbereiche in diesen Programmabschnitten abbilden. Solange das SPS-Programm in sochen Abschnitten unverändert bleibt, sind daher auch keine Veränderungen der Konfigurationen möglich.

[53] Bei den meisten Editoren für SPS-Programme wird die Schreibweise bei der Eingabe automatisch korrigiert. Vielfach gilt das auch dann, wenn anstelle von Großbuchstaben Kleinbuchstaben verwendet worden sind.

Daraus kann abgeleitet werden, dass für alle Programmzyklen gilt: Wurde das zugehörige Teilprogramm bereits verifiziert, ergibt die neuerliche oder auch mehrfach wiederholte Verifikation keine neuen Ergebnisse. Es muss daher möglich sein, solche Programmabschnitte durch Übergangsfunktionen der Konfigurationen zu beschreiben, ohne dass ein zugehöriger Programmabschnitt neuerlich durchlaufen werden muss.

Die Idee ist, verifizierte Programmteile wie einen einzelnen Befehl im Verifikator zu behandeln, den zugehörigen SPS-Code mit einem Änderungsindex zu versehen und danach solange zu maskieren, bis der Änderungsindex eine Programmänderung im betroffenen Programmabschnitt anzeigt. Anzumerken ist, dass Programmeditoren für die Programmierung von SPS-Programmen auch Zeitstempel vorsehen, die für solche Zwecke ausgewertet werden könnten.

9.3 Testen

Als „*State Transition Machine*“ (STM) führt eine SPS Zustandsübergänge aus. Die grundlegende Idee der Verifikation ist, alle möglichen Zustandsübergänge zu verifizieren, indem das SPS-Programm Zustandsübergänge erzeugt, um diese mit der Vorgabe zu vergleichen.

Das in [KP12] diskutierte Konzept einer reaktionsschnellen SPS (RSPS) stellt ebenfalls eine STM dar. Dort wird der Verifikator dazu verwendet, entsprechend zugeordnet zu Eingangszuständen die passenden Ausgangszustände klassifiziert zu sammeln. In diesem Konzept emuliert der Verifikator eine herkömmlichen SPS und dient nur zur Erzeugung von Ausgangsvektoren für die reaktionsschnelle SPS. Damit bildet er zusammen mit einem Programmeditor einer herkömmlichen SPS das Programmiersystem für diese Steuerung. Dieser Zwischenschritt, das Programm zuerst mit einem Programmeditor einer herkömmlichen SPS zu erzeugen, bietet den Anwendern eine gewohnte Programmierumgebung, deren Programm in Folgeschritten erst in das RSPS-Programm konvertiert wird. Erst nach erfolgter Konvertierung[54] kann der Programmierer durch Untersuchung des geplanten Ausgangsabbildes feststellen, ob das konvertierte Programm den Anforderungen entspricht oder nicht.

Diskussion

Der Ablauf einer Verifikation mit dem Verifikator ist der Nachweis darüber, ob zu einer gegebenen Eingangskonfiguration die gewünschte Ausgangskonfiguration erreicht wird. Eingangskonfiguration und Ausgangskonfiguration werden aus der Aufgabenstellung abgelesen und dienen dabei als die Vor- bzw. Nachbedingung des finalen Verifikationsschrittes, also als Schlussfolgerung, ob der Beweis gelungen ist oder nicht. Wird dieser Schritt weggelassen,

[54] Das Systemverhalten wird durch den Verifikator auf einer abstrakten Ebene emuliert.

wird eine Ausgangskonfiguration erzeugt, über deren „Wert“ zunächst keine Aussage getroffen werden kann. Im Falle der RSPS dient die Ausgangskonfiguration bereits als in diesem Fall ungeprüfte Ausgabe, mit denen die Ausgänge gesteuert werden. Erst mit dem Testen kann die Korrektheit, in diesem Fall die richtige Zuordnung, überprüft werden.

Zusammengefasst kann festgestellt werden, dass eine „abgebrochene“ Verifikation Ausgangskonfigurationen und damit Ausgangsabbilder erzeugt.

Teststrategie

Der wesentliche Unterschied einer Verifikation zum Testen ist im Umstand begründet, dass im Rahmen einer Teststrategie eine Auswahl von Bedingungen getroffen wird, und dass anhand dieser Bedingungen Testszenarien bestimmt werden. Die ausgewählten Bedingungen stellen bestimmte Zustände dar, die eine durch das SPS-Programm gesteuerte Einrichtung einnehmen kann. Dies entspricht einer Anfangskonfiguration, die den weiter oben vorgestellten Eingangsvektoren entspricht. Beim Testvorgang selbst wird anhand einer Reihe derartiger Testszenarien überprüft, ob genau die geplante Reaktion einer durch die SPS gesteuerten Einrichtung gegeben ist und damit die vom Programmierer beabsichtigte Funktionalität erfüllt wird.

Derartige Testszenarien können bereits während der Programmerstellungsphase geplant werden. Mit Hilfe des vorgestellten Verifikators können diese aber auch im Zuge der Programmierarbeit, noch bevor das Programm ausgeführt wird, einem Test unterzogen werden. Dazu wird durch das Weglassen des finalen Verifikationsschritts anstelle von so genannten „wenn – dann“ Beziehungen „was wäre wenn“ Beziehungen definiert und aufgestellt.

Der Verifikator ist in diesem Fall als Simulator genutzt worden, wobei zur Simulation keine Hardware benötigt worden ist und damit die vollständige Programmsimulation vor einer Inbetriebnahme durchgeführt werden kann.[55]

[55] Im Verifikator ist ein abstraktes Modell realisiert, dass ausschließlich aus Fakten und Regeln definiert worden ist. Damit sind Struktur, Funktion und Verhalten des zu simulierenden Systems einer SPS bestimmt worden. Konkrete Zuweisungen von Werten, also die Parametrierung, erfolgt durch das Bestimmen der Konfigurationen. Die Übertragung auf das zu simulierende System, die SPS, ist eine Interpretation der formallogisch bestimmten Ergebnisse in Bezug auf das Verhalten einer SPS.

Diskussion

Solange von Verifikation gesprochen wird, ist das Bestimmen der Ein- und Ausgangskonfigurationen ein wesentlicher Teil der Verifikation. Andererseits kann überlegt werden, die Funktionalität des Verfikators nur zum Bestimmen von Ausgangskonfigurationen zu nutzen um die finalen Verifikationsschritte anstelle einer automatisierten Verifikation erst nachträglich durchzuführen. Es wäre zu untersuchen, ob dadurch der Gesamtaufwand einer Verifikation durch die Vereinfachung bei der Bestimmung von Ausgangskonfigurationen verringert werden kann.

9.4 Erweiterungen

In [KP12] ist das PROLOG-Programm[56] genannt „Verifikator“ und Überlegungen, die zu seiner Entwicklung geführt haben, ausführlich diskutiert. Zusätzlich sind dort weitere Hilfsprogramme und deren Programmkonzepte vorgestellt.

Eines dieser Programme dient als Übersetzungsprogramm für in Anweisungsliste geschriebene oder transformierte SPS-Programme in eine durch PROLOG bearbeitbare Form. Für die verschiedenen Steuerungsfamilien, die in der Regel „Dialekte“ der normativ vorgesehenen Schreibweise für SPS-Programmiersprachen nutzen, müssen die Befehlssätze entsprechend ausgetauscht werden. Ein universell nutzbares Übersetzungsprogramm muss konfigurierbar sein. Um die Position der Variablen im Eingangsvektor festzulegen, ordnet das Hilfsprogramm die Variablenwerte in aufsteigender Reihenfolge an: zuerst jene Variablenwerte, die den physikalischen Eingängen zugeordnet sind, unmittelbar anschließend die weiteren. Auslassungen, um etwa ungenutzte Adressbereiche ähnlich jener einer realen SPS freizulassen, sind in diesem Konzept nicht vorgesehen.[57] Diese gedrängte Anordnung kann jedoch insbesondere dann, wenn Teilbereiche untersucht werden, bei der Rekombination nach einer Teilverifikation zu Verwechslungen oder Fehlzuordnungen führen.

[56] PROLOG, *„programming in logic“* ist unter der Bezeichnung SWI-PROLOG als Freeware nutzbar. Die PROLOG Programme in [KP12] wurde auf Basis der Version 5.4 erstellt, mittlerweile existiert bereits eine Version 7.1.x
(Quelle: http://www.swi-prolog.org/versions.html, Abfragedatum 18.04.2015)

[57] Ein Grund dafür war auch, die Dimension der Vektoren auf das unbedingt nötige Ausmaß zu beschränken. Dadurch ist die anschauliche Darstellung der Vektoren im Text erleichtert worden (einzeilig).

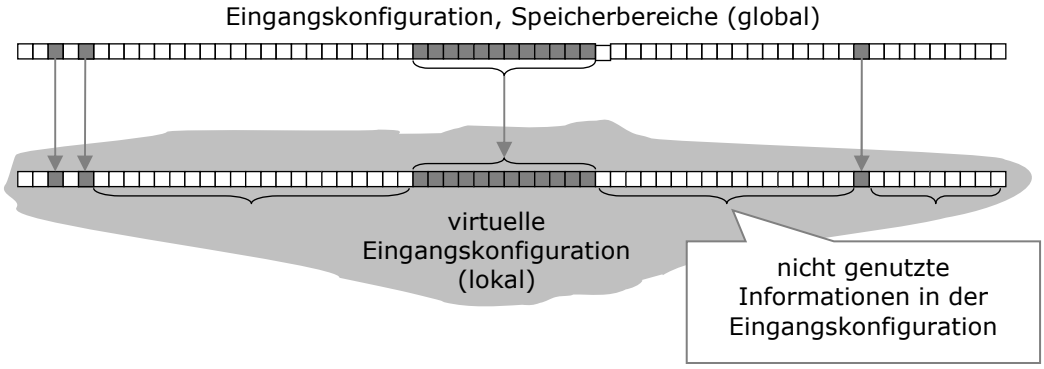


Fig. 46: Virtuelle Eingangskonfigurationen

Praktisch bedeutet das, dass eine Unterscheidung von den lokal genutzten Eingangskonfigurationen zu einer laut der Aufgabenstellung aufgestellten Konfiguration nicht notwendig ist (Fig. 46). Zweckmäßigerweise sollte das gesamte Speicherabbild der in einer SPS genutzten Bereiche für die Zwischenspeicherung von Eingangs- und Ausgangsabbildern vordefiniert werden. Das Problem dabei ist, dass die Konfigurationen einerseits nicht genutzte Elemente enthalten und daher in Folge dessen sehr groß werden. Andererseits sind dadurch Zuordnungsfehler beim Zusammensetzen von Teilkonfigurationen in vollständige Ein- und Ausgangskonfigurationen einfacher vermeidbar.

Auch der Verifikator selbst muss erweitert werden. In seiner in [KP12] vorgestellten Grundversion werden erfolgreich verifizierte Tupel nicht zwischengespeichert. Wie weiter oben dargestellt, sind die darin enthaltenen Wertpaare Bestandteile im Gesamtschema der für die Verfikation benötigten Ein- und Ausgangskonfigurationen.

9.5 Kombination der Teilergebissen

Ausgehend von einem SPS-Programm bietet sich die nachstehend vogeschlagene Vorgehensweise an: Zerlegung eines SPS-Programms in Programmblöcke mit einer frei wählbaren Anzahl von Variablen. Im Wesentlichen wird diese Anzahl durch die Komplixität einer Aufgabenstellung beeinflusst werden. Gesucht werden dabei in sich geschlossene Teilfunktionen, wie z.B. Programme, die einzelne Arbeitsschritte einer automatisierten Fertigung steuern.[58]

[58] In der Praxis wird bei der Inbetriebsetzung von Funktionen einer Maschine oder Anlage sehr ähnlich vogegengrn: Programmfunktionen werden einzeln schrittweise nacheinander überprüft. Wurde bereits bei der Programmerstellung des SPS-Programm entsprechend struktuiert, liegen damit „natürliche“ Gerzen des Unterungungsraumes bereits fest.

Solche vergleichsweise kleinen Programmsegmente reduzieren das sonst auftretende state explosion Problem. Für die Beantwortung der Frage, wie solche Blöcke interagieren oder ob diese isoliert betrachtet werden dürfen, sind einige Festlegungen zu treffen.

Festlegung: Definition der Programmsegmente

In der Regel wird sich ein Programmsegment an einer bestimmten Funktionalität einer Anlage oder Maschine orientieren, die bereits bei der Umsetzung einer Aufgabenstellung in ein zugehöriges SPS-Programm feststeht. Zu prüfen wird dabei sein, inwieweit die Anzahl der Variablen so groß ist, dass bereits solche Programmsegmente weiter unterteilt werden müssen. Die Festlegung auf eine Anzahl der Variablen wird im Wesentlichen von der Komplexität der Aufgabenstellung und der Erfahrung des Programmierers abhängen.

Festlegung: Lebendigkeitseigenschaft des SPS-Programms

Ein SPS-Programm ist aus mehreren Gründen nur temporär gültig. Am Ende der Phase „Programmerstellung“ bleibt es zunächst in der Regel bis zur Phase „Inbetriebsetzung“ unverändert, bei der Inbetriebsetzung erfolgen üblicherweise jedoch Anpassungen. Daraus folgt, dass im Code eines SPS-Programms bis zur Übergabe für den geplanten Einsatz einer Maschine oder Anlage Programmteile verändert werden. Eine Verifikation ist daher nur solange gültig, solange keine Veränderungen im SPS-Programm vorgenommen werden. Gelöst kann dieses Problem dadurch werden, dass z.B. die Historie des SPS-Programms, die durch Zeitmarkierungen beim Abspeichern des Programms automatisch zur Verfügung gestellt sind, bei der Verifikation berücksichtigt wird.

Das Gleiche gilt bei Verbesserungen des Programmablaufs. Bezogen auf das Beispielprogramm „Schere“ könnte überlegt werden, ob nach erfolgtem Schnitt die Schneidklingen zuerst geöffnet werden müssen bevor die Rückwärtsbewegung eingeleitet werden kann, oder ob etwa zur Beschleunigung des Ablaufs die Rückwärtsbewegung und das Öffnen der Schneidklingen zeitgleich erfolgen darf. Wird beim Vorgang des Schneidens sauber getrennt und das Schneidgut nicht zwischen den Schneidklingen eingeklemmt, ist diese Verbesserung möglich, wenn gleichzeitig sichergestellt werden kann, dass vor dem nächstfolgenden Schnitt die Schneidklingen wieder vollständig geöffnet sind.

Erzeugung von Zwischenergebnissen im Rahmen des Verifikationsprozesses

In den Programmsegmenten sind auf Grund der Variablenzuordnung nur bestimmte Teile des Adressraums einer SPS abgebildet. Welche Variablen betroffen sind, ist aus der Variablenbelegung ablesbar. Entsprechend der in Fig. 46 dargestellten Zuordnung werden virtuelle Ein-

gangskonfigurationen gebildet und im Zuge des Verfikationsprozesses mit den zugehörigen virtuellen Endkonfigurationen eines Programmabschnitts verifiziert.

Definitionsgemäß muß wegen der einmaligen Zuordnung von Ausgängen gelten, dass alle durch Zuweisungsoperationen gewonnenen Informationen der Endkonfiguration eines Programmsegments mit den korrespondierenden Informationen in der Ausgangskonfiguration übereinstimmen. Dieser Umstand ist unmittelbar einleuchtend, da im weiteren Programmverlauf des SPS-Programms die genannten Ausgänge nicht mehr schreibend bearbeitet werden können und daher ihren einmal eingenommenen Wert mindestens bis zum Ende eines betrachteten SPS-Programmzyklus beibehalten.

Wird daher nur ein ganz bestimmtes Programmsegment untersucht, ändert sich das Ergebnis der Untersuchung nicht, wenn alle übrigen Programmteile unbeachtet bleiben. Der gleiche Effekt mit einem identen Ergebnis würde erzielt werden, wenn die für einen bestimmten Untersuchungsfall unerheblichen Programmteile einfach weggelassen werden. Auf diese Art und Weise gewonnene Zwischenergebnisse kann als Zwischenverifikation bezeichnet werden.

Interpertation der Zwischenverifikationen

Im Idealfall ist der Programmcode verifizierter Programmsegmente ein zusammenhängender Abschnitt eines SPS-Programms. Ideal ist das deshalb, weil der Programmcode für weitere Zwischenverfikationen anderer Programmsegmente keinesfalls ein Verifikationsergebnis beeinflussen kann. Das bedeutet, dass in solchen Fällen keine weiteren zusätzlichen Untersuchungen notwendig werden.

Liegt der Idealfall vor, können ohne dass das Ergebnis verändert wird, alle Programmzeilen vor und nach dem zu untersuchenden Progammsegment weggelassen werden. Insbesondere wenn ein SPS-Prorgamm bereits bei der Programmerstellung strukturiert aufgebaut worden ist, liegt in der Regel der Idealfall vor.

In jedem Fall sind die Ausgangskonfigurationen gleichzusetzen mit dem Ergebnis einer Teilverifikation, die als Teile der Eingangskonfigurationen der im SPS-Programm nachfolgenden Programmsegmente zu berücksichtigen sind. Dies gilt jedoch nur für den Fall, dass die so ermittelten Zwischenwerte in den folgenden Programmsegmenten abgefragt werden. Jedenfalls gilt auf Grund der Forderung, dass schreibend bearbeitete Variable, die in den verbleibenden Programmsegmenten nicht mehr bearbeitet werden, an dieser Stelle bereits ihren Endwert eingenommen haben, dass also ein Teilverifikationsergebnis vorliegt, das in Teilen mit dem Ergebnis der Verifikation für die untersuchte Eingangskonfiguration bereits vollständig übereinstimmt.

Dargestellt wurde die schrittweise Entwicklung der Konfigurationen $K_i \rightarrow K_i^n$. Innerhalb eines SPS-Zyklus gilt für jeden Befehlsaufruf, dass aus der vor dem Befehlaufruf ermittelten Konfiguration durch die Befehlsbearbeitung die daraus folgende Konfiguration entsteht. Dabei beschreibt jede Konfiguration den Zustand der SPS zu einem genau definierten Zeitpunkt. Konfigurationen sind Abbilder eines gerade aktuellen Zustands der SPS und repräsentieren die Variablenzustände. Bekanntlich wird bei der Verifikation gefragt, ob ein vorgegebener Variablenzustand am Beginn des Programmzyklusses in den korrespondierenden Variablenzustand am Ende dieses Programmzyklusses überführt wird. Exakt derselbe Vorgang erfolgt bei der Zwischenverifikation, wobei hier Beginn und Ende innerhalb der Kette der Programmschritte beliebig festgesetzt werden können. Es gilt:

$$K_i \rightarrow K_i^n \Leftrightarrow K_i \rightarrow K_i^j \rightarrow \boxed{K_i^{j+1} \rightarrow K_i^m} \rightarrow K_i^{m+1} \rightarrow K_i^n$$

Ein für die Teilkonfigurationen als interessiernder Bereich festgelegter Teilabschnitt der Konfigurationen beginnt mit der Eingangskonfiguration K_i^j und endet mit der Ausgangskonfiguration K_i^m. [59]

Wird ein SPS-Programm betrachtet ist festzustellen, dass es bestimmte Punkte in der Befehlsfolge gibt, an denen das Bestimmen von Ein- und Ausgangskonfigurationen für Programmsegmente insoweit vereinfacht wird, weil diese Konfiugurationen als Teilaufgabenstellung die Basis für die Programmentwickung bilden. Im Wesentlichen ist die automatisierte Programmanlyse jenes Werkzeug, mit dem genau solche Grenzziehungen in SPS-Programmen unterstützt werden.

Zusätzlich ist festzustellen, dass je kleiner so definierte Programmsegmente sind, desto weniger Variablenzustände betroffen sein werden. Daraus folgt, dass Unterschiede zwischen Ein- und Ausgangskonfigurationen in abgekapselten Bereichen der Konfigurationen auftreten können während die übrigen Bereiche unverändert beibehalten werden. Damit steht fest, dass bei der Zwischenverifikation ein Teil der durch die Konfigurationen repräsentierten Variabenzustände unverändert bleiben und für den internen Ablauf des Verifikationsprozesses nicht relevant sind und daher auch unbeachtet bleiben können.

---

[59] Anmerkung: Der Laufindex n repräsentiert die Programmschritte innerhalb eind SPS-Zyklus. Der interessierende Bereich wird durch das Intervall] j+1 ... m] dargestellt. Auf Grund der eingangs festgelegten Schreibweise sind die Konfigurationen fest mit einem zugehörigen Programmschritt verbunden, gleichzusetzen mit der korrespondierenden Variablenbelegung einer Anweisung im SPS-Prograng. Aus diesem Grund ist die Eingangskonfiguration des betrachteten Programmabschnitts die zuletzt bestimmte Konfiguration des vorhergendenden Programmabschnitts.

Der Index i bezeichnet die unterschiedlichen Variablenbelegungen, die im Programmfortschritt des Ablaufs der gesteuerten Maschine oder Anlage aufscheinen.

Zusammenfassen von Zwischenergebnissen

Wie gerade festgestellt, liegen jeweils Teile der Verifkationsergebnisse nach erfolgter Zwischenverifikation vor. Diese Zwischenergebnisse der Verifikation von den Programmsegmenten werden als Gesamtergebnis zusammengesetzt, ebenso wie die erzeugenden Eingangskonfigurationen zusammengefasst werden.

Als zusätzliche Überprüfung können diese Eingangs- und Ausgangskonfiguartionen neuerlich mit dem kompletten SPS-Programm verifiziert werden. Für die gesteuerte Maschine oder Anlage bilden die Eingangs- und Ausgangskonfigurationen darüber hinaus alle Testfälle ab, die bei der Abnahme der Funktionen im Betrieb überprüfbar sind. Durch eine solche Überprüfung der Funktion wird die gesteuerte Maschine oder Anlage validiert.

9.6 Verifikation

Die vorgeschlagene Vorgehensweise lässt die stufenweise Verifikation von SPS-Programmen zu. Während Verifikationsmethoden wie Model-Checking von formalen Spezifikationen ausgehen, ist dies bei der hier vogeschlagenen Methode nicht notwendig. Andererseits schließt das Analysieren und Segmentieren des SPS-Programms zum Zweck der Definition von Ein- und Ausgangskonfigurationen nicht aus, diese als Modell für Verifikationsalgorithmen zu nutzen, die auf Techniken des Model-Checkings basieren.

Dass die Verifikationen von Teilprogrammen ausreichend ist, ist durch die auf Grund der durchgeführten Analyse festgestellte Vollständigkeit der Variablen im Zustandsraum einer SPS-Applikation begründet.

Theoretisch können nach erfolgter Teilverifikation sogar die zugrunde gelegten Teil-Konfigurationen zu einer Gesamtkonfiguration kombiniert werden und der Verifikationsvorgang auf das gesamte SPS-Programm angewendet werden. Ein Mehrwert dürfte sich allerdings daraus nicht ableiten lassen.

Andererseits ist die formale Definition aller Zustandsübergänge eines SPS-Programms geeignet, an die Konstruktion eines Modells zu denken, das genau solche Zustandsübergänge derart nutzt, um damit eine reaktionsschnelle SPS zu definieren. In einem solchen Modell werden alle Tupel der Zustandsübergänge < Eingangszustand (i), Ausgangszustand (i) > als Beschreibung der Gesamtfunktion einer gesteuerten Anlage oder Maschine in einem Speicher abgelegt. Wird ein bestimmter Eingangszustand erkannt, kann unmittelbar darauf der zugehörige Ausgangszustand abgelesen und ausgegeben werden. Damit kann das System unmittelbar und reaktionsschnell auf jede Veränderung der Eingangszustände folgerichtig reagieren (vgl.

[KP12]). Ein nicht uninteressanter Nebeneffekt ist, dass es keine Fehlfunktionen mehr geben kann, alleine deshalb, weil unerwünschte Zustandsübergänge im Speicher einfach nicht abgelegt sind und damit auch nicht aufgerufen werden können.

Anwendung

Das Hoare-Kalkül [HO69] beschreibt, wie ein Programmteil den Zustand z.B. einer Berechnung verändert. In der Schreibweise $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ bedeuten S ein Programm und die Zusicherungen $\mathcal{P}$ (als Vorbedingungen) bzw. $\mathcal{Q}$ (als Nachbedingungen). Wenn eine Vorbedingung zutrifft, gilt nach der Ausführung des Programmsegments die Nachbedingung. Nach diesem Prinzip arbeitet der in [KP12] vorgestellte Verifikator. Die Zusicherungen sind definiert als die Ein- und Ausgangsvektoren bzw. auch als Konfigurationen bei den Zwischenschritten. In den Ein- bzw. Ausgangsvektoren sind die Zustände eines Programms jeweils am Anfang und am Ende eines Programmdurchlaufs der SPS, dem SPS-Zyklus, definiert. Alle während des Programmdurchlaufes auftretenden Zustände sind als Konfigurationen bezeichnet. Konfigurationen $\{\mathcal{K}\}$ sind also die während der Programmbearbeitung auftretenden Zwischenergebnisse und gleichzeitig die Vorbedingungen für den nächstfolgenden Programmschritt.

Als Zerlegung des Programms werden funktionell im Sinne einer Aufgabenstellung zusammenhängende Programmfunktionen $S^{[i]}$ voneinander isoliert betrachtet. Erweitert man dazu die Schreibweise des Hoare-Kalküls entsprechend, gilt

$$\{\mathcal{P}\}\ S^{[1]}\ \boxed{\{\mathcal{K}^{[1]}\}\ S^{[2]}\ \{\mathcal{K}^{[2]}\}}\ \ldots\ \{\mathcal{Q}\}.$$

Dieser Ansatz wird bei der Verifikation eines Teilprogrammabschnittes ausgenutzt, in dem die Verifikation für einzelne Programmabschnitte in der Form $\boxed{\{\mathcal{K}^{[i-1]}\}\ S^{[i]}\ \{\mathcal{K}^{[i]}\}}$ durchgeführt wird. Ausgenützt wird dazu der Umstand, dass jedem Programmabschnitt Elemente des Ausgangsvektors exklusiv zuordenbar sind, die von den übrigen Programmabschnitten nicht bearbeitet werden und daher nach dem Durchlauf durch den interessierenden Programmbereich ihre finalen Zustandswerte eingenommen haben, die verifiziert werden können.

Die weitere Überlegung betrifft die Anzahl der in den Konfigurationen enthaltenen Zustandsinformationen. Für die Verifikation müssen grundsätzlich alle Zustände und alle Zustandsübergänge berücksichtigt werden. Bezogen auf den Teilprogrammabschnitt existieren innerhalb jeder Konfiguration jedoch Zustandsinformationen, die in einem zu untersuchenden Teilprogramm weder abgefragt noch bearbeitet werden. Im Teilprogramm sind diese Zustandsinformationen als Konstante zu betrachten und bleiben daher bei der Verifikation des gerade untersuchten Programmabschnitts unberücksichtigt. Der Zustandsraum wird dadurch derart in Bereiche untergliederbar, so dass einerseits die Überschaubarkeit gesteigert wird und

andererseits damit einhergehend auch die Komplexität der Untersuchung durch die getroffenen Eingrenzungen deutlich verringert werden kann.

Die durch die Konfigurationen beschriebenen Zustände sind gegliedert in solche, die vom betrachteten System (die gesteuerte Anlage oder Maschine) stammen und in solche, die im Programmablauf bearbeitet und gegebenenfalls verändert werden. Die zuerst genannten Zustände sind innerhalb der Programmbearbeitung in einem Programmzyklus des SPS-Programms als temporär statisch, die anderen als dynamisch zu bezeichnen, weil diese im Programmzyklus veränderbar sind. Die im Zyklus statischen Zustände entsprechend den Eingangsinformationen an die SPS, die als Rückmeldungen von der gesteuerten Anlage oder Maschine stammen, die dynamischen den rückgelieferten Steuersignalen.

Aus der Programmanalyse wurde bekannt, welche Elemente in den Konfigurationen bestimmten Teilbereichen der Anlage oder Maschine zuzuordnen sind. Insbesondere von Interesse sind die dynamischen Elemente, die zusammen genommen das vollständige Ausgangsabbild des Systems darstellen. Die Programmanalyse musste auch zeigen, dass dynamische Elemente im Ausgangsabbild eineindeutig den verifizierten Teilabschnitten und damit den zugehörigen Steuerungsfunktionen zugeordnet sind. Sind daher alle Teilabschnitte des SPS-Programms verifiziert, ist das gesamte Programm verifiziert. Die Rückwärtsanalyse ausgehend von den SPS-Ausgängen zeigt im Ergebnis alle einen bestimmten Ausgang beeinflussenden Programmelemente und deren Zustände, das bedeutet, dass die Konfigurationen eine vollständige Darstellung der von der SPS eingenommenen Zustände zu jedem Zeitpunkt der Bearbeitung auch innerhalb der Zyklen des SPS-Programms wiedergibt.

Praktische Durchführung

Erprobt wurde die Arbeit mit dem Verifikator bereits in [KP12] im Abschnitt „Praktische Anwendung“. Dort wurden einige Varianten eines einfachen SPS-Programms für eine Schleifmaschine als Beispiel einer gut abgrenzbaren Teilfunktion einer größeren Maschine untersucht. Letztendlich wurde mit dem Beispiel auch gezeigt, dass Flüchtigkeitsfehler im ursprünglichen SPS-Programm aufgedeckt wurden und dass nach Korrektur dieser Fehler das Programm erfolgreich verifiziert werden konnte.

Als Beispiel wurde dort das SPS-Programm einer Schleifmaschine herangezogen. Diese Schleifmaschine dient dazu, ein in eine Klemmvorrichtung eingespanntes Werkstück in der gewünschten Lage fest zu halten und dieses in einem Schleifvorgang zu kalibrieren. Als Einheit bewirkt die Schleifeinheit einen Teilbearbeitungsschritt in einer Bearbeitungskette mehrerer weiterer Maschinen und Werkzeuge.

Programmbeschreibung und Verifikation

Der Programmablauf ist gekennzeichnet durch eine einfache Schrittfolge. In dieser werden die Bewegungen „in Schleifposition stellen“ (1), „Schleifen“ (2), „Schleifscheibe abheben“ (3) und „in Grundstellung fahren“ (4) nacheinander ausgeführt. Zu Einstellzwecken der Maschine ist ein Anhaltepunkt nach dem Schleifvorgang vorgesehen. Programmeingaben zur Manipulation sowie Statusrückmeldungen von erreichten Positionen sind als SPS-Eingänge der Steuerung vorgesehen, Merker und Ausgänge zur Ansteuerung von Magnetventilen zum Auslösen von Bewegungsabläufen als SPS-Ausgänge behandelt. In den diskutierten Programmvarianten waren bis zu 17 Variable vorgesehen.

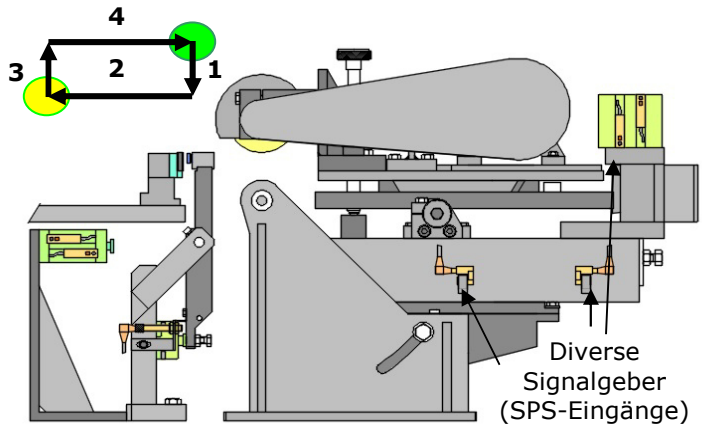


Fig. 47: Schleifmaschine mit Klemmvorrichtung

Fig. 47 zeigt die Schleifmaschine, die ähnlich der Darstellung aufgebaut ist. Die Klemmvorrichtung (links) kann separat betrachtet werden. Der zugehörige Bewegungsablauf ist symbolisch dargestellt. Die Ausgangslage und der Anhalte Punkt sind in dieser Darstellung farbig hinterlegt.

Der Verifikationsvorgang wurde mehrfach auch mit Programmvarianten durchgeführt. Analytisch wurden die benötigten Zustände als Ein- und Ausgangszustände (Ein- und Ausgangsvektoren) ermittelt. Beschrieben sind auch jene Festlegungen, die zur Vereinfachung getroffen worden sind. Das ebenfalls vorgestellte Programm AWLPRO generierte die Repräsentation des zugehörigen SPS-Programms in PROLOG, das als Grundlage für die Untersuchungen diente. Die Zuordnungstabelle für die Ressourcen wurde zur besseren Übersicht und Verfolgbarkeit in [KP12] durch „sprechende“ Namen ergänzt, die bei der Vorstellung der Aufgabenstellung eingeführt worden waren. Die bei der Programmerzeugung automatisch eingefügten Vektoren sind Platzhalter. Diese wurden entsprechend der Aufgabenstellung durch einen Satz aktueller Zustandsvektoren ersetzt (vgl. [KP12]).

Im ersten Testlauf wurde der oben erwähnte Flüchtigkeitsfehler sofort aufgedeckt. Nach Korrektur der Fehler lieferte die Verifikation die erwartete Bestätigung der Korrektheit des Programms.

Zusammenfassung

Im Ergebnis konnte festgestellt werden, dass für die Arbeit mit dem Verifikator ein schrittweises Vorgehen angezeigt ist. Durch die zusätzliche Überlegung, welche Zielzustände erreicht werden sollen, kann ein SPS-Programm bereits durch Nutzung des Verifikators auf Fehlerfreiheit und Vollständigkeit überprüft werden. Diese Methode geht über einen sonst üblichen Fehlersuchvorgang hinaus (vgl. [KP12]). Im Weiteren können die vordefinierten Zustandsübergänge als Teststellungen auch später im realen System für die Validierung der Funktionalität genutzt werden.

10 Schlussbetrachtungen

Ausgehend von der Tatsache, dass in jeder algorithmischen Vorgehensweise in Rechnersystemen, wo die Entwicklung eines Ergebnisses entlang des Programmpfades exakt vorgegeben ist, daraus abgeleitet werden kann, dass unabhängig davon, an welcher Stelle ein Algorithmus angehalten wird um diesen zu einem späteren Zeitpunkt fortzusetzen gilt: Wurde der Datensatz, der als Prämissen für den betrachteten Algorithmus nicht verändert muss notwendigerweise das Ergebnis einer solchen Berechnung eineindeutig wiederholbar sein.

Schlussfolgerung: alle Zwischenwerte, die entsprechend der betrachteten Vorschrift bestimmt worden sind, sind bezogen auf einen bestimmten Rechenschritt bei unveränderten Angaben identisch. In Rechnersystemen sind derartige Zwischenwerte und Angaben Bereiche, die im Datenspeicher abgebildet sind, während der Algorithmus, also die Rechenvorschrift im Codebereich gespeichert wird. Soll also eine Rechenvorschrift verifiziert werden, wird diese entsprechend der Angaben (Anfangswerte) angewendet und die Korrektheit der Ergebnisse als Aussage für die Verifikation des Algorithmus genutzt. Andererseits könnte auch die Entwicklung der im Datenspeicher abgebildeten Zwischenergebnisse genutzt werden. Wenn ausgehend von einem Anfangswert nur die Zwischenergebnisse betrachtet werden, und diese wiederholgenau bei jeder Anwendung eines in diesem Fall nicht notwendigerweise bekannten Algorithmus deckungsgleich vorliegen, folgt daraus, dass der genutzte Algorithmus korrekt sein muss.

In einem weiteren Gedankenschritt kann gefragt werden: welchen Einfluss haben die Zwischenergebnisse. Dem Grund nach sind die Zwischenergebnisse nur dann notwendig, wenn die Berechnung aus irgendeinem Grund unterbrochen wird, um diese zeitversetzt fortzusetzen. Unter den weiter oben diskutierten Voraussetzungen ist das zulässig und wird das Ergebnis nach der Fortsetzung einer unterbrochenen Berechnung auch nicht beeinflussen.

Wendet man diese Überlegung auf das Modell einer Rechenanlage an mit dem Blickwinkel auf eine SPS, wird eine Berechnungskette nur dann unterbrochen werden müssen, wenn etwa ein wichtiges Ereignis eine Systemreaktion erfordert. Der dafür typisch genutzte Mechanismus sind Interrupts. Zusammengefasst: Zwischenergebnisse sind dann notwendig, wenn nach einem Interrupt ein Rechenprozess an der Unterbrechungsstelle mit dem Zustand bei der Unterbrechung fortgesetzt werden soll.

Andererseits kann überlegt werden: Eingangs- und Ausgangszustände sind Wertpaare, die für die Verifikation genutzt werden. Auf Grund der eindeutigen Zuordnung, das für jeden Eingangszustand genau ein Ausgangzustand existiert, kann jeder Eingangszustand als die Adres-

sinformation für einen ihm zugeordneten Ausgangszustand dienen. Mit dieser Modellvorstellung lässt sich ein reaktives System konstruieren, dass überhaupt nichts berechnet, sondern wie aus einer sehr großen Wissensdatenbank zugehörige Ergebnisse parat hat. Der benötigte Zeitraum für das Aufsuchen von Lösungen reduziert sich auf Speicherzugriffszeiten. Weil in diesem Konstrukt Ergebnisse nur mehr abgerufen werden und damit vergleichsweise auch sehr schnell zur Verfügung stehen, kann auf Interrupts verzichtet werden. Abhängig von den Speicherzugriffssteuerungen und den Suchalgorithmen für Speicherstellen kann praktisch unmittelbar reagiert werden.[60]

Eine weitere Deutungsmöglichkeit für Wertpaare von Eingangs- und Ausgangszuständen ist, diese als intelligentes Informationssystem zu betrachten. Alle Reaktionen einer SPS sind als Wissensrepräsentation abgebildet. Gewöhnlich werden derartige Systeme mit dem Begriff eines Expertensystems in Zusammenhang gebracht. Das darin deklarativ repräsentierte Wissen wird in Form von einfachen Produktionsregeln daraus abgerufen.[61] Dabei genügt es, die in der Wissensbasis voneinander unabhängigen, überschaubaren Wissenseinheiten zu verstehen. Diese Basis kann auch vergleichsweise einfach korrigiert werden, weil es genügt, die Richtigkeit der einzelnen Wissenseinheiten zu überprüfen und diese gegebenenfalls zu ändern. Auch das Aktualisieren der Basis ist möglich, etwa um Sicherheitsreaktionen auszulösen.

Ähnlich dem klassischen V-Modell für die Softwareentwicklung kann die Entwicklung eines SPS-Programms durchgeführt werden. Der Anforderungsdefinition des V-Modells steht am Ende der Entwicklungsphase die Abnahme eines Gesamtsystems gegenüber: der zu steuernden Maschine oder Anlage. Bei der Verifikation wird nur überprüft, ob die Software mit seiner Spezifikation übereinstimmt, bei der Validation zusätzlich, ob der Einsatzzweck erfüllt ist. Erst die Abnahme als letzter Schritt eines Entwicklungsprozesses für SPS gesteuerte Systeme überprüft das System auch funktionell auf Korrektheit. In dieser Arbeit wird nur die Vorstufe zur Validation, die Verifikation erreicht.

Der Verifikator ist dabei ein zusätzliches Werkzeug zur Programmieroberfläche von SPS gesteuerten Systemen und soll bereits schrittweise beim Programmentwicklungsprozess eines SPS-Programms eingesetzt werden. Der dabei simulierte Systemtest erfolgt noch ohne angeschlossene Peripheriegeräte. Als Verifikationsschritte werden dazu die Systemreaktionen auf

---

[60] Bezeichnet als reaktionsschnelle speicherprogrammierbare Steuerung (RSPS) sind weitere Überlegungen dazu in [KP12] gemacht worden. Insbesondere für Aspekte zur Echtzeitproblematik in reaktiven Systemen ergeben sich erweiterte Möglichkeiten in der Anwendung für als kritisch einzustufende Funktionalitäten.

[61] Das deklarative Wissen wird auf Grund der Bestimmung der Ein- und Ausgangszustände produziert, Produktionsregeln sind z.B. Implikationen, also wenn - dann Beziehungen.

wechselnde Eingangsbedingungen überprüft. Das Ergebnis der Verifikation kann dabei den sonst üblichen Systemtest an dieser Stelle der Überprüfung zum Teil [62] ersetzen und gibt Auskunft, ob die geplanten Systemfunktionen korrekt implementiert worden sind.

Als weitere Vorstufen zur Verifikation sind im V-Modell der technische Systementwurf und Spezifikationen der Komponenten zu identifizieren. In der Regel sind SPS-Systeme aus Baugruppen zusammengestellt, die konfiguriert werden. Die Konfiguration legt dabei fest, welche Ein- bzw. Ausgangsadressen der Steuerung zugeordnet sind. Für den Test werden dazu in der Form von so genannten Pin-Tests die ankommenden und abgehenden Signalzustände zur gesteuerten Maschine oder Anlage überprüft. Die Summe dieser Signalzustände sind Abbilder von Zuständen und in den Ein- bzw. Ausgangsvektoren und Konfigurationen gespeichert und bildet die Basis für die Verifikationsschritte. Typisch erfolgt die Programmentwicklung für SPS-Steuerungen offline. Das bedeutet, dass der Pin-Test, der die korrekte Zuordnung der erfassten Zustände sicherstellt, erst direkt an der Maschine oder Anlage erfolgen kann. Die Verifikation mit dem Verifikator ist daher erst dann vollständig, wenn auch im Bereich der Erfassung der Systemzustände alle Zuordnungen auf Korrektheit überprüft worden sind.

Der Verifikator selbst kann interaktiv im Wechselspiel mit der Erstellung von SPS-Programmen oder Programmsegmenten genutzt werden. Durch das Festlegen der dazu notwendigen Eingangs- bzw. Ausgangsbedingungen werden gleichzeitig jene Testfälle bestimmt, die bei der Validation der Anlage genutzt werden können. Vorteilhaft erscheint dabei, dass einerseits der Programmierer seine Gedankengänge dokumentiert und gleichzeitig eine Rückmeldung darüber erhält, ob die Umsetzung in das SPS-Programm folgerichtig durchgeführt worden ist.

Der Beitrag der Programmanalyse ist einerseits Fehler in der Programmierung aufzudecken, andererseits bei der Eingrenzung von Programmsegmenten die Zugehörigkeiten von verwendeten Adressen zu untersuchen. Gedacht sind die beschriebenen Funktionalitäten der Programmanalyse als unterstützende Funktion. Eine Entscheidung darüber, ob Änderungen im SPS-Programm vorzunehmen sind oder nicht, bleibt dem Programmierer vorbehalten. Wesentlich dabei ist, dass sämtliche Ausgänge der SPS genau einem Programmsegment zugeordnet sind.

Die Mächtigkeit der sich daraus ergebenden (Teil-) Zustandsmengen wird im Wesentlichen nur durch die Überschaubarkeit auf Grund der Anzahl der kombinatorisch zu berücksichtigen Einzelelemente gegeben sein. Neben den vom Nutzer des Verifikators bestimmten Zustandskombinationen können durch Expansion von als „dont care“ definierten Zuständen auch die

[62] Zum Teil nur deshalb, weil auch die korrekte Zuordnung der Ein- und Ausgangssignale sichergestellt sein muss. Dieser Überprüfungsschritt kann erst bei der Inbetriebsetzung einer Maschine oder Anlage erfolgen.

darüber hinaus theoretisch möglichen Zustände und Zustandskombinationen in die Berechnungen durch den Verifikator mit einbezogen werden.

Ergänzungen

Zur Verbesserung der Nutzung des Verifikators bietet es sich an, als ein weiteres Hilfsprogramm einen SPS-Emulator als off-line Testsystem zu entwickeln. Der SPS-Emulator soll die Möglichkeit bieten, kleinere Programmabschnitte auf Funktionalität vor einer realen Inbetriebsetzung auf die gewünschte Funktionalität zu überprüfen (in einer Minimalversion mit Hilfe eines einfachen Ein-Ausgabefensters). Es sind mehrere Betriebsarten des Emulators möglich, die über eine Bedienoberfläche gesteuert werden. Aufgabenstellung ist, ein SPS-Programm zu emulieren, damit es auch off-line getestet werden kann. Auf Basis der Programmsprachkonzepte nach der DIN EN 61131-3 (IEC 1131-3) soll für SPS-Programme von Siemens Steuerungen (Programmiersprache STEP7) der Emulator konzipiert werden, der STEP7 SPS-Programme in der Darstellungsart Anweisungsliste einlesen kann und mit dem die eingelesenen Programme getestet werden können.

Der „Programmdialekt“ STEP7 ist dabei die Ausgangsbasis. Es soll jedoch auch möglich sein, den Befehlssatz im Emulator auszutauschen, um auch andere Programmdialekte anderer Hersteller implementieren zu können. In der ersten Entwicklungsstufe wird der Grundbefehlssatz auf Steuerungsprogramme (Bit-Verknüpfungen) ergänzt mit Zeit- und Zählerfunktionen beschränkt. In dieser Entwicklungsstufe sollen Eingabe-„Bitmuster“ zeilenweise über eine Eingabemaske eingelesen werden können und die zugehörigen Ausgabe-„Bitmuster“ angezeigt werden. Dies entspricht den für die Verifikation benötigten Ein- bzw. Ausgabekonfigurationen.

Wünschenswert ist eine intuitiv gestaltete Bedienoberfläche, die mehrere Betriebsarten zulässt. Vorgesehen ist eine Bedienerführung mit Hilfefunktionen. Hauptfunktion ist die Ein- und Ausgabe und das Monitoring der Ergebnisse.

Vogesehende Steuerungsfunktionen für den Emulator:

- Eingabelisten (Szenarien): Eingabe von Tabellen, in der in Listenform unterschiedliche Eingabekonfigurationen abgespeichert sind. Durch Weiterschaltung werden die Sequenzen in wählbarer Reihenfolge im Normalbetrieb bearbeitet. Weitere Optionen für die Steuerung des Emulators sind für die Programmanalyse des SPS-Programms vorzusehen:
- Normalbetrieb (continuous): das SPS-Programm arbeitet zyklisch so schnell wie möglich. Die Zykluszeit wird berechnet und angezeigt.
- Debug-Modus 1 (single-step): das SPS-Programm arbeitet im Einzelschrittbetrieb Programmanweisung für Programmanweisung ab und zeigt die Veränderungen der Pro-

grammabbilder an. Die Weiterschaltung erfolgt durch eine Eingabefunktion (z.B. Tastendruck, Maustaste).

- Debug-Modus 2 (one-cycle): das SPS-Programm arbeitet im Zyklusbetrieb einen Programmdurchlauf ab und zeigt die Veränderungen der Programmabbilder an. Der neuerliche Start eines Zyklus erfolgt durch eine Eingabefunktion (z.B. Tastendruck, Maustaste).
- Debug-Modus 3 (one-cycle-single-step-triggered): das SPS-Programm arbeitet im Zyklusbetrieb einen Programmdurchlauf ab und zeigt die Veränderungen der Programmabbilder an. Um die schrittweise Bearbeitung nachvollziehen zu können, ist ein einstellbarer Zeittakt für die schrittweise Weiterschaltung vorzusehen (z.B. Verzögerung eine Sekunde). Der neuerliche Start eines Zyklus erfolgt durch eine Eingabefunktion (z.B. Tastendruck, Maustaste).

Literatur

[AB09] Allen Bradley: „Logix5000 Controllers“, Publication 1756-QS001E-EN-P - October 2009

[Bc et.al.08] Baier, Christel, Katoen, Joost-Pieter : “Priciples od model checking.”. MIT Press 2008, ISBN 978-0-262-02649-0.

[Bd et al. 08] Bender Darlam Fabio, Benoît Combemale, Xavier Crégut, Jean Marie Farines, Bernard Berthomieu, and François Vernadat. “Ladder Metamodeling and PLC Program Validation through Time Petri Nets.” Model Driven Architecture–Foundations and Applications, 121–36. Springer, 2008. http://link.springer.com/chapter/10.1007/9783540691006_9.

[Br et al 08] Bhoi, R. R., S. Hansda, P. Venkateswaran, S. K. Sanyal, R. Nandi: “PLC veriſcation using symbolic model checking”, 2008. http://www.ee.co.za/wpcontent/uploads/legacy/AutT%20Symbolic.pdf

[Bs et al. 12] Biallas, Sebastian, Jörg Brauer, and Stefan Kowalewski. “Arcade. PLC: A Verification Platform for Programmable Logic Controllers”. Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on, 338–41. IEEE, 2012. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6494950.

[CC10] Roberto Cavada, Alessandro Cimatti, Marco Benedetti, Marco Pistore, Marco Roveri and Roberto Sebastiani. NuSMV: a new symbolic model checker. http://nusmv.itc.it/, 2010

[Cl et al.09] Edmund M. Clarke, Orna Grumberg and Doron Peled: Model Checking”, December 1999 | ISBN: 9780262032704

[CP97] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Electronic Notes in Theoretical Computer Science 6 (1997). URL: http://www.elsevir.nl/locate/entcs/volume6.html, (Abfragedatum April 2010)

[FG07] G. Fendoglu. Überführung eines AWL.Modells in ein NuSMV-Modell. Master thesis. Technische Universität Braunschweig. 2007

[FN92] Nissim Frances. Program Verification. Addison-Wesley Publishers Ltd. 1992

[HG97] Gerald J. Holzmann. The Model Checker SPIN. IEEE volume 23, 1997: Transactions on Software-Engineering.

[HO69] C. A. R. Hoare: „An axiomatic basis for computer programming“ Communications of the ACM. 12(10): 576–585, Oktober 1969.

[HP85] D. Harel, A. Pnueli: development of reactive systems. Logic and Models of Concurrent Systems, Vol. 13 of NATO ASI Series, Springer. 1985

[Hr13] Hametner, Reinhard: “Test Driven Software Development for Improving the Quality of Control Software for Industrial Automation Systems”, Dissertation TU Wien, Fakultät für Elektrotechnik und Informationstechnik, 2013

[KM96] Keneth L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers. 1996

[KG17] Kucera, G.: „Automatisierung in der Produktion. Steuerung von Fertigungsautomaten“, Buch, FH Campus Wien, 2017

[KG13] Kucera, G.: „Programmanalyse von SPS-Programmen als Teilschritt für die automatisierte Verifikation“, Konferenzbeitrag, Tagungsband zum 7. Forschungsforum der österreichischen Fachhochschulen, Seite 47-54, FH Vorarlberg GmbH, ISBN: 978-3-86573-709-0, 2013

[KP12] Kucera, G. Paulis, H.: „Schritte zur Verifikation von SPS-Programmen“, FH Campus Wien, ISBN: 978-3-902614-24-7, 2012

[KS09] Kleuker, Stephan: „Formale Modelle der Softwareentwicklung, Model-Checking, Verifikation, Analyse und Simulation“, 1. Auflage, Vieweg+Teubner | GWV Fachverlage GmbH, ISBN 978-3-8348-0669-7, Wiesbaden 2009

[Ks et al. 99] Kowalewski, S., N. Bauer, J. Preußig, O. Stursberg, H. Treseler: “An Environment for ModelChecking of Logic Control Systems with Hybrid Dynamics”. Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium, 97–102. IEEE, 1999. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=808631.

[MC10] Carnegie Mellon. Model Checking @ CMU. (Symbolic Model Verifier, release SMV 2.5, last version 2001). http://www-2.cs.cmu.edu/~modelcheck/smv.htmlllan. (Abfragedatum März 2012)

[MI12] Misubishi: https://my.mitsubishi-automation.com/downloads/view/doc_loc/6641/208660.pdf?id=6641&saveAs=0&form_submit=Jetzt+anzeigen, Abfragedatum März 2012

[MW99] Mader, Angelika, Hanno Wupper: “Timed Automaton Models for Simple Programmable Logic Controllers”. RealTime Systems, 1999. Proceedings of the 11th Euromicro Conference on , 106–13. IEEE, 1999. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=777456.

[NF92] Nissim Francez: „Program Verification“, Addison-Wesley, ISBN 0-201-41608-5, 1992

[OM11] Omron: Programmierhandbuch „CX-Programmer V9 Operation Manual.pdf“, Revision 2011

[OW12] Olderog, E; Wilhelm, R: „Turing und die Verifikation“, Universität Oldenburg Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken,
Informatik Spektrum 35, Seite 271-279, 2012

[Po et al. 07] Pavlovic, Olivera, Ralf Pinger, Maik Kollmann: “Automated Formal Verification of PLC Programs Written in IL”. Conference on Automated Deduction (CADE), 152–63, 2007.

[RK98] M. Rausch, B. H. Krogh: „Formal verification of PLC programs.” American Control Conference, Philadelphia. 1998

[RS02] Richard Šusta. Verification of PLC Programs. Doctoral Dissertation. CTU-Faculty of Electrotechnical Engineering Prague. 2002

[SE11] Schneider-Electric: „LL984-Editor“, Referenzhandbuch, LL984-Sonderfunktio-nen, EIO0000000803.01, Ausgabe 07/2011

[SF13] Soliman, D.; Frey, G.: „Verifikation und Validierung sicherheitsgerichteter SPS-Programme“, Seite 109 – 126, erschienen im Sammelband „Funktionale Sicherheit, Echtzeit 2013“ der Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V., VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG), Boppard, 21. und 22. November 2013

[SI11] Siemens: Programmierhandbuch „SIMATIC STEP 7 Basic V11.0 SP2 System-handbuch“, Dezember 2011

[VÖ98] Völker, N.: „Ein Rahmen zur Verifikation von SPS-Funk-tionsbausteinen in HOL“ Shaker Verlag 1998

[XL01] Xiying Weng, L., Litz, L.: “Model checking of signal interpreted Petri nets”. 2001 IEEE International Conference on Systems, Man and Cybernetics 2001, Vol.4, pp.2748-2752

Norm

[1] Norm DIN IEC 60050-351:2014-09: „Internationales Elektrotechnisches Wörterbuch - Teil 351: Leittechnik“ (IEC 60050-351:2013), 2014

Anhang A: Algorithmen für SPS-Befehle

In der Folge ist der Befehlsatz der dem Modell zugrunde gelegten SPS zusammengestellt. Entsprechende Befehle werden auch im Verifikator verwendet.

Logische Verknüpfungen

`Wertebereich [0, 1]`

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i : **L** $[s_j]$	**LOAD** s_j	**if** s_j == 1 **then** z_i := 1 **else** z_i := 0 **fi**
c_i : **LN** $[s_j]$	**LOAD NOT** s_j	**if** s_j == 0 **then** z_i := 1 **else** z_i := 0 **fi**
c_{i-1} :.... c_i : **U** $[s_j]$ c_{i+1} :....	z_{i-1} **AND** s_j	Berechungsergebnis der vorherigen Operation: z_{i-1} **if** z_{i-1} == 0 **then** z_i := 0 **else** **if** s_j == 0 **then** z_i := 0 **else** z_i := 1 **fi** **fi**
c_i : **UN** $[s_j]$	z_{i-1} **AND NOT** s_j	**if** z_{i-1} == 0 **then** z_i := 0 **else** **if** s_j == 0 **then** z_i := 1 **else** z_i := 0 **fi** **fi**
c_i : **O** $[s_j]$	z_{i-1} **OR** s_j	**if** z_{i-1} == 1 **then** z_i := 1 **else** **if** s_j == 1 **then** z_i := 1 **else** z_i := 0 **fi** **fi**

Tab. 8: Abfragen und Verknüpfungen

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i : **ON** $[s_j]$	z_{i-1} **OR NOT** s_j	**if** z_{i-1} == 1 **then** z_i := 1 **else** **if** s_j == 0 **then** z_i := 1 **else** z_i := 0 **fi** **fi**
c_i : **U (**	**AND (**	Berechnungsergebnis der vorhergehenden Operation und das Kommando werden zwischengespeichert [Z^+] **push** [z_{i-1}] **push 'AND'**
c_i : **O (**	**OR (**	**push** [z_{i-1}] **push 'OR'**

c_i : **)**	**)**	**pop**
		if 'AND' then
		pop
		if [z_{i-1}] == 0 **then** z_i := 0
		else
		if z_{i-1} == 0 **then** z_i := 0
		else
		z_i := 1
		fi
		fi
		else
		if 'OR' then
		pop
		if [z_{i-1}] == 1 **then** z_i := 1
		else
		if z_{i-1} == 1 **then** z_i := 1
		else
		z_i := 0
		fi
		fi
		fi
		fi

Tab. 9: Klammerbefehle

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i :=[s_j]	**EQUAL** s_j	Zuweisungsoperation **if** z_{i-1} == 1 **then** s_j := 1 **else** s_j := 0 **fi**
c_i :**S**[s_j]	**SET** s_j	Zuweisungsoperation Befehl SETZEN **if** z_{i-1} == 1 **then** s_j := 1 **fi**
c_i :**R**[s_j]	**RESET** s_j	Zuweisungsoperation Befehl RÜCKSETZEN **if** z_{i-1} == 1 **then** s_j := 0 **fi**

Tab. 10: Zuweisungsoperationen

Erweiterungen im Wertebereich

„Don't cares" filtern Variablenbereiche, die bei der Verifikation eines Teilprogramms ohne Belang sind. Idealerweise sollten Variable, deren Wert mit „don't care" vorgegeben war, durch das Programm nicht verändert werden. Werden derartige Variablen dennoch verändert, muss die dafür verantwortliche Programmstelle untersucht werden.

Erweiterter Wertebereich [0, 1, _] ... _ 'DONT CARE'

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i :**L**[s_j]	**LOAD** s_j	**if** s_j == _ **then** z_i := _ **else** **if** s_j == 1 **then** z_i := 1 **else** z_i := 0 **fi** **fi**
c_i :**LN**[s_j]	**LOAD NOT** s_j	**if** s_j == _ **then** z_i := _ **else** **if** s_j == 0 **then** z_i := 1 **else** z_i := 0 **fi** **fi**

Tab. 11: Funktionalität (dont_care) 1

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i : **L** $[s_j]$	**LOAD** s_j	**if** s_j == _ **then** z_i := _ **else** **if** s_j == 1 **then** z_i := 1 **else** z_i := 0 **fi** **fi**
c_i : **LN** $[s_j]$	**LOAD NOT** s_j	**if** s_j == _ **then** z_i := _ **else** **if** s_j == 0 **then** z_i := 1 **else** z_i := 0 **fi** **fi**

c_i : **U** $[s_j]$	z_{i-1} **AND** s_j	**if** z_{i-1} == _ **then**
		if s_j == 0 **then** z_i := 0
		else
		z_i := _
		fi
		else
		if s_j == _ **then**
		if z_{i-1} == 0 **then** z_i := 0
		else
		z_i := _
		fi
		else
		if s_j == 0 **then** z_i := 0
		else
		if z_{i-1} == 1 **then** z_i := 1
		else
		z_i := 0
		fi
		fi
		fi
		fi

Tab. 12: Funktionalität (dont_care) 2

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung

c_i : **UN** $[s_j]$	z_{i-1} **AND NOT** s_j	**if** z_{i-1} == _ **then**
		  if s_j == 1 **then** z_i := 0
		  else
		  z_i := _
		  fi
		  else
		  if s_j == _ **then**
		  if z_{i-1} == 0 **then** z_i := 0
		  else
		  z_i := _
		  fi
		  else
		  if s_j == 1 **then** z_i := 0
		  else
		  if z_{i-1} == 1 **then** z_i := 1
		  else
		  z_i := 0
		  fi
		  fi
		  fi
		fi

c_i : **O** $[s_j]$	z_{i-1} **OR** s_j	**if** z_{i-1} == _ **then**
		  if s_j == 1 **then** z_i := 1
		  else
		  z_i := _
		  fi
		  else
		  if z_{i-1} == 1 **then** z_i := 1
		  else
		  if s_j == 1 **then** z_i := 1
		  else
		  z_i := 0
		  fi
		  fi
		fi

Tab. 13: Funktionalität (dont_care) 3

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung

c_i : **ON** $[s_j]$	z_{i-1} **OR NOT** s_j	**if** z_{i-1} == _ **then**
		  if s_j == 0 **then** z_i := 1
		  else
		  z_i := _
		  fi
		  else
		  if z_{i-1} == 1 **then** z_i := 1
		  else
		  if s_j == 0 **then** z_i := 1
		  else
		  z_i := 0
		  fi
		  fi
		fi

Tab. 14: Funktionalität (dont_care) 4

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung

c_i :)	)	pop
		if 'AND' then
		  pop
		  if [z_{i-1}]== _ then z_i:= _
		  else
		  if z_{i-1} == 0 then z_i:= 0
		  else
		  if [z_{i-1}] == 0 then z_i:= 0
		  else
		  z_i:= 1
		  fi
		  fi
		  fi
		  else
		  if 'OR' then
		  pop
		  if [z_{i-1}] == _ then
		  if z_{i-1} == 1 then z_i:= 1
		  fi
		  else
		  z_i:= _
		  fi
		  else
		  if [z_{i-1}] == 1 then z_i:= 1
		  else
		  if z_{i-1} == 1 then z_i:= 1
		  else
		  z_i:= 0
		  fi
		  fi
		  fi
		fi

Tab. 15: Funktionalität (dont_care) 5

Code	Verknüpfung	Anmerkung, Regeln für die Berechnung
c_i :=[s_j]	**EQUAL** s_j	**if** z_{i-1} == _ **then** s_j := _ **else** **if** z_{i-1} == 1 **then** s_j := 1 **else** s_j := 0 **fi** **fi**
c_i :**S**[s_j]	**SET** s_j	**if** z_{i-1} == _ **then** s_j := _ **else** **if** z_{i-1} == 1 **then** s_j := 1 **fi** **fi**
c_i :**R**[s_j]	**RESET** s_j	**if** z_{i-1} == _ **then** s_j := _ **else** **if** z_{i-1} == 1 **then** s_j := 0 **fi** **fi**

Tab. 16: Funktionalität (dont_care) 6

Die Variable „calculate“ (X) wird bei den anderen Operationen wie „don’t care“ behandelt. Ausnahme sind die Zuweisungen. In diesem Fall muss der Programmdurchlauf abgebrochen werden. Die Idee von „calculate“ ist, ausgehend von Annahmen zu möglichen Variablenzuständen zu ermitteln, welche Zustände nach der Programmbearbeitung erreicht werden, um diese danach durch den Vergleich mit der Aufgabenstellung zu verifizieren. Berechnet kann nur der Wert für eine Zuweisung werden.

Erweiterter Wertebereich [0, 1, _, X] ... X 'CALCULATE'

c_i :=$[s_j]$	**EQUAL** s_j	**if** z_{i-1} == X **then**
		  println “Error in program line”,[c_i],“Wert kann nicht zugewiesen werden”;
		  break
		 else
		  if z_{i-1} == 1 **then** s_j:= 1
		  else s_j:= 0
		  fi
		fi

c_i :**S**$[s_j]$	**SET** s_j	**if** z_{i-1} == X **then**
		  println “Error in program line”,[c_i],“Wert kann nicht zugewiesen werden”;
		  break
		 else
		  if z_{i-1} == 1 **then** s_j:= 1
		  fi
		fi

c_i :**R**$[s_j]$	**RESET** s_j	**if** z_{i-1} == X **then**
		  println “Error in program line”,[c_i],“Wert kann nicht zugewiesen werden”;
		  break
		 else
		  if z_{i-1} == 1 **then** s_j:= 0
		  fi
		fi

Tab. 17: Funktionalität (dont_care & Calculate)

Anhang B: SPS-Programme

In diesem Abschnitt sind die Quellprogramme einzelner Bausteine einer Fertigungsanlage für Kohlebürsten dargestellt, die weiter durch die Programmanalyse untersucht werden sollen (Anhang C). Verdeutlicht wird dabei auch, dass die gegebenen informellen Beschreibungen der Aufgabenstellung ergänzt werden müssen, um die gewünschte Funktionalität sicher zu stellen. Die Progammdarstellung erfolgt in Anweisungsliste im Sprachdialekt STEP 7 für eine SPS von Siemens der Baureihe 300. Generell wurde die Darstellungsart mit symbolischer Adressierung und ausführlicher Symbolkommentierung deshalb gewählt, weil dadurch auch die Funktionalität der einzelnen Funktionsblöcke (Netzwerke) besser nachvollzogen werden kann.

Allgemein gilt, dass symbolische Adressen gleichwertig mit der üblichen Adressierung bei der SPS-Programmerstellung genutzt werden kann. Für die Analyse sind diese aber weniger gut geeignet, da jedenfalls eine Zuordnung z.B. in der Form von Listen vorgenommen werden muss. Neben der sybolischen Adressierung sind den verwendeten Symoblen auch Symbolinformationen zuordenbar, mit denen die Symbole genauer beschrieben werden können.

Fertigungsautomat für Kohlebürsten

Der hier als Beispiel herangezogene Automat erzeugt Kohlebürsten für Elektromotoren. Die betrachtete Anlage wurde mit einer neuen Steuerung ausgerüstet und verfügt für die Prozessbeobachtung und Bedienung der Fertigungseinrichtung auch über ein Bildschirmterminal, dass im hier betrachteten Konzept überwiegend für die Überwachung des Prozesses eingesetzt wird. Aspekte dazu sind in [KG17] diskutiert.

Kohlebürsten sind Gleitkontakte in Elektromotoren, die zur Übertragung von elektrischer Energie auf bewegliche Kontakte dienen. Der Name Kohlebürste stammt von den früher zur Kontaktierung genutzten bürstenartigen Bündeln dünner Kupferdrähte, die später wegen besserer elektischer und mechanischer Eigenschaften durch Materialien wie Graphit und Kohlematerialien ersetzt worden sind.

Hauptbearbeitungsschritte der Fertigungsanlage sind das Bohren, Stampfen und Lacken, daher stammt die Bezeichung „BSL“. Eine ganze Reihe weitere zuschaltbarer Fertigungsschritte können notwendig werden, die in der Folge nicht detailliert untersucht werden weil sich die Vorgehensweise mit den als Beispiel herangezogenen SPS-Programmteilen deckt.

Das Programm ist bei dieser Anlage bereits in einzelne Programmbausteine gegliedert, die der Reihe nach aufgerufen und abgearbeitet werden. Der Schritt für die Aufteilung des Programmes in Programmsegmente kann daher hier entfallen. Die grundsätzliche Forderung, dass die Ausgänge der SPS exklusiv den Fertigungsmaschinen für die einzelen Bearbeitsungsschritte in der Fertigungskette zugeteilt sind, ist erfüllt.

Bausteine für Bohrstationen

Informell wurde der Ablauf für die Borstation bereits beschrieben. In Summe sind in der Fertigungskette mehrere Einzelmaschinen für die Bearbeitung notwendig, die entweder als Bohrmaschinen genutzt werden oder die im Funktionsablauf einen ähnlichen Ablauf besitzen. Ergänzend ist anzumerken, dass zusätzliche Forderungen implementiert worden sind, z.B. wann dieser Ablauf überhaupt gestartet werden darf und weitere Details, z.B. dass für den händischen Start eine Zustimmfunktion vorgesehen ist (Start des Ablaufes nur, wenn zeitgleich zwei Tasten ausgelöst werden) oder dass eine Schrittfunktion zu realisieren war, damit Haltepunkte im Bewegungsablauf das Justieren von Endlagenschaltern für das Einstellen des Bohrhubes erleichtern usw.

Das generelle Hauptproblem von Ablaufbeschreibungen ist ihre Unvollständigkeit, die oft erst im Zuge der Programmentwicklung entsprechend ergänzt wird. Andererseits lässt sich jedoch durch die Analyse die Systemreaktionen ableiten und den den Vorgaben in Einklang bringen.

Die Kurzbezeichnungen bei den folgenden Netzwerken wie z.B. „Bohren 1 Sicherheit“ umreißt zum besseren Verständnis die geplante Funktionalität der Kommandofolge. Die dort vorgesehene Zeitfunktion (Zeitdauer 200 ms) gibt z.B. jenen Zeitraum an, innerhalb dessen die Auslösung der Handfunktionen zugelassen ist.

```
SIMATIC BSL244_OP\S7-369_E\...\FC21 - <offline> 21.03.2017 12:23:04

Baustein: FC21 Bohrmaschine 1

Netzwerk: 1 Bohren 1 Sicherheit
U(
O "T_BOHREN1"           E4.3 -- Taster Bohren 1
O "ZW_SICH"             E126.7 -- Zweihand Sicherheit
ON "HANDBETRIEB"        M0.0 -- Maschine ist in Handbetrieb
)
L S5T#200MS
SE "B1_ZWEIH"           T11 -- Bohren 1 Zweihand
NOP 0
NOP 0
NOP 0
NOP 0
```

```
Netzwerk: 2 Bohren 1 Schritt 1
U(
U "HANDBETRIEB"         M0.0 -- Maschine ist in Handbetrieb
U "Z11_OB"              E3.7 -- Z11 Bohrzyl. 1 oben
U "T_BOHREN1"           E4.3 -- Taster Bohren 1
U "ZW_SICH"             E126.7 -- Zweihand Sicherheit
UN "B1_ZWEIH"           T11 -- Bohren 1 Zweihand
FP "B1_IMP_H"           M20.6 -- Bohren 1 Impuls
O
U "AUTO_START"          M0.1 -- Automatik Start auf alle Stationen
UN "B1_FERTIG"          M10.1 -- Bohren 1 Fertigmeldung
O "B1_SCHR_1"           M20.1 -- Bohren 1 Schritt 1
)
U "VW_B1"               E3.6 -- VW Bohren 1
U "B1_MOTOR"            E4.2 -- VW Motor Bohren 1 + 2 Ein-Aus
U "STARTBEDINGUNG"      M0.2 -- Rundtisch ist in Position, Motore sind
eingeschaltet
UN "B1_RESET"           M20.0 -- Bohren 1 Schrittkette Reset
= "B1_SCHR_1"           M20.1 -- Bohren 1 Schritt 1

Netzwerk: 3 Bohren 1 Verzögerung unten (freischneiden)
U "B1_SCHR_1"           M20.1 -- Bohren 1 Schritt 1
U "Z11_UN"              E4.0 -- Z11 Bohrzyl. 1 unten
L S5T#200MS
SE "B1_UNTEN"           T12 -- Bohren 1 Verzögerung unten (freischneiden)
NOP 0
NOP 0
NOP 0
NOP 0

Netzwerk: 4 Bohren 1 Schritt 2
U(
U "B1_SCHR_1"           M20.1 -- Bohren 1 Schritt 1
U "Z11_UN"              E4.0 -- Z11 Bohrzyl. 1 unten
U "B1_UNTEN"            T12 -- Bohren 1 Verzögerung unten (freischneiden)
U "B1_WEITER"           M20.4 -- Bohren 1 Weiterschaltung
O "B1_SCHR_2"           M20.2 -- Bohren 1 Schritt 2
)
UN "B1_RESET"           M20.0 -- Bohren 1 Schrittkette Reset
= "B1_SCHR_2"           M20.2 -- Bohren 1 Schritt 2

Netzwerk: 5 Bohren 1 Schrittkette Reset
U "B1_SCHR_2"           M20.2 -- Bohren 1 Schritt 2
U "Z11_OB"              E3.7 -- Z11 Bohrzyl. 1 oben
ON "VW_B1"              E3.6 -- VW Bohren 1
ON "B1_MOTOR"           E4.2 -- VW Motor Bohren 1 + 2 Ein-Aus
O
U "B1_KOHLE_F"          T52 -- Bohren 1 Kohle fehlt
U "B1_FEHLER"           M20.3 -- Bohren 1 FEHLER
O "RESET_STAT"          M0.6 -- Reset Stationen (Ablauf unterbrechen und
zurücksetzen)
= "B1_RESET"            M20.0 -- Bohren 1 Schrittkette Reset

Netzwerk: 6 Bohren 1 Fertigmeldung
U "B1_SCHR_2"           M20.2 -- Bohren 1 Schritt 2
FP "IMP_M100.0"         M100.0 -- Impulsmerker
S "B1_FERTIG"           M10.1 -- Bohren 1 Fertigmeldung
U "RES_FERTIG"          M0.3 -- Reset aller Fertigmeldungen
```

```
R "B1_FERTIG"          M10.1 -- Bohren 1 Fertigmeldung
NOP 0
```

Netzwerk: 7 Bohren 1 Blasen

```
U "B1_SCHR_1"          M20.1 -- Bohren 1 Schritt 1
UN "Z11_OB"            E3.7 -- Z11 Bohrzyl. 1 oben
UN "B1_SCHR_2"         M20.2 -- Bohren 1 Schritt 2
L S5T#500MS
SA "B1_BLASEN"         T13 -- Bohren 1 Blasen
NOP 0
NOP 0
NOP 0
NOP 0
```

Netzwerk: 8 Bohren 1 Überwachung Kohle fehlt

```
U "B1_SCHR_1"          M20.1 -- Bohren 1 Schritt 1
L S5T#300MS
SE "B1_KOHLE_F"        T52 -- Bohren 1 Kohle fehlt
NOP 0
NOP 0
NOP 0
NOP 0
```

Netzwerk: 9 Bohren 1 FEHLER

```
U(
U "B1_SCHR_1"          M20.1 -- Bohren 1 Schritt 1
U "B1_KOHLE"           E4.1 -- Keine KB bei Bohren 1
O "B1_FEHLER"          M20.3 -- Bohren 1 FEHLER
)
U(
ON "T_BOHREN1"         E4.3 -- Taster Bohren 1
O "ZW_SICH"            E126.7 -- Zweihand Sicherheit
O "B1_ZWEIH"           T11 -- Bohren 1 Zweihand
)
U "STARTBEDINGUNG"     M0.2 -- Rundtisch ist in Position, Motore sind
                            eingeschaltet
= "B1_FEHLER"          M20.3 -- Bohren 1 FEHLER
```

Netzwerk: 10 Bohren Motor

```
U "VW_B1"              E3.6 -- VW Bohren 1
U "B1_MOTOR"           E4.2 -- VW Motor Bohren 1 + 2 Ein-Aus
U "MOTOR_EIN"          A125.1 -- Motore Ein
= "MOT_B1"             A0.7 -- Motor - Bohren 1
```

Netzwerk: 11 Bohren Vorschub

```
U "B1_SCHR_1"          M20.1 -- Bohren 1 Schritt 1
U "B1_KOHLE_F"         T52 -- Bohren 1 Kohle fehlt
UN "B1_FEHLER"         M20.3 -- Bohren 1 FEHLER
UN "B1_SCHR_2"         M20.2 -- Bohren 1 Schritt 2
U "MOT_B1"             A0.7 -- Motor - Bohren 1
= "MV_Z11"             A32.4 -- Bohrzylinder 1
```

Netzwerk: 12 Bohren Blasventil

```
U "B1_BLASEN"          T13 -- Bohren 1 Blasen
U "MOT_B1"             A0.7 -- Motor - Bohren 1
= "MV__1"              A31.0 -- Abblasen Bohren 1
```

```
Netzwerk: 13 Bohren 1 Weiterschaltung
U "VW_SCHRITT"          E23.5 -- Schrittkettensteuerung
U "HANDBETRIEB"         M0.0 -- Maschine ist in Handbetrieb
U "T_BOHREN1"           E4.3 -- Taster Bohren 1
U "ZW_SICH"             E126.7 -- Zweihand Sicherheit
UN "B1_ZWEIH"           T11 -- Bohren 1 Zweihand
FP "B1_IMP"             M20.5 -- Bohren 1 Impuls
ON "VW_SCHRITT"         E23.5 -- Schrittkettensteuerung
= "B1_WEITER"           M20.4 -- Bohren 1 Weiterschaltung
```

Baustein für Wenden

Wenden ist eine Station und gleichzeitig der Arbeitschritt, bei dem eine bereits zum Teil bearbeitete Kohlebürste in eine Lage gebracht wird, um sie an einer anderen Stelle auf derselben Kohlebürste bearbeiten zu können. Dazu wird die Kohlebürste aus der Maschine entnommen um sie, um eine oder mehrere Achsen gedreht, wieder in die Haltevorrichtung einzusetzen. Das hier genutzte SPS-Programm kennt zwei unterschiedliche Abläufe, die bei der Einstellung des Fertigungsauftrags durch Vorwahlschalter bestimmt werden.

Von Interesse ist der in der Folge vorgestellte Programmbaustein auch deshalb, weil Überlegungen angestellt werden können, ob die Komplexität des SPS-Programms durch Aufsplitten in (hier) zwei Teilprogramme verringert werden könnte.

Informelle Beschreibung des Ablaufes „Wenden“

Wenden Programm 1	
Startbedingung:	Drehtisch in Endlage
Grundstellung:	Kohleklemmzylinder Endlage hinten, (Ventil aus)
	Ausschubzylinder Endlage hinten, (Ventil aus)
	Annäherungszylinder Endlage hinten, (Ventil aus)
	Einschubzylinder Endlage hinten, (Ventil aus)
	Querzylinder Endlage links, (Ventil aus)
	Wendezylinder Endlage links, (Ventil aus)
	Niederhaltezylinder Endlage hinten, (Ventil aus)
Ablauf:	
Schritt 1:	Kohleklemmung öffnen (Ventil Kohleklemmung ein)
	Endlage vorne erreicht:
Schritt 2:	Ausschubzylinder vor (Ventil Ausschubzylinder ein)
	Endlage vorne erreicht: Wartezeit 0,2 sec
Schritt 3:	Ausschubzylinder rück (Ventil Ausschubzylinder aus)
	Endlage hinten erreicht:
Schritt 4:	Hubzylinder vor (Ventil Hubzylinder ein)
	Querzylinder vor (Ventil Querzylinder ein)
	Wendezylinder rechts (Ventil Wendezylinder ein)
	Endlage Wendezyl. vorne erreicht:

Schritt 5:	Annäherungszylinder vor (Ventil Annäherungszylinder ein)
	Endlage vorne erreicht:
Schritt 6:	Kohleklemmung schließen (Ventil Kohleklemmung aus)
	Einschubzylinder vor (Ventil Einschubzylinder ein)
	Endlage vorne erreicht:
Schritt 7:	Niederhaltezylinder vor (Ventil Niederhaltezylinder ein)
	Endlage vorne erreicht:
Schritt 8:	Niederhaltezylinder rück (Ventil Niederhaltezylinder aus)
	Einschubzylinder rück (Ventil Einschubzylinder aus)
	Annäherungszylinder rück (Ventil Annäherungszylinder aus)
	Endlage Annäherungszyl. hinten erreicht:
Schritt 9:	Hubzylinder rück (Ventil Hubzylinder aus)
	Querzylinder links (Ventil Querzylinder aus)
	Wendezylinder links (Ventil Wendezylinder aus)
	Endlage Querzyl. Und Endlage Wendezyl. hinten erreicht:
Fertig:	Grundstellung wieder erreicht
Drehbedingung:	
	Kohleklemmzylinder Endlage hinten,
	Ausschubzylinder Endlage hinten,
	Annäherungszylinder Endlage hinten
	Niederhaltezylinder Endlage hinten

Wenden Programm 2

Startbedingung:	Drehtisch in Endlage
Grundstellung:	Kohleklemmzylinder Endlage hinten, (Ventil aus)
	Ausschubzylinder Endlage hinten, (Ventil aus)
	Annäherungszylinder Endlage hinten, (Ventil aus)
	Einschubzylinder Endlage hinten, (Ventil aus)
	Querzylinder Endlage links, (Ventil aus)
	Wendezylinder Endlage links, (Ventil aus)
	Niederhaltezylinder Endlage hinten, (Ventil aus)
	WB Klemmzylinder Endlage hinten, (Ventil aus)

Ablauf:

Schritt 1:	Annäherungszylinder vor (Ventil Annäherungszylinder ein)
	Endlage vorne erreicht:
Schritt 2:	Saugen EIN (Ventil Saugen ein)
Schritt 3:	Kohleklemmung öffnen (Ventil Kohleklemmung ein)
	Endlage vorne erreicht:
Schritt 4:	Annäherungszylinder rück (Ventil Annäherungszylinder aus)
	Endlage hinten erreicht:
Schritt 5:	Hubzylinder vor (Ventil Hubzylinder ein)
	Querzylinder vor (Ventil Querzylinder ein)
	Wendezylinder vor (Ventil Wendezylinder ein)
	WB Klemmung öffnen (Ventil WB Klemmung ein)
	Endlage Querzyl. und Endlage Wendezyl. vorne erreicht:
	Endlage WB Klemmung vorne erreicht:
Schritt 6:	Annäherungszylinder vor (Ventil Annäherungszylinder ein)

	Kohleklemmung schließen (Ventil Kohleklemmung aus)
	Einschubzylinder vor (Ventil Einschubzylinder ein)
	Beide Endlagen vorne erreicht:
Schritt 7:	Niederhaltezylinder vor (Ventil Niederhaltezylinder ein)
	Endlage vorne erreicht:
Schritt 8:	Saugen AUS (Ventil Saugen aus)
	Blasen EIN (Ventil Blasen ein) für 0,2 sek
Schritt 9:	WB Klemmung schließen (Ventil WB Klemmung aus)
Schritt 10:	Niederhaltezylinder rück (Ventil Niederhaltezylinder au s)
	Einschubzylinder rück (Ventil Einschubzylinder aus)
	Annäherungszylinder rück (Ventil Annäherungszylinder au s)
	Endlage Annäherungszyl. hinten erreicht:
Schritt 11:	Hubzylinder rück (Ventil Hubzylinder aus)
	Querzylinder rück (Ventil Querzylinder aus)
	Wendezylinder rück (Ventil Wendezylinder aus)
	Endlage Querzyl. Und Endlage Wendezyl. hinten erreicht:
Fertig:	Grundstellung wieder erreicht
Drehbedingung:	
	Kohleklemmzylinder Endlage hinten,
	(Ausschubzylinder Endlage hinten)
	Annäherungszylinder Endlage hinten
	Niederhaltezylinder Endlage hinten
	WB Klemmzylinder Endlage hinten

Ergänzngen der Aufgabenstellung

Auch die informelle Beschreibung des Ablaufes „Wenden“ muss durch Erklärungen ergänzt werden. Obwohl der Ablauf bereits gut erkennbar ist, fehlt zu allererst eine Zuordnung der einzelnen Bezeichnungen zu konkreten Adressen in der SPS. Betrachtet man die Bezeichnungen wie z.B. „Einschubzylinder“, muss dieser genauer bezeichnet werden, weil es mehere Funktionen an der Maschine in unterschiedlichen Stationen gibt, wo etwas „eingeschoben“ wird und daher auch die gleiche Bezeichnung in einer informell beschriebenen Aufgabenstellung mehrfach genutzt wird. Auch muss etwa klargestellt sein, ob z.B. der Niederhaltezylinder vom Progammablauf „Wenden 1“ der gleiche Zylinder ist, der beim Programmablauf „Wenden 2“ angesprochen wird.

Die Aussage „Endlage erreicht“ ist so zu verstehen, dass wenn in einem Bewegungsablauf eine bestimmte Position erreicht wird z.B. ein nachfolgender Schritt ausgelöst werden soll. Ist es wie im gegebenen Ablauf der Fall, dass Bewegungsabläufe durch mechanische Anschläge begrentzt sind, genügt es fallweise beim Ansprechen der Enlagenüberwachung den Folgeschritt sofort zu starten. Muss aber sichergestellt werden, dass der mechanische Anschlag die Bewegung unterbrochen hat, wird z.B. eine kurze Verzögung des nächstfolgenden Schrittes

notwendig werden. Eine weitere Frage ist, ob es notwendig ist, auch bereits vorher erreichte Positionen als Weiterschaltbedingung in die Überwachungskette mit einzubeziehen.

```
SIMATIC BSL244_OP\S7-369_E\...\FC34 - <offline> 22.04.2017 09:08:50

Netzwerk: 1      Wenden Sicherheit
U(
O     "T_WENDEN"        E11.4 -- Taster Wenden
O     "ZW_SICH"         E126.7 -- Zweihand Sicherheit
ON    "HANDBETRIEB"     M0.0 -- Maschine ist in Handbetrieb
)
L     S5T#200MS
SE    "WENDEN_ZWEIH"    T53 -- Wenden Zweihand
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 2      Wenden 1 Grundstellung
U     "Z28_HI"          E11.6 -- Z28 Kohleklemmzyl. hinten
U     "Z29_HI"          E12.0 -- Z29 Ausschubzyl hinten
U     "Z30_HI"          E12.2 -- Z30 Annäherungzyl hinten
U     "Z34_HI"          E13.0 -- Z34 Einschubzyl hinten
U     "Z33_LINKS"       E12.5 -- Z33 Wendezyl links
U     "Z35_OB"          E13.1 -- Z35 Niederhaltezyl oben W-Prog. 1)
=     "W1_GST"          M50.0 -- Wenden 1 Grundstellung


Netzwerk: 3      Wenden 1 Schritt 1
U(
U     "HANDBETRIEB"     M0.0 -- Maschine ist in Handbetrieb
U     "W1_GST"          M50.0 -- Wenden 1 Grundstellung
U     "T_WENDEN"        E11.4 -- Taster Wenden
U     "ZW_SICH"         E126.7 -- Zweihand Sicherheit
UN    "WENDEN_ZWEIH"    T53 -- Wenden Zweihand
O
U     "AUTO_START"      M0.1 -- Automatik Start auf alle Stationen
UN    "W1_FERTIG"       M11.7 -- Wenden 1 Fertigmeldung
O     "W1_SCHR_1"       M51.1 -- Wenden 1 Schritt 1
)
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                                eingeschaltet
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_1"       M51.1 -- Wenden 1 Schritt 1

Netzwerk: 4      Wenden 1 Schritt 2
U(
U     "W1_SCHR_1"       M51.1 -- Wenden 1 Schritt 1
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z28_VO"          E11.5 -- Z28 Kohleklemmzyl. vorne
O     "W1_SCHR_2"       M51.2 -- Wenden 1 Schritt 2
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_2"       M51.2 -- Wenden 1 Schritt 2
```

```
Netzwerk: 5      Wenden 1 Wartezeit vorne
U     "W1_SCHR_2"       M51.2 -- Wenden 1 Schritt 2
U     "Z29_VO"          E11.7 -- Z29 Ausschubzyl vorne
L     S5T#200MS
SE    "W1_WARTE"        T66 -- Wenden 1 Wartezeit vorne
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 6      Wenden 1 Schritt 3
U(
U     "W1_SCHR_2"       M51.2 -- Wenden 1 Schritt 2
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z29_VO"          E11.7 -- Z29 Ausschubzyl vorne
U     "W1_WARTE"        T66 -- Wenden 1 Wartezeit vorne
O     "W1_SCHR_3"       M51.3 -- Wenden 1 Schritt 3
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_3"       M51.3 -- Wenden 1 Schritt 3

Netzwerk: 7      Wenden 1 Schritt 4
U(
U     "W1_SCHR_3"       M51.3 -- Wenden 1 Schritt 3
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z29_HI"          E12.0 -- Z29 Ausschubzyl hinten
O     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4

Netzwerk: 8      Wenden 1 Schritt 5
U(
U     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z33_RECHTS"      E12.6 -- Z33 Wendezyl rechts
O     "W1_SCHR_5"       M51. - Wenden 1 Schritt 5
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_5"       M51.5 -- Wenden 1 Schritt 5

Netzwerk: 9      Wenden 1 Schritt 6
U(
U     "W1_SCHR_5"       M51.5 -- Wenden 1 Schritt 5
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z30_VO"          E12.1 -- Z30 Annäherungzyl vorne
O     "W1_SCHR_6"       M51.6 -- Wenden 1 Schritt 6
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_6"       M51.6 -- Wenden 1 Schritt 6

Netzwerk: 10     Wenden 1 Schritt 7
U(
U     "W1_SCHR_6"       M51.6 -- Wenden 1 Schritt 6
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z28_HI"          E11.6 -- Z28 Kohleklemmzyl. hinten
U     "Z34_VO"          E12.7 -- Z34 Einschubzyl vorne
O     "W1_SCHR_7"       M51.7 -- Wenden 1 Schritt 7
```

```
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Rese
t
=     "W1_SCHR_7"       M51.7 -- Wenden 1 Schritt 7

Netzwerk: 11     Wenden 1 Schritt 8
U(
U     "W1_SCHR_7"       M51.7 -- Wenden 1 Schritt 7
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z35_UN"          E13.2 -- Z35 Niederhaltezyl unten (W-Prog. 1)
O     "W1_SCHR_8"       M52.0 -- Wenden 1 Schritt 8
)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_8"       M52.0 -- Wenden 1 Schritt 8

Netzwerk: 12     Wenden 1 Schritt 9
U(
U     "W1_SCHR_8"       M52.0 -- Wenden 1 Schritt 8
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z30_HI"          E12.2 -- Z30 Annäherungzyl hinten
U     "Z34_HI"          E13.0 -- Z34 Einschubzyl hinten
U     "Z35_OB"          E13.1 -- Z35 Niederhaltezyl oben (W-Prog. 1)
O     "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9)
UN    "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset
=     "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9

Netzwerk: 13     Wenden 1 Schrittkette Reset
U     "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9
U     "W1_GST"          M50.0 -- Wenden 1 Grundstellung
ON    "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O     "RESET_STAT"      M0.6 -- Reset Stationen (Ablauf unterbrechen und
                              zurücksetzen)
=     "W1_RESET"        M51.0 -- Wenden 1 Schrittkette Reset

Netzwerk: 14     Wenden 1 Fertigmeldung
U     "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9
FP    "IMP_M102.1"      M102.1 -- Impulsmerker
S     "W1_FERTIG"       M11.7 -- Wenden 1 Fertigmeldung
U     "RES_FERTIG"      M0.3 -- Reset aller Fertigmeldungen
R     "W1_FERTIG"       M11.7 -- Wenden 1 Fertigmeldung
NOP   0

Netzwerk: 15     Wenden 2 Grundstellung
U     "Z28_HI"          E11.6 -- Z28 Kohleklemmzyl. hinten
U     "Z29_HI"          E12.0 -- Z29 Ausschubzyl hinten
U     "Z30_HI"          E12.2 -- Z30 Annäherungzyl hinten
U     "Z34_HI"          E13.0 -- Z34 Einschubzyl hinten
U     "Z32_LINKS"       E12.3 -- Z32 Querzyl links
U     "Z33_LINKS"       E12.5 -- Z33 Wendezyl links
U     "Z35_OB"          E13.1 -- Z35 Niederhaltezyl oben (W-Prog. 1)
U     "Z37_ZU"          E13.6 -- Z37 Wendebacke Klemmzyl ZU (W-Prog. 2)
=     "W2_GST"          M50.1 -- Wenden 2 Grundstellung

Netzwerk: 16     Wenden 2 Schritt 1
U(
U     "HANDBETRIEB"     M0.0 -- Maschine ist in Handbetrieb
U     "W2_GST"          M50.1 -- Wenden 2 Grundstellung
U     "T_WENDEN"        E11.4 -- Taster Wenden
```

```
U     "ZW_SICH"          E126.7 -- Zweihand Sicherheit
UN    "WENDEN_ZWEIH"     T53 -- Wenden Zweihand
O
U     "AUTO_START"       M0.1 -- Automatik Start auf alle Stationen
UN    "W2_FERTIG"        M12.0 -- Wenden 2 Fertigmeldung
O     "W2_SCHR_1"        M53.1 -- Wenden 2 Schritt 1
)
U     "VW_WENDEN2"       E11.3 -- VW Wenden Programm 2
U     "STARTBEDINGUNG"   M0.2 -- Rundtisch ist in Position, Motore sind
                               eingeschaltet
UN    "W2_RESET"         M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_1"        M53.1 -- Wenden 2 Schritt 1
```

Netzwerk: 17 Wenden 2 Schritt 2

```
U(
U     "W2_SCHR_1"        M53.1 -- Wenden 2 Schritt 1
U     "W_WEITER"         M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z30_VO"           E12.1 -- Z30 Annäherungzyl vorne
O     "W2_SCHR_2"        M53.2 -- Wenden 2 Schritt 2
)
UN    "W2_RESET"         M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_2"        M53.2 -- Wenden 2 Schritt 2
```

Netzwerk: 18 Wenden Saugen

```
U     "W2_SCHR_2"        M53.2 -- Wenden 2 Schritt 2
L     S5T#400MS
SE    "WENDEN_SAUG"      T54 -- Wenden Saugen
NOP   0
NOP   0
NOP   0
NOP   0
```

Netzwerk: 19 Wenden 2 Schritt 3

```
U(
U     "W2_SCHR_1"        M53.1 -- Wenden 2 Schritt 1
U     "WENDEN_SAUG"      T54 -- Wenden Saugen
O     "W2_SCHR_3"        M53.3 -- Wenden 2 Schritt 3
)
UN    "W2_RESET"         M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_3"        M53.3 -- Wenden 2 Schritt 3
```

Netzwerk: 20 Wenden 2 Schritt 4

```
U(
U     "W2_SCHR_2"        M53.2 -- Wenden 2 Schritt 2
U     "W_WEITER"         M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z28_VO"           E11.5 -- Z28 Kohleklemmzyl. vorne
O     "W2_SCHR_4"        M53.4 -- Wenden 2 Schritt 4
)
UN    "W2_RESET"         M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_4"        M53.4 -- Wenden 2 Schritt 4
```

Netzwerk: 21 Wenden 2 Schritt 5

```
U(
U     "W2_SCHR_4"        M53.4 -- Wenden 2 Schritt 4
U     "W_WEITER"         M50.3 -- Wenden Weiterschaltung im
Einrichtbetrieb
U     "Z30_HI"           E12.2 -- Z30 Annäherungzyl hinten
O     "W2_SCHR_5"        M53.5 -- Wenden 2 Schritt 5
```

```
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5

Netzwerk: 22     Wenden 2 Schritt 6
U(
U     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z32_RECHTS"      E12.4 -- Z32 Querzyl rechts
U     "Z33_RECHTS"      E12.6 -- Z33 Wendezyl rechts
U     "Z37_AUF"         E13.5 -- Z37 Wendebacke Klemmzyl AUF (W-Prog. 2)
O     "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6

Netzwerk: 23     Wenden 2 Schritt 7
U(
U     "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z28_HI"          E11.6 -- Z28 Kohleklemmzyl. hinten
U     "Z30_VO"          E12.1 -- Z30 Annäherungzyl vorne
O     "W2_SCHR_7"       M53.7 -- Wenden 2 Schritt 7
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_7"       M53.7 -- Wenden 2 Schritt 7

Netzwerk: 24     Wenden 2 Schritt 8
U(
U     "W2_SCHR_7"       M53.7 -- Wenden 2 Schritt 7
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z36_VO"          E13.3 -- Z36 Niederhaltezyl vorne (W-Prog. 2)
O     "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8

Netzwerk: 25     Wenden Blasen
U     "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8
L     S5T#200MS
SE    "WENDEN_BLAS"     T55 -- Wenden Blasen
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 26     Wenden 2 Schritt 9
U(
U     "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8
U     "WENDEN_BLAS"     T55 -- Wenden Blasen
O     "W2_SCHR_9"       M54.1 -- Wenden 2 Schritt 9
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_9"       M54.1 -- Wenden 2 Schritt 9

Netzwerk: 27     Wenden 2 Schritt 10
U(
U     "W2_SCHR_9"       M54.1 -- Wenden 2 Schritt 9
```

```
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z37_ZU"          E13.6 -- Z37 Wendebacke Klemmzyl ZU (W-Prog. 2)
O     "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10

Netzwerk: 28     Wenden 2 Schritt 11
U(
U     "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10
U     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
U     "Z30_HI"          E12.2 -- Z30 Annäherungzyl hinten
O     "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
)
UN    "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset
=     "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11

Netzwerk: 29     Wenden 2 Reset
U     "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
U     "W2_GST"          M50.1 -- Wenden 2 Grundstellung
ON    "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
O     "RESET_STAT"      M0.6  -- Reset Stationen (Ablauf unterbrechen und
                              zurücksetzen)
=     "W2_RESET"        M53.0 -- Wenden 2 Schrittkette Reset

Netzwerk: 30
U     "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
FP    "IMP_M102.2"      M102.2 -- Impulsmerker
S     "W2_FERTIG"       M12.0 -- Wenden 2 Fertigmeldung
U     "RES_FERTIG"      M0.3 -- Reset aller Fertigmeldungen
R     "W2_FERTIG"       M12.0 -- Wenden 2 Fertigmeldung
NOP   0

Netzwerk: 31     Kohleklemmzylinder (Progr. 1)
U(
U     "W1_SCHR_1"       M51.1 -- Wenden 1 Schritt 1
UN    "W1_SCHR_6"       M51.6 -- Wenden 1 Schritt 6
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U     "W2_SCHR_3"       M53.3 -- Wenden 2 Schritt 3
UN    "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV_Z28"          A30.4 -- Kohleklemmzylinder (Progr. 1)

Netzwerk: 32     Ausschubzylinder (Progr. 1)
U     "W1_SCHR_2"       M51.2 -- Wenden 1 Schritt 2
UN    "W1_SCHR_3"       M51.3 -- Wenden 1 Schritt 3
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV_Z29"          A36.0 -- Ausschubzylinder (Progr. 1)

Netzwerk: 33     Annäherungszylinder (Progr. 1 + 2)
U(
U     "W1_SCHR_5"       M51.5 -- Wenden 1 Schritt 5
```

```
UN    "W1_SCHR_8"       M52.0 -- Wenden 1 Schritt 8
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U(
U     "W2_SCHR_1"       M53.1 -- Wenden 2 Schritt 1
UN    "W2_SCHR_4"       M53.4 -- Wenden 2 Schritt 4
O
U     "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6
UN    "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10
)
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
=     "MV_Z30"          A36.2 -- Annäherungszylinder (Progr. 1 + 2)

Netzwerk: 34     Hubzylinder (Progr. 1 + 2)
U(
U     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4
UN    "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5
UN    "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
=     "MV_Z31"          A36.4 -- Hubzylinder (Progr. 1+ 2)

Netzwerk: 35     Querzylinder (Progr. 1 + 2)
U(
U     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4
UN    "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5
UN    "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
=     "MV_Z32"          A36.6 -- Querzylinder (Progr. 1 + 2)

Netzwerk: 36     Wendezylinder (Progr. 1 + 2)
U(
U     "W1_SCHR_4"       M51.4 -- Wenden 1 Schritt 4
UN    "W1_SCHR_9"       M52.1 -- Wenden 1 Schritt 9
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5
UN    "W2_SCHR_11"      M54.3 -- Wenden 2 Schritt 11
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
=     "MV_Z33"          A37.0 -- Wendezylinder (Progr.1 + 2)
```

```
Netzwerk: 37     Einschubzylinder (Progr. 1 + 2)
U(
U     "W1_SCHR_6"       M51.6 -- Wenden 1 Schritt 6
UN    "W1_SCHR_8"       M52.0  -- Wenden 1 Schritt 8
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
O
U     "W2_SCHR_6"       M53.6 -- Wenden 2 Schritt 6
UN    "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
)
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV_Z34"          A37.2 -- Einschubzylinder (Progr. 1 + 2)

Netzwerk: 38     Wendebacke Niederhaltezylinder (Progr. 1)
U     "W1_SCHR_7"       M51.7 -- Wenden 1 Schritt 7
UN    "W1_SCHR_8"       M52.0 -- Wenden 1 Schritt 8
U     "VW_WENDEN1"      E11.2 -- VW Wenden Programm 1
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV_Z35"          A30.5 -- Wendebacke Niederhaltezylinder (Progr. 1)

Netzwerk: 39     Wendebacke öffnen (Progr. 2)
U     "W2_SCHR_5"       M53.5 -- Wenden 2 Schritt 5
UN    "W2_SCHR_9"       M54.1 -- Wenden 2 Schritt 9
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV Z37"          A37.6 -- Wendebacke öffnen (Progr. 2)

Netzwerk: 40     Niederhaltezylinder (Progr. 2)
U     "W2_SCHR_7"       M53.7 -- Wenden 2 Schritt 7
UN    "W2_SCHR_10"      M54.2 -- Wenden 2 Schritt 10
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "MV_Z36"          A37.4 -- Niederhaltezylinder (Progr. 2)

Netzwerk: 41     Wenden 2 Saugdüse
U     "W2_SCHR_2"       M53.2 -- Wenden 2 Schritt 2
UN    "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "W2_SAUGEN"       A2.4 -- Wenden 2 Saugdüse

Netzwerk: 42     Wenden 2 Blasdüse
U     "W2_SCHR_8"       M54.0 -- Wenden 2 Schritt 8
UN    "WENDEN_BLAS"     T55 -- Wenden Blasen
U     "VW_WENDEN2"      E11.3 -- VW Wenden Programm 2
U     "STARTBEDINGUNG"  M0.2 -- Rundtisch ist in Position, Motore sind
                             eingeschaltet
=     "W2_BLASEN"       A2.5 -- Wenden 2 Blasdüse


Netzwerk: 43     Wenden Weiterschaltung im Einrichtbetrieb
U     "VW_SCHRITT"      E23.5 -- Schrittkettensteuerung
U     "HANDBETRIEB"     M0.0 -- Maschine ist in Handbetrieb
```

```
U     "T_WENDEN"        E11.4 -- Taster Wenden
U     "ZW_SICH"         E126.7 -- Zweihand Sicherheit
UN    "WENDEN_ZWEIH"    T53 -- Wenden Zweihand
FP    "W_IMP"           M50.2  -- Wenden Impulsmerker
ON    "VW_SCHRITT"      E23.5 -- Schrittkettensteuerung
=     "W_WEITER"        M50.3 -- Wenden Weiterschaltung im Einrichtbetrieb
```

Baustein für Stampfen

Das Einstampfen des Kabels in die Kohlebürste ist der Kernprozess bei der betrachteten Fertigungseinrichtung.

Informelle Beschreibung des Ablaufes „Stampfen“

Stampfen
Startbedingung:
Drehtisch in Endlage
Grundstellung:
Stampfkopf Endlage oben
Schere Bewegung Endlage hinten
Schere Schnitt Endlage hinten
Ablauf:
Schritt 1: Stampfkopf senken (Ventil Stampfkopf ein)
Endlage unten erreicht: Start Wartezeit 0,2 sec (Knotenbildung)
Schritt 2: Spannzange öffnen (Ventil Spannzange ein)
Start Wartezeit 0,2 sec
Schritt 3: Stampfvorgang (Ventil Stampfen alternierend ein/aus, solange,
bis Niveaukontrolle erreicht)
Stampfvorgang Abbruch: Schlagzahl überschritten
Schritt 4: Kabelbremse auf (Ventil Kabelbremse ein)
Stampfkopf heben (Ventil Stampfkopf aus)
Endlage oben erreicht:
Schritt 5: Kabelbremse zu (Ventil Kabelbremse aus)
Spannzange zu (Ventil Spannzange aus)
Bei VW ohne Schere ist Grundstellung erreicht!
Schritt 6: Schere Bewegung vor (Ventil Schere Bewegung ein)
Endlage vorne erreicht:
Schritt 6a: Schere Schnitt vor (Ventil Schere Schnitt ein)
Endlage vorne erreicht:
Schritt 7: Schere Schnitt zurück (Ventil Schere Schnitt aus)
Endlage hinten erreicht:
Schritt 7a: Schere Bewegung zurück (Ventil Schere Bewegung aus)
Fertig: Grundstellung wieder erreicht
Drehbedingung:
Stampfkopf Endlage oben
Schritt 7 und Endlage Schere Schnitt vorne wieder verlassen

Ergänzungen der Aufgabenstellung

Ergänzungen zur informellen Beschreibung des Ablaufes sind wie zuvor erforderlich. Zusätzlich ist festzustellen, dass die Bewegung der Schere den Transport nicht beeintächtigen kann, wenn das Kabel bereits abgeschnitten worden ist. Da das Stampfen die längste Zeit im Prozessablauf erfordert, kann zur Beschleunigung die Transporteinrichtung bereits ihren Bewegungsablauf starten, während die Schere ihre Rückwärtsbewegung beendet.

```
SIMATIC BSL244_OP\S7-369_E\...\FC38 - <offline> 25.04.2017 12:18:23

Netzwerk: 1      Stampfen in Grundstellung
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
U     "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
U     "Z41_HI"          E15.2  -- Z41 Scherenzyl 2 (kurz) hinten
=     "ST2_GST"         M36.0  -- Stampfen 2 Grundstellung

Netzwerk: 2      Stampfen Start
U(
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_H_STAMPFEN"  M38.1  -- Stampfen 2 Hand Stampfen Start
O     "AUTO_START"      M0.1  -- Automatik Start auf alle Stationen
)
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
U     "VW_ST2"          E14.4  -- VW Stampfen 2
=     "ST2_START"       M36.1  -- Stampfen 2 gestartet

Netzwerk: 3      Stampfkopf senken
U     "ST2_START"       M36.1  -- Stampfen 2 gestartet
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
S     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken

Netzwerk: 4      Stampfen Start
U     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken
U     "Z39_UN"          E14.6  -- Z39 Kopfhubzyl unten
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
S     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start

Netzwerk: 5      Kabel abschneiden
U     "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
S     "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden

Netzwerk: 6      Kabel abschneiden
O     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
O     "RES_FERTIG"      M0.3  -- Reset aller Fertigmeldungen
```

```
ON    "VW_ST2"          E14.4  -- VW Stampfen 2
O     "RESET_STAT"      M0.6  -- Reset Stationen (Ablauf unterbrechen und
                              zurücksetzen)
R     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken
R     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
R     "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden

Netzwerk: 7      Stampfstation fertig für drehen
U     "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden
U     "Z40_VO"          E14.7  -- Z40 Scherenzyl 1 (lang) vorne
U     "Z41_VO"          E15.1  -- Z41 Scherenzyl 2 (kurz) vorne
O
U(
O     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken
O     "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
)
UN    "NOT_AUS"         E124.5  -- NOT - AUS
S     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen

Netzwerk: 8      Stampfstation fertig für drehen
U     "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
U     "Z41_HI"          E15.2  -- Z41 Scherenzyl 2 (kurz) hinten
ON    "VW_ST2"          E14.4  -- VW Stampfen 2
R     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen

Netzwerk: 9      Stampfkopf Heben
U(
UN    "SZ_F2"           E124.7  -- Schlagzahlfehler 2
FP    "IMP_M101.5"      M101.5  -- Impulsmerker
U     "Z39_UN"          E14.6  -- Z39 Kopfhubzyl unten
U     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
O
U     "ST2_ENDE"        E15.3  -- Stampfen Ende von Stampfcontroller
U     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
)
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
S     "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende

Netzwerk: 10     Stampfkopf Heben
O     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
O     "RES_FERTIG"      M0.3  -- Reset aller Fertigmeldungen
ON    "VW_ST2"          E14.4  -- VW Stampfen 2
R     "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende

Netzwerk: 11     Stampfen fertig
U(
U     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken
UN    "NOT_AUS"         E124.5  -- NOT - AUS
O(
U     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
FP    "IMP_M101.6"      M101.6  -- Impulsmerker
)
)
S     "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
U(
O     "RES_FERTIG"      M0.3  -- Reset aller Fertigmeldungen
```

```
ON    "VW_ST2"          E14.4  -- VW Stampfen 2
)
R     "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
NOP   0

Netzwerk: 12     Schere vor
U(
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_H_SCHNEIDEN" M38.0  -- Stampfen 2 Hand Kabel 1 abschneiden
O
U     "ST2_Schere_vor"  M36.7  -- Stampfen 2 Schere vor
UN    "Z41_VO"          E15.1  -- Z41 Scherenzyl 2 (kurz) vorne
)
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
UN    "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                                eingeschaltet
=     "ST2_Schere_vor"  M36.7  -- Stampfen 2 Schere vor

Netzwerk: 13     Schere zurück
U(
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_H_SCHERE_RÜCK" M38.2  -- Stampfen 2 Hand Schere zurück
O
U     "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück
UN    "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
)
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                                eingeschaltet
=     "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück

Netzwerk: 14     Kabelklemmung_AUF_ZU
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U(
O     "ST2_H_SCHNEIDEN" M38.0  -- Stampfen 2 Hand Kabel abschneiden
O     "T_ST2_ZU"        E16.2  -- Taster Spannzange zu
O     "T_ST2_SENKEN"    E16.0  -- Taster Stampfkopf senken
)
FP    "IMP_M101.7"      M101.7  -- Impulsmerker
=     "ST2_KKL_auf_zu"  M37.1  -- Stampfen 2 Kabelklemmung auf/zu

Netzwerk: 15     Kabelklemmung auf
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U(
O     "ST2_H_STAMPFEN"  M38.1  -- Stampfen 2 Hand Stampfen Start
O     "T_ST2_AUF"       E16.1  -- Taster Spannzange auf
)
O
U     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
U     "ST2_KNOTEN"      T48  -- Stampfen 2 Knotenzeit
UN    "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden
S     "ST2_KKL_auf"     M37.2  -- Stampfen 2 Kabelklemmung auf
                               (invertiertes Signal)

Netzwerk: 16     Kabelklemmung zu
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_KKL_auf_zu"  M37.1  -- Stampfen 2 Kabelklemmung auf/zu
O
```

```
U     "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden
UN    "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig fürdrehen
O     "ST2_Schere_vor"  M36.7  -- Stampfen 2 Schere vor
O     "ST2_START"       M36.1  -- Stampfen 2 gestartet
R     "ST2_KKL_auf"     M37.2  -- Stampfen 2 Kabelklemmung auf
                               (invertiertes Signal)

Netzwerk: 17     Stampfen Kabelklemmung
U     "ST2_KKL_auf"     M37.2  -- Stampfen 2 Kabelklemmung auf
                               (invertiertes Signal)
=     "MV_Z42"          A41.0  -- ST2 Spannzangenzylinder

Netzwerk: 18     Stampfkopf senken
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_H_SENKEN"    M37.7  -- Stampfen 2 Hand senken
U     "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                               eingeschaltet
=     "ST2_Kopf_senken" M37.3  -- Stampfen 2 Stampfkopf senken

Netzwerk: 19     Ventil senken
U(
U     "ST2_SENKEN"      M36.2  -- Stampfen 2 Stampfkopf senken
UN    "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
UN    "ST2_FERTIG"      M11.6  -- Stampfen 2 Fertigmeldung
O     "ST2_Kopf_senken" M37.3  -- Stampfen 2 Stampfkopf senken
)
U     "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
eingeschaltet
UN    "Z39_UN"          E14.6  -- Z39 Kopfhubzyl unten
U     "VW_ST2"          E14.4  -- VW Stampfen 2
S     "MV_Z39A"         A45.4  -- ST2 Kopfhubzylinder
S     "MV_Z39B"         A45.6  -- ST2 Kopfhubzylinder

Netzwerk: 20     Ventil Heben
U(
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "ST2_H_KOPF_HEBEN" M38.3  -- Stampfen 2 Hand Kopf heben
O
U     "ST2_H_SCHNEIDEN" M38.0  -- Stampfen 2 Hand Kabel abschneiden
UN    "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
O     "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
ON    "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                               eingeschaltet
)
UN    "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
R     "MV_Z39A"         A45.4  -- ST2 Kopfhubzylinder
R     "MV_Z39B"         A45.6  -- ST2 Kopfhubzylinder
=     "ST2_Kopf_heben"  M37.4  -- Stampfen 2 Stampfkopf heben

Netzwerk: 21     Schere Bewegung
U(
U     "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden
UN    "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig fürdrehen
O     "ST2_Schere_vor"  M36.7  -- Stampfen 2 Schere vor
)
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
```

```
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
UN    "Z40_VO"          E14.7  -- Z40 Scherenzyl 1 (lang) vorne
U     "VW_ST2"          E14.4  -- VW Stampfen 2
S     "MV_Z40"          A40.4  -- ST2 Scherenzylinder lang (Bewegung)

Netzwerk: 22     Schere Bewegung zurück
U(
U     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
U     "Z41_HI"          E15.2  -- Z41 Scherenzyl 2 (kurz) hinten
ON    "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
ON    "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
O
U     "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück
U     "Z41_HI"          E15.2  -- Z41 Scherenzyl 2 (kurz) hinten
)
UN    "Z40_HI"          E15.0  -- Z40 Scherenzyl 1 (lang) hinten
R     "MV_Z40"          A40.4  -- ST2 Scherenzylinder lang (Bewegung)

Netzwerk: 23     Schere Schnitt vor
U     "Z40_VO"          E14.7  -- Z40 Scherenzyl 1 (lang) vorne
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
UN    "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
UN    "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück
UN    "Z41_VO"          E15.1  -- Z41 Scherenzyl 2 (kurz) vorne
U     "VW_ST2"          E14.4  -- VW Stampfen 2
S     "MV_Z41"          A40.6  -- ST2 Scherenzylinder kurz (Schnitt)

Netzwerk: 24     Schere Schnitt zurück
U(
U(
O     "ST2_DREHEN"      M36.5  -- Stampfen 2 fertig für drehen
O     "ST2_Schere_rück" M37.0  -- Stampfen 2 Schere zurück
)
U     "Z40_VO"          E14.7  -- Z40 Scherenzyl 1 (lang) vorne
ON    "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
)
UN    "Z41_HI"          E15.2  -- Z41 Scherenzyl 2 (kurz) hinten
R     "MV_Z41"          A40.6  -- ST2 Scherenzylinder kurz (Schnitt)

Netzwerk: 25     Kabelbremse
U     "ST2_Kopf_heben"  M37.4  -- Stampfen 2 Stampfkopfheben
UN    "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
U     "STARTBEDINGUNG"  M0.2  -- Rundtisch ist in Position, Motore sind
                              eingeschaltet
R     "MV_Z43"          A44.2  -- ST2 Kabelbremse

Netzwerk: 26     Ventil Kabelbremse
U     "Z39_OB"          E14.5  -- Z39 Kopfhubzyl oben
S     "MV_Z43"          A44.2  -- ST2 Kabelbremse

Netzwerk: 27     Knotenzeit
U     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
UN    "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
L     S5T#250MS
SE    "ST2_KNOTEN"      T48  -- Stampfen 2 Knotenzeit
NOP   0
```

```
NOP   0
NOP   0
NOP   0

Netzwerk: 28     Kabelklemmung offen, Stampfen Start
U     "ST2_KNOTEN"      T48  -- Stampfen 2 Knotenzeit
UN    "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
L     S5T#250MS
SE    "ST2_ST_START"    T49  -- Stampfen 2 Startfreigabe
NOP   0
NOP   0
NOP   0
U     "ST2_ST_START"    T49  -- Stampfen 2 Startfreigabe
=     "ST2_STAMPEN_START" M37.6  -- Stampfen 2 Stampfcontroller Start

Netzwerk: 29     Zweihandsicherheit
U(
O     "T_ST2_SENKEN"    E16.0  -- Taster Stampfkopf senken
O     "T_ST2_HEBEN"     E15.7  -- Taster Stampfkopf heben
O     "T_ST2_VOR"       E16.3  -- Taster Schere vor
O     "T_ST2_ZURÜCK"    E16.4  -- Taster Schere zurück
O     "T_ST2_STAMPFEN"  E16.5  -- Taster Stampfen
O     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
ON    "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
)
L     S5T#200MS
SE    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 30     Hand Senken
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "T_ST2_SENKEN"    E16.0  -- Taster Stampfkopf senken
U     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
UN    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
=     "ST2_H_SENKEN"    M37.7  -- Stampfen 2 Hand senken

Netzwerk: 31     Hand Schneiden
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "T_ST2_VOR"       E16.3  -- Taster Schere vor
U     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
UN    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
=     "ST2_H_SCHNEIDEN" M38.0  -- Stampfen 2 Hand Kabel abschneiden

Netzwerk: 32     Hand Stampfen
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "T_ST2_STAMPFEN"  E16.5  -- Taster Stampfen
U     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
UN    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
=     "ST2_H_STAMPFEN"  M38.1  -- Stampfen 2 Hand Stampfen Start

Netzwerk: 33     Hand Schere zurück
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "T_ST2_ZURÜCK"    E16.4  -- Taster Schere zurück
U     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
UN    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
```

```
=     "ST2_H_SCHERE_RÜCK" M38.2  -- Stampfen 2 Hand Schere zurück

Netzwerk: 34     Hand Heben
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "T_ST2_HEBEN"     E15.7  -- Taster Stampfkopf heben
U     "ZW_SICH"         E126.7  -- Zweihand Sicherheit
UN    "ST2_ZWEIH"       T50  -- Stampfen 2 Zweihand
=     "ST2_H_KOPF_HEBEN" M38.3  -- Stampfen 2 Hand Kopf heben

Netzwerk: 35     Stampfen 2 Start
U     "ST2_STAMPF"      M36.3  -- Stampfen 2 Stampfen Start
U     "ST2_STAMPEN_START" M37.6  -- Stampfen 2 Stampfcontroller Start
UN    "ST2_ABLAUF"      M36.6  -- Stampfen 2 Ablauf Ende
UN    "ST2_SCHNEIDEN"   M36.4  -- Stampfen 2 Kabel abschneiden
=     "START_ST2"       A2.3  -- Stampfen 2 Start

Netzwerk: 36     Motor - Kabelabroller
U     "VW_ST2"          E14.4  -- VW Stampfen 2
U     "MOTOR_EIN"       A125.1  -- Motore Ein
U(
UN    "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "KABEL_T"         T78  -- Verzögerung Kabelabroller
O
U     "HANDBETRIEB"     M0.0  -- Maschine ist in Handbetrieb
U     "KABEL_EA"        E23.0  -- Kabelabroller Ein/Aus
)
UN    "KABEL_DL"        E22.7  -- Kabelabroller Dauerlauf aus
=     "MOT_KAB2"        A1.6  -- Motor - Kabelabroller

Netzwerk: 37     Verzögerung Kabelabroller
U     "KABEL_EA"        E23.0  -- Kabelabroller Ein/Aus
L     S5T#10S
SE    "KABEL_T"         T78  -- Verzögerung Kabelabroller
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 38     Stampfen 2 Blasen
U     "ST2_START"       M36.1  -- Stampfen 2 gestartet
U     "MOTOR_EIN"       A125.1  -- Motore Ein
U     "VW_ST2"          E14.4  -- VW Stampfen 2
L     S5T#1S
SA    "ST2_BLASEN"      T51  -- Stampfen 2 Blasen
NOP   0
NOP   0
NOP   0
NOP   0

Netzwerk: 39     Stampfen 2 Blasen
U     "ST2_BLASEN"      T51  -- Stampfen 2 Blasen
U     "MOTOR_EIN"       A125.1  -- Motore Ein
U     "VW_ST2"          E14.4  -- VW Stampfen 2
=     "MV__12"          A44.1  -- Stampfen 2 Blasen
```

Anhang C: Analyse durch Hilfsprogramme

In der Folge wird das Analyseprogramm als Hilfsfunktion beschrieben. Als Beispiele werden das bereits genannte SPS-Programm der Bohrmaschine und weitere Bausteine herangezogen.

Beispiele von SPS Programmbausteinen

Aufgabe des Programms „SPS“ ist die Analyse von Textdateien von SPS-Programmen. In seiner Testversion ist des dazu geeignet, in AWL formulierte SPS-Programme, die für die Steuerung S7 Baureihe 300 entwickelt worden sind, aufzubereiten. Dazu werden Programmbausteine im Programmeditor in AWL dargestellt und im Gegensatz zur Darstellungsart im Anhang B die Funktionen „Anzeigen mit symbolischer Darstellung“ und „Symbolinformationen“ ausgeschaltet. Die Kommandofolge wird mit „Ausdrucken in eine Datei“ erzeugt und ist die Quelle für die Analyse.

Zuerst wird der Text zeilenweise analysiert. In Folge wird dabei ein erweiterter STEP 7-Programmtext erzeugt, Klammerungen entsprechend der beschriebenen Programmlogik eingeführt und wo erforderlich Ladebefehle eingesetzt. Als zweiter Schritt wird der STEP 7-Code in eigenen Code umgewandelt. Im dritten Schritt wird der eigene Code in Textform umgewandelt und ausgegeben. Im vierten Schritt wird die Verwendung der genutzten Referenzen für die Zustände der Programmelemente ermittelt und als „cross reference“ ausgegeben. Während für die Quelldatei die Extension „**`.prn`**“ vorgesehen ist erfolgt die Ausgabe in einer gleichnamigen Datei mit der Extension „**`.txt`**“.

In dieser Textdatei sind Eingänge, Ausgänge usw. geordnet. Diese Textdatei ist geeignet, als Quelle für den Verifikator nach [KP12] zu dienen, wobei im Verifikator selbst einige Ergänzungen erforderlich sind.

Baustein FC21 Bohren

Das Quellprorgamm des Programmbausteins FC21wird mit der Funktion „Drucke in eine Datei“ als Textdatei erzeugt. Dieses Quellprogramm wird eingelesen und durchläuft einige Analyseschriite.

Es sind mehrere Arbeitsschritte vorgesehen. Im Schritt 1 wird der Quelltext formatiert und die Elemente jeder Programmzele in einen Code umgeandelt. In der Einstellung der Ausgabe wird auf die symbolische Darstellung und die Kommentierung aus Platzgründen verzichtet. Uuter anderem wird auch der Speicherbedarf reduziert. Aus dem Quelltext (Tab. 18 links)

wird der Code gewonnen (Tab. 18 rechts). Als Code für das Netzwerk wurde als interne Bezeichnung die Abkürzung „NET“ gewählt, der zugehörige Code im Hilfsprogramm dafür ist die Zahl 17.

Quelltext (Textfile, erzeugt im Programmeditor	Transformiertes Ergebnis	Code	Op	Byte	Bit
Netzwerk: 1 Bohren 1 Sicherheit	**NET**	**17**			
U(	**U(**	**2**			
O E 4.3	**O E 4.3**	**3**	**1**	**4**	**3**
O E 126.7	**O E 126.7**	**3**	**1**	**126**	**7**
ON M 0.0	**ON M 0.0**	**4**	**3**	**0**	**0**
)	**)**	**6**			
L S5T#200MS	**L S5T#200MS**	**7**			
SE T 11	**SE T 11**	**13**	**4**	**11**	
NOP 0	**NOP 0**	**18**			
NOP 0	**NOP 0**	**18**			
NOP 0	**NOP 0**	**18**			
NOP 0	**NOP 0**	**18**			

Tab. 18: FC21 Umwandlung (Analyseschritt 1)

In der gleichen Art und Weise wird der ausgewählte Programmabschnitt analysiert.

Die eigentliche Analyse erfolgt im Schritt 2. In diesem Schritt werden diverse Ergänzungen bzw. auch der Austausch von Befehlen in die benötigten „Lade-Kommandos“ vorgenommen. Wie bereits im Zuge der Ausführungen diskutiert, sind die Parameter für die Zeitvorwahl hier nicht von Interesse und können genauso wie die vom Programmeditor eingefügten NOP (no Operation) weggelassen werden. Auch die Parametrierung von Funktionen wie hier die Zeitfunktion ist nicht von Interesse, weil wie bereits ausgeführt nur die Zustände „Zeit abgelaufen“ bzw. „nicht abgelaufen“ Auswirkungen auf den Programmablauf haben. Damit reduziert sich der für die Verifikation bereitgestellte Code in die Befehlsfolge (Tab. 19)

```
NET
U(
LD  E 4.3
O   E 126.7
ON  M 0.0
)
SE  T 11
```

Tab. 19: FC21 Ergebnis (Analyseschritt 2)

Bezogen auf die Funktion ist das Einklammern der Kommandofolge im Beispielnetzwerk nicht notwendig, schadet aber nicht (Tab. 19).

Im Schritt 3 erfolgt die Ausgabe in eine Ergebnisdatei im Textformat. Im Schritt 4 wird eine geordnete Liste der verwendeten Speicherplätze für die Zustandsinformationen erzeugt. Im Beispiel Bohrstation 1 (Tab. 20) sind das:

```
Cross-Reference:
Eingänge
E 3.6   E 4.0   E 4.1   E 4.2   E 4.3   E 23.5  E 126.7
Ausgänge
A 0.7   A 31.0  A 32.4  A 125.1
Merker
M 0.0   M 0.1   M 0.2   M 0.3   M 0.6   M 10.1  M 20.0  M 20.1
M 20.2  M 20.3  M 20.4  M 20.5  M 20.6  M 100.0
Zeitglieder
T 11    T 12    T 13    T 52
```

Tab. 20: FC21 Querverweise (kompletter Baustein)

Baustein FC34 Wenden

Ähnlich wie voher wird der Baustein mit dem Analyseprogramm bearbeitet. Im Baustein für Wenden treten einige komplexere Netzwerke auf. Für das Analyseprogramm werden folgende Regeln im Programm umgesetzt (Beispiele Tab. 21 und Tab. 22):

- Wenn ein neues Netzwerk erkannt wird, wird das Verknüpfungsergebnis auf logisch EIN gesetzt. Das ist deshalb notwendig, weil bei einem Kommando **UND KLAMMER** an der ersten Position das Klammerergebnis mit einem vorher festgestellten Zwischenergebnis verknüpft wird. Gleichzeitig wird die Markierung eines Netzwerks dazu genutzt, den Austausch des ersten Kommandos in ein entsprechendes LADE Kommando zu steuern.
- Wenn das erste Kommando ein Kommando mit Argumenten in einem neuen Netzwerk an ester Stelle steht wird dieses Kommando durch einen Ladebefehl ersetzt.
- Das Kommando ODER ohne Argumente wird durch das Kommando **ODER KLAMMER** ersetzt. Wurde ein solcher Klammbefehl eingefügt, muss an geeigneter Stelle das Kommando **KLAMMER SCHLIESSEN** eingefügt werden.
- Wenn ein Kommando mit Argumenten unmittelbar auf eine geöffnete Klammer folgt wird dieses Kommando durch einen Ladebefehl ersetzt.
- Wenn ein Klammerbefehl eingefügt worden ist dann muss diese Klammer vor der nächstfolgenden schließenden Klammer geschlossen werden. Dazu wird das Kommando **KLAMMER SCHLIESSEN** eingefügt.
- Wenn ein Klammerbefehl eingefügt worden ist dann muss die Klammer einer weiteren neu eingefügten Oder Klammer geschlossen werden. Dazu wird das Kommando **KLAMMER SCHLIESSEN** eingefügt.
- Wird innerhalb eines UND Kammerausdrucks ein ODER Klammerausdruck eingefügt, wird das Kommando **KLAMMER SCHLIESSEN** entweder vor einer weiteren ODER Klammer oder unmittelbar bevor die UND Klammer geschlossen wird eingefügt.

Transformiertes Ergebnis							Neuer Code	Anmerkung
NET			17				**NET**	
U(			2				**U(**	
U	M	0.0	0	3	0	0	**LD M 0.0**	← Ladebefehl
U	M	50.0	0	3	50	0	**U M 50.0**	
U	E	11.4	0	1	11	4	**U E 11.4**	
U	E	126.7	0	1	126	7	**U E 126.7**	
UN	T	53	1	4	53		**UN T 53**	
O			3				**O(**	← Oder Klammer auf
U	M	0.1	0	3	0	1	**LD M 0.1**	← Ladebefehl
UN	M	11.7	1	3	11	7	**UN M 11.7**	
O	M	51.1	3	3	51	1	**O M 51.1**	
							)	← Oder Klammer zu
)			6				**)**	
U	E	11.2	0	1	11	2	**U E 11.2**	
U	M	0.2	0	3	0	2	**U M 0.2**	
UN	M	51.0	1	3	51	0	**UN M 51.0**	
=	M	51.1	10	3	51	1	**= M 51.1**	

Tab. 21: FC34 Netzwek 3

Transformiertes Ergebnis							Neuer Code	Anmerkung
NET			17				**NET**	
U(			2				**U(**	
U	M	51.5	0	3	51	5	**LD M 51.5**	← Ladebefehl
UN	M	52.0	1	3	52	0	**UN M 52.0**	
U	E	11.2	0	1	11	2	**U E 11.2**	
O			3				**O(**	← Oder Klammer auf (1)
U(			2				**U(**	
U	M	53.1	0	3	53	1	**LD M 53.1**	← Ladebefehl
UN	M	53.4	1	3	53	4	**UN M 53.4**	
O			3				**O(**	← Oder Klammer auf (2)
U	M	53.6	0	3	53	6	**LD M 53.6**	
UN	M	54.2	1	3	54	2	**UN M 54.2**	
							)	← Oder Klammer zu (2)
)			6				**)**	
U	E	11.3	0	1	11	3	**U E 11.3**	
							)	← Oder Klammer zu (1)
)			6				**)**	
U	M	0.2	0	3	0	2	**U M 0.2**	
=	A	36.2	10	2	36	2	**= A 36.2**	

Tab. 22: FC34 Netzwek 33

```
Cross-Reference:
Eingänge:
E 11.2  E 11.3  E 11.4  E 11.5  E 11.6  E 11.7  E 12.0  E 12.1
E 12.2  E 12.3  E 12.4  E 12.5  E 12.6  E 12.7  E 13.0  E 13.1
E 13.2  E 13.3  E 13.5  E 13.6  E 23.5  E 126.7
Ausgänge:
A 2.4   A 2.5   A 30.4  A 30.5  A 36.0  A 36.2  A 36.4  A 36.6
A 37.0  A 37.2  A 37.4  A 37.6
Merker:
M 0.0   M 0.1   M 0.2   M 0.3   M 0.6   M 11.7  M 12.0  M 50.0
M 50.1  M 50.2  M 50.3  M 51.0  M 51.1  M 51.2  M 51.3  M 51.4
M 51.5  M 51.6  M 51.7  M 52.0  M 52.1  M 53.0  M 53.1  M 53.2
M 53.3  M 53.4  M 53.5  M 53.6  M 53.7  M 54.0  M 54.1  M 54.2
M 54.3  M 102.1 M 102.2
Timer:
T 53    T 54    T 55    T 66
```

Tab. 23: FC34 Querverweise (kompletter Baustein)

Aus der Liste der verwendeten Elemente können die Zustandsabbilder generiert werden (Tab. 23).

Die Idee, den Baustein auf Grund der sehr großen Anzahl von zu beachtenden Zuständen drängt sich auf, weil zwei unterschiedliche Abläufe für den Vorgang Wenden zu realisieren waren. Untersucht wird eine Teilung in den Programmteil „Wenden 1“, „Wenden 2“ und den allgemein genutzen Teil bestehend aus der Auswertung und der in beiden Programmteilen genutzen gemeinsamen Ansteuerungen der Ausgänge. Intuitiv auf Grund der Netzwerkkommentierungen bietet sich an für den Programmteil „Wenden 1“ die Netzwerke 2 bis 14 und für den Programmteil „Wenden 2“ die Netzwerke 15 bis 30 zusammenzufassen und den Rest des Programms als allgemeinen Teil zu betrachten. Analysiert man diese Programmteile einzeln, erhält man folgende Referenzen (Tab. 24, Tab. 25 und Tab. 26):

```
Cross-Reference:
Eingänge:
E 11.2  E 11.4  E 11.5  E 11.6  E 11.7  E 12.0  E 12.1  E 12.2
E 12.3  E 12.5  E 12.6  E 12.7  E 13.0  E 13.1  E 13.2  E 13.6
E 126.7
Merker:
M 0.0   M 0.1   M 0.2   M 0.3   M 0.6   M 11.7  M 50.0  M 50.3
M 51.0  M 51.1  M 51.2  M 51.3  M 51.4  M 51.5  M 51.6  M 51.7
M 52.0  M 52.1  M 102.1
Timer:
T 53    T 66
```

Tab. 24: FC34 Querverweise Programmteil Wenden 1

```
Cross-Reference:
Eingänge:
E 11.3  E 11.4  E 11.5  E 11.6  E 12.1  E 12.2  E 12.4  E 12.6
E 13.3  E 13.5  E 13.6  E 126.7
Merker:
M 0.0   M 0.1   M 0.2   M 0.3   M 0.6   M 12.0  M 50.1  M 50.3
M 53.0  M 53.1  M 53.2  M 53.3  M 53.4  M 53.5  M 53.6  M 53.7
M 54.0  M 54.1  M 54.2  M 54.3  M 102.2
Timer:
T 53    T 54    T 55
```

Tab. 25: FC34 Querverweise Programmteil Wenden 2

```
Cross-Reference:
Eingänge:
E 11.2  E 11.3  E 11.4  E 23.5  E 126.7
Ausgänge:
A 2.4   A 2.5   A 30.4  A 30.5  A 36.0  A 36.2  A 36.4  A 36.6
A 37.0  A 37.2  A 37.4  A 37.6
Merker:
M 0.0   M 0.2   M 50.2  M 50.3  M 51.1  M 51.2  M 51.3  M 51.4
M 51.5  M 51.6  M 51.7  M 52.0  M 52.1  M 53.1  M 53.2  M 53.3
M 53.4  M 53.5  M 53.6  M 53.7  M 54.0  M 54.1  M 54.2  M 54.3
Timer:
T 53    T 55
```

Tab. 26: FC34 Querverweise Programmteil Wenden allgemein

Die Station Wenden wird optional an die Fertigungseinrichtung angedockt. Ist die Station abgeschaltet oder gar nicht montiert, müssen wengistens die Überwachungen melden, dass kein Eingriff auf den Werkstücktransport gegeben ist (hier die so genannten „Drehbedingungen“). Der Programmteil „Wenden 1“ bzw. „Wenden 2“ beschreibt den jeweiligen Programmablauf, der Programmteil „Wenden allgemein“ die Ansteuerung der Ausgänge. Zum besseren Verständnis sollten daher jeweils der Programmablauf mit dem allgemeinen Teil betrachtet werden. Grundsätzlich können konkurrierende Programmabläufe nie zeitgleich arbeiten. Das bedeutet, dass die Programme „Wenden 1“ und „Wenden 2“ sich wechhselseitig ausschließen und gegeneinander gesperrt sein müssen. Durch genauere Untersuchung des SPS-Programmcodes ist dieser Umstand einfach feststellbar. In Summe hält sich die Reduktion der Komplexität in Grenzen.

Baustein FC38 Stampfen

Im Wesentlichen bietet der Baustein für das Kabeleinstampfen gegenüber den vorher untersuchten Netzwerken keine Besonderheiten. Nur das Netzwerk 24 beginnt mit zwei öffenden **UND KLAMMER** Ausdrücken (Tab. 27). Die einzige notwendige Veränderung beschränkt sich auf das Ersetzen eines Kommandos in ein **LADE** Kommando:

Transformiertes Ergebnis							Neuer Code		Anmerkung
NET			17				**NET**		
U(			2				**U(**		
U(			2				**U(**		
O	M	36.5	3	3	36	5	**LD**	**M 36.5**	← Ladebefehl
O	M	37.0	3	3	37	0	**O**	**M 37.0**	
)			6				**)**		
U	E	14.7	0	1	14	7	**U**	**E 14.7**	
ON	E	14.5	4	1	14	5	**ON**	**E 14.5**	
)			6				**)**		
UN	E	15.2	1	1	15	2	**UN**	**E 15.2**	
R	A	40.6	12	2	40	6	**R**	**A 40.6**	

Tab. 27: FC38 Netzwek 24

```
Cross-Reference:
Eingänge:
E 14.4  E 14.5  E 14.6  E 14.7  E 15.0  E 15.1  E 15.2  E 15.3
E 15.7  E 16.0  E 16.1  E 16.2  E 16.3  E 16.4  E 16.5  E 22.7
E 23.0  E 124.5 E 124.7 E 126.7
Ausgänge:
A 1.6   A 2.3   A 40.4  A 40.6  A 41.0  A 44.1  A 44.2  A 45.4
A 45.6  A 125.1
Merker:
M 0.0   M 0.1   M 0.2   M 0.3   M 0.6   M 11.6  M 36.0  M 36.1
M 36.2  M 36.3  M 36.4  M 36.5  M 36.6  M 36.7  M 37.0  M 37.1
M 37.2  M 37.3  M 37.4  M 37.6  M 37.7  M 38.0  M 38.1  M 38.2
M 38.3  M 101.5 M 101.6 M 101.7
Timer:
T 48    T 49    T 50    T 51    T 78
```

Tab. 28: FC38 Querverweise Programmteil Stampfen

Zusammenfassung

Der Test des Analyseprogramms zeigt, dass selbst als eher ungewöhnlich zu bezeichnende Kombinationen von Kommandos sich in eine Darstellungsart umformen lassen, die dem Quellprogramm entspricht. Damit lässt sich jede Programmzeile in den eingeführten Algorithmus, der durch Flussdiagramme dargestellt werden kann, umwandeln.

Die notwendigen Startbedingungen für den Algorithmus stammen aus Kombinationen der Zustandswerte, die durch das Analysewerkzeug in ihrer Anzahl auf die untersuchten Programmbausteine eingeschränkt werden und damit eine Teilmenge des gesamten Zustandsraumes abbilden.

Als Voraussetzung muss sichergestellt sein, dass die Zuweisung insbesondere von Ausgängen exklusiv erfolgt, d.h. dass schreibende Operartionen nur in einem Baustein stattfinden. Vergleicht man die Querverweise für die Ausgänge in den genannten Bausteinen, wird der Ausgang **`A 125.1`** offenbar mehrfach verwendet (Tab. 29). Im gegenständlichen Programm

stellt das keinen Fehler dar, weil dieser Ausgang nur lesend in den Kommandoketten der betrachteten Bausteine genutzt worden ist.

Cross-Reference:							
Ausgänge:							
A 0.7	A 1.6	A 2.3	A 2.4	A 2.5	A 30.4	A 30.5	A 31.0
A 32.4	A 36.0	A 36.2	A 36.4	A 36.6	A 37.0	A 37.2	A 37.4
A 37.6	A 40.4	A 40.6	A 41.0	A 44.1	A 44.2	A 45.4	A 45.6
A 125.1							

Tab. 29: Querverweise Ausgänge untersuchte Programmteile

Legende:
- Referenzen im Programmbaustein Bohren
- Referenzen im Programmbaustein Wenden
- Referenzen im Programmbaustein Stampfen
- gemeinsam genutzte Referenzen

Anwendung der Rückwärtsanalyse

Zweck der Rückwärtsanalyse ist, Zusammenhänge aufzudecken. Die Rückwärtsanalyse kann sehr unterschiedliche Formen der Programmuntersuchung unterstützen. Im Wesentlichen wird das SPS-Programm vom Programmende ausgehend schrittweise analysiert, um Abhängigkeiten in spezifische Cross-Referenzen im SPS-Programm zu lokalisieren und in Form einer Aufzählung auszugeben. Ausgehend von der Frage, wie die gegenseitigen Abhängigkeiten von Zuweisungen (z.B. von Ausgängen) feststellbar ist, werden innerhalb eines vordefinierten Suchraums für alle Zuweisungsoperationen die bestimmenden Programmreferenzen zusammengestellt.

Zweck des vorgestellten Algorithmus ist auch, den Untersuchungsraum der Zustände durch isolieren von nicht erforderlichen Zuständen zu mimimieren, sodass nur die benötigten Informationen in eine Untersuchung einbezogen werden.

Als vordefinierter Suchraum wird ein Teil des SPS-Programms als Prüfling festgelegt. Es bietet sich dabei an, funktionell zusammenhängende Programmbereiche dann zu wählen, wenn auch der unmittelbare Bezug zu einer Aufgabenstellung für die gesteuerte Funktion gegeben ist. Das ist jedoch keine Bedingung.

Algorithmus zur Eingrenzung auf bestimmte Programmreferenzen

Im Prüfling wird festgestellt, welche Programmreferenzen als Zuweisungen von Zuständen genutzt sind. Für jede Programmreferenz wird dazu eine Aufstellung gemacht, in der alle „Er-

zeugenden“ bereitgestellt werden. Erzeugende in diesem Sinn sind Eingänge, Merker bzw. Ausgänge und sonstige Programmelemente.

- Eingänge sind bezogen auf ihre spätere Nutzung der Ergebnisse unproblematisch, weil deren Zustände unmittelbar aus der Konfiguration der Zustände abgelesen werden können. Eingänge oder nicht zugewiesene Merker und Ausgänge können daher ohne weitere Zwischenschritte in die Aufstellung der Erzeugenden zu einer im Prüfling zugewiesenen Programmreferenz aufgenommen werden. Sie könnten informell als Erzeugende der Grundstufe bezeichnet werden (z.B. Erz(0)).
- Neben den Eingängen können aber auch Merker bzw. Ausgänge als Verknüpfungselemente verwendet worden sein. Diese Gruppe von Programmreferenzen sind im SPS-Programm aus der im Programm festgelegten Kombination von Verknüpfungen definiert worden. Die dort verwendeten Programmreferenzen sind Träger von Zustandsinformationen uns müssen genauso wie zuvor untersucht werden. Die dort verwendeten Erzeugenden können Erzeugende von den ursprünglichen Erzeugenden sein. Sie könnten informell als Erzeugende einer nächst höheren Stufe bezeichnet werden. Die Schreibweise könnte Erz(1) lauten.
- Der Vorgang wird iterativ solange fortgesetzt, bis alle Erzeugenden für die angefragte Zuweisung festgestellt worden sind. Es entsteht damit eine spezielle Zuordnungsliste der Erzeugenden für die als Zuweisung verwendete Programmreferenz.
- Werden z.B. Ausgänge oder Merker nur lesend im untersuchten Programmabschnitt verwendet, gelten diese bereits als Erzeugende. Das ist immer dann der Fall, wenn die Ansteuerung solcher Ausgänge oder Merker in einem anderen Programmteil vorgenommen wird.
- Danach wird der beschriebene Vorgang auf die nächste Zuweisung angewendet, solange, bis alle Zuweisungen bestimmt sind. Es entsteht eine ganz spezielle Cross-Referenz Liste, die angibt, welche Programmreferenzen aus dem Zustandsraum das ausgewiesene Programmelement manipulieren können.

Zum besseren Verständnis wird ein einfaches Beispiel herangezogen.

Gegeben seien drei Netzwerke, die voneinander abhängig sind. Ohne genauer auf das SPS-Proramm einzugehen gelten folgende Abhängigkeiten:

- Der Ausgang **A7.7** wird von **E3.3**, **M4.6**, und **A0.2** beeinflusst, (Erz(0))
- Der Merker **M4.6** wird von **E2.4**, **E2.6** und **M0.6** beeinflusst, (Erz(1))
- Der Merker **M0.6** wird von **E3.3**, **E1.6** und **A0.2** beeinflusst, (Erz(2))
- Der Ausgang **A0.2** wird nur lesend verwendet.

Anmerkung: in diesem Beispiel ist der Ausgang **A7.7** oder der Merker **M4.6** nicht von sich selbst abhängig, da sie bei den Erzeugenden nicht angeführt sind. Wäre eine dieser Pro-

grammreferenzen auch von sich selbst abhängig, muss diese in der Aufzählung der beeinflussenden Programmreferenzen enthalten sein.

Gesucht wird unter den gegebenen Voraussetzungen nach den Abhängigkeiten von Ausgang **A7.7** mit dem Ziel zu fragen, welche Programmreferenzen den genannten Ausgang bestimmen. Sind diese Programmreferenzen bekannt, ist auf Grund ihrer Zustände der Zustand von Ausgang **A7.7** bestimmt. Zusätzlich gilt, dass der Zustand von Ausgang **A7.7** nicht durch sich selbst beeinflusst werden kann. Das ist insoweit von Bedeutung, weil dadurch der Zustand Ausgang **A7.7** eindeutig bestimmbar wird.

Abarbeitung und Auswertung durch den Algorithmus:

- Schritt 1: Ausgangslage, Übernahme der Erzeugenden(0)
  E3.3, **M4.6**, **A0.2**→ **A7.7**
- Schritt 2: es existiert im Progamm eine Programmreferenz **M4.6**, die ihrerseits zugewiesen worden ist. Diese Programmreferenz **M4.6** ist bezogen auf **A7.7** eine Erzeugende(1). Die Liste der Erzeugenden wird durch Einsetzen erweitert.
  E3.3, [**E2.4**, **E2.6**, **M0.6**, **A0.2]**→ **A7.7**
- Schritt 3: Es werden jene Programmreferenzen gestrichen, die bereits in der ursprünglichen Aufstellung vorhanden waren. Schritt 3 ist das Ersetzen der neu hinzugekommenen Erzeugenden(2): **E3.3**, [**E2.4**, **E2.6**, [~~**E3.3**~~, **E1.6**, ~~**A0.2**~~**]**, **A0.2]**→ **A7.7** (Programmreferenz gestrichen, weil bereits vorhanden; Ersetzen der neu hinzugekommenen Erzeugenden(2))
- Schritt 4: Zusammenfassen, weil keine weiteren Programmbeziehungen gegeben sind
  E1.6, **E2.4**, **E2.6**, **E3.3**, **A0.2**→ **A7.7**
  Anmerkung: **A0.2** ist laut der Angabe nur lesend verwendet, ist daher in dieser Konstellation keine Erzeugende und wird daher auch nicht ersetzt. Die Liste wird geordnet, weil keine weiteren zuordenbaren Erzeugenden im Programmabschnitt des Prüflings vorhanden sind. Zuletzt enthält die Zuordnung nur mehr Eingänge und im untersuchten Programmabschnitt nicht zugewiesene Merker und Ausgänge.

Für den untersuchten Ausgang gilt, dass sein logischer Zustand durch die Zustände seiner erzeugenden Programmreferenzen bestimmt ist. Eine eineindeutige Bestimmung ist nur dann möglich, wenn alle Werte für die Zustände festgelegt werden können.

Innere Abhängigkeiten

Innere Abhängigkeiten sind dann direkt aus der Cross-Reference feststellbar, wenn diese nicht nur als Zuweisung, sondern auch auch als Erzeugende einer Programmreferenz angeführt sind. Im hier angeführten Beispiel ist das der Merker **M 53.5** (Tab. 30).

M 53.5 E 12.2 M 50.3 M 53.0 M 53.4 **M 53.5**

Tab. 30: Querverweise Beispiel

Das bedeutet, dass solche Programmreferenzen bei der Ersetzung nicht gestrichen werden dürfen, sondern dass sie auch in die Aufstellung der Erzeugenden mit aufgenommen werden müssen. Grund dafür ist, dass in derartigen Fällen das Ergebnis einer Berechnung für einen bestimmten Programmzyklus vom Ergebnis der Berechnung für einen anderen Programmzyklus abhängig ist.

Nicht aus der Cross-Reference ablesbar ist das für die Kommando SET und RESET. Hier liegt immer eine innere Abhängigkeit vor, d.h. dass für alle Kommandos „**S**“ und „**R**“ ihre Programmreferenz in die Aufstellung der Erzeugenden mit aufzunehmen sind.

Anwendungsbeispiel: Bestimmen von Zusammenhängen

Die Rückwärtsananlyse ist vorgesehen, um insbesondere auch in „unbekannten“ SPS-Logiken Zusammenhänge herauszufinden. Da bei einer SPS nur die Ausgänge Wirkung auf den Zustand der gesteuerten Einrichtung zeigen können, wird diese Methode in erster Linie auf Ausgänge anzuwenden sein.

Hier wird die Methode auf einen bekannten Baustein angewendet. In der Programmbeschreibung des Bausteins FC38 Stampfen existiert die Teilfunktion für das Abschneiden eines Kabels. Einige Programmvarianten für eine „Schere“ wurden bereits ausführlich diskutiert.

Bei der Rückwärtsanalyse von beliebigen Prorgammen muss die Position der Kommandofolgen nicht bekannt sein. In der Regel wird ein SPS-Programm mehrere Funktionen zu erfüllen haben, die auch in beliebiger Ordnung programmtechnisch umgesetzt sein können.

Eine SPS bewirkt etwas durch die Ansteuerung ihrer Ausgänge. Mit Hilfe der Rückwärtsanalyse wird das SPS-Programm untersucht, in dem an einer noch unbekannten Stelle im Prorgammverlauf die die Ausgänge der beiden Ventile zur Ansteuerung der Scherenzylinder programmiert sind. Gefragt wird, welche Zustände von den im Untersuchungsraum enthaltenen Programmreferenzen bekannt sein müssen, um die möglichen Programmzustände für die Ausgänge der Ventile „Schere Bewegung“ und „Schere Schnitt“ zu bestimmen. Der Vorgang ist ähnlich wie im vorher diskutierten Beispiel. Bekannt sein muss, welche Ausgänge aus dem Adressraum der SPS verwendet worden sind. Diese sind Ausgangspunkt für die Untersuchung. Im Beispiel des FC38 sind das die Ausgänge **"MV_Z40" A40.4** (Stampfstation 2,

Scherenzylinder lang für die Bewegung der Schere) und **"MV_Z41" A40.6** (Stampfstation 2, Scherenzylinder kurz zum Abschneiden des Kabels).

Der Algorithmus liefert als Ergebnisse im Schritt 1 die Ausgangslage (Tab. 31):

```
A 40.4
E 14.4 E 14.5 E 14.7 E 15.0 E 15.2 M 0.2  M 36.4 M 36.5 M 36.7 M 37.0
A 40.6
E 14.4 E 14.5 E 14.7 E 15.1 E 15.2 M 36.5 M 37.0
```

Tab. 31: Querverweise Ausgangslage (Schritt 1)

Gefunden wurden die zur Ansteuerung der Ausgänge genutzten Programmreferenzen. Signalzustände von Eingängen entsprechen bestimmten Eingangskonfigurationen dieser Programmelemente, deren Zustände als Festwerte einem bestimmten Arbeitsschritt zuordenbar sind (z.B. Schere in Grundstellung). Signalzustände von Merkern sind das innere „Gedächtnis“, das den Programmablauf widerspiegelt. Als variable Zustände werden sie weiter untersucht und gefragt, ob auch ihnen Zustände als Festwerte zugeordnet werden können. Jeweils die neu in einen Untersuchungsschritt mit aufgenommene Referenz ist durch Fettdruck hervorgehoben.

Schritt 2: Gefragt ist, welche Programmreferenzen sind auf Grund der ersten Auswertung und der als Basiswerte betrachteten Zustände neu hinzugekommen, welche Zustände der Programmreferenzen werden durch das SPS-Programm in der Form von Zuweisungen gebildet (z.B. verändert) und welche wurden nur lesend abgefragt (bleiben unverändert). Diese Programmreferenzen werden weiter untersucht. Existieren bereits untersuchte Programmreferenzen, brauchen diese nicht neuerlich berücksichtigt werden.

Besonders zu beachten sind Programmreferenzen mit Gedächtnisfunktion wie z.B. SET und RESET Kommandos. Auf diese Programmreferenzen kann bei der Zustandsbestimmung nicht verzichtet werden. Sie müssen in die Aufstellung als Erzeugende mit aufgenommen werden, In der Aufstellung sind diese Programmreferenzen gekennzeichnet (Tab. 32).

```
M 0.2  ... nicht zugewiesen, wird übernommen
M 36.4
E 14.4 E 14.5 M 0.2  M 0.3  M 0.6  M 11.6 M 36.6
M 36.5
E 14.4 E 14.7 E 15.0 E 15.1 E 15.2 E124.5 M 36.2 M 36.4 M 36.6
M 36.7
E 14.5 E 15.1 M 0.0  M 0.2  M 36.7 M 37.0 M 38.0
M 37.0
E 15.0 M 0.0  M 0.2  M 37.0 M 38.2
```

Tab. 32: Querverweise Schritt 2

Schritt 3: in den weiteren Schritten wird wie zuvor vorgegangen. Neu hinzugekommen sind sechs Programmreferenzen (Tab. 33).

```
M 0.0  ... nicht zugewiesen, wird übernommen
M 0.3  ... nicht zugewiesen, wird übernommen
M 0.6  ... nicht zugewiesen, wird übernommen
M 11.6
E 14.4 E124.5 M 0.3  M 36.2 M 36.5 M101.6
M 36.2
E 14.4 M 0.2  M 0.3  M 0.6  M 11.6 M 36.1 M 36.5
M 36.6
E 14.4 E 14.6 E 15.3 E124.7 M 0.2  M 0.3 M 11.6 M 36.3 M 36.5 M101.5
M 38.0
E 16.3 E126.7 M 0.0  T 50
M 38.2
E 16.4 E126.7 M 0.0  T 50
```

Tab. 33: Querverweise Schritt 3

Schritt 4: wie zuvor. Neu hinzugekommen sind zwei Programmreferenzen (Tab. 34).

```
M 36.1
E 14.4 M 0.0 M 0.1 M 0.2 M 11.6 M 38.1
M101.5
E124.7
M101.6
E124.5 M 36.2 M 36.5
T 50
E 15.7 E 16.0 E 16.3 E 16.4 E 16.5 E126.7 M 0.0
```

Tab. 34: Querverweise Schritt 4

Schritt 5: wie zuvor. Neu hinzugekommen sind die letzten zwei offenen Programmreferenzen (Tab. 35).

```
M 0.1  ... nicht zugewiesen, wird übernommen
M 38.1
E 16.5 E126.7 M 0.0 T 50
```

Tab. 35: Querverweise Schritt 5

In der Zusammenfassung der Programmreferenzen Schere können alle mehrfach angeführten weggelassen werden. Die derart zusammengefasste Aufzählung beinhaltet alle Programmreferenzen für die Schere, die für die Bildung der bei der Verifikation benötigten Zustandsvektoren gebraucht werden.

Je mehr Zustände in die Untersuchung mit einbezogen werden müssen, desto umfangreicher wird der Aufwand (und führt im ungünstigen Fall an Grenzen der Berechenbarkeit also zum „state explosion problem“) aber auch desto weniger übersichtich ist die Untersuchung selbst. Dennoch wird der Aufwand auf das genau notwendige Maß in Bezug auf die zu betrachten-

den Zustände beschränkt, weil alle interessierenden Zusammenhänge für diese Untersuchung aufgedeckt worden sind. Das bedeutet, dass es keine weiteren Zustände aus anderen Programmelementen gibt, die direkt oder indirekt die betrachtete Funktion beeinflussen können. Daraus folgt, dass für diesen Untersuchungsfall alle anderen Zustände als nicht relevant von der Untersuchung ausgeschlossen werden können, ohne dass Einschränkungen in Bezug auf die Vollständigkeit gemacht werden muss.

Dazu ist anzumerken, dass in diesem Beispiel eine Teilfunktion eines Ablaufs mit allen Querbeinflussungen zum Gesamtablauf der Stampfmaschine berücksichtigt worden ist. Das inkludiert alle im SPS-Programm vorgesehenen Betriebsarten der Schere wie Automatikbetrieb, Handbetrieb und schrittweiser Betrieb. Auch bei einem nicht so gut bekannten Ablauf kann mit Hilfe dieser Analysetechnik herausgefunden werden, welche Programmsequenzen beispielsweise für die Handbetriebsart vorgesehen sind indem z.B. die Eingänge für die Bedienung als Ausgangspunkt für die Analyse herangezogen werden. Wenn etwa Betriebsarten einander ausschließen, das ist in der Regel der Fall, können bei getrennter Untersuchung weitere Zustände unberücksichtigt bleiben (z.B. es interessiert für die Untersuchung nur als Teilaspekt der automatische Ablauf).

Anhang D: Verifikation mit dem Verifikator

In [KP12] ist ein Werkzeug vorgestellt worden, das als Verifikator bezeichnet worden ist. In diesen Abschnitt wird die formale Darstellung der SPS mit den Konzepten des Verifikators in Relation gesetzt.

Allgemeines

Ein SPS-Programm besteht aus einer Folge von Befehlen, die in einer genau definierten Reihenfolge abgearbeitet werden. So gesehen besitzt ein SPS-Programm große Ähnlichkeit mit einem in Assembler geschriebenen Programm und den zugehörigen Flussdiagrammen, eine weniger große Ähnlichkeit mit einem in einer imperativen Programmiersprache geschriebenen Programm.

Für die Verifikation von SPS-Programmen können daher bekannte theoretische Ansätze für diese Sprachen weiterentwickelt werden. Der erste Schritt ist dabei, eine formale mathematische Schreibweise für SPS-Programme zu definieren. Im folgenden Abschnitt wird die Grundlage für diese Formalisierung gebildet. Dazu werden SPS-Programme in Prädikatenlogik (engl. first order logic) abgebildet. Die Prädikatenlogik erlaubt es, SPS-Programme zu formalisieren und auf ihre Gültigkeit zu überprüfen. Es ist erforderlich, dass zur korrekten syntaktischen Formulierung Zeichenfolgen, Terme und Formeln verwendet werden.

Wie bezogen auf ein SPS-Programm Zeichenfolgen, Terme und Formeln zu interpretieren sind, wird in der Folge erläutert. Ausgangspunkt ist der in [KP12] definierte und in Prädikatenlogik entwickelte Verifikator. Die dazu notwendigen Formalismen werden zuerst eingeführt, deren Bedeutung für das SPS-Programm danach. Die Umsetzung der Befehle eines SPS-Programms in Prädikatenlogik ist die Aufgabe eines Hilfsprogramms, das im Wesentlichen ein Programmlisting, wie es typisch von den Programmeditoren für SPS-Programme zur Dokumentation erzeugt werden kann, in die Syntax der Prädikatenlogik transformiert.

Informell dient die Prädikatenlogik dazu, Sachverhalte und Zusammenhänge zu beweisen. Sie bedient sich dazu einer streng formalen Ausdrucksweise. Die Prädikatenlogik als Erweiterung der Aussagenlogik untersucht atomare Aussagen hinsichtlich ihrer inneren Struktur. Die verwendeten Prädikate besitzen klar definierte Leerstellen, die, wenn in jede Leerstelle eine Bezeichnung eingesetzt wird, zu einer wahren oder falschen Aussage werden. Die Ausdrucksweise selbst hängt zunächst nicht unmittelbar mit einer Aufgabenstellung zusammen, im Gegenteil, sie ist streng formal gesehen sogar für die Aufgabenstellung ohne Bedeutung. Dieser Umstand erschwert es, eine Beweisführung anschaulich zu gestalten. Der Zusammenhang mit

der Aufgabenstellung wird erst durch die Interpretation hergestellt, die einerseits sowohl der formalen Ausdrucksweise der Prädikatenlogik genügen muss als auch andererseits mit der gefragten Aufgabenstellung übereinstimmt. Anders ausgedrückt ist die eigentliche Funktion, die eine SPS kraft ihrer Programmierung erfüllen soll, für die Verifikation des SPS-Programms vollkommen irrelevant.

Begriffe

Die Datenstruktur einer SPS kennt neben dem Programmspeicher für das SPS-Programm noch weitere Speicherbereiche, die den Programmablauf beeinflussen. Für die folgenden Erläuterungen sind von besonderem Interesse solche für die Eingangsinformationen, für Ausgangsinformationen und für variable Werte als Speicher für Zwischenergebnisse.

Die Menge der Eingangsinformationen wird als elektrische Signale an Eingangsklemmen einer SPS bereitgestellt und nach Filterung im zugehörigen Speicherbereich für die Programmbearbeitung bereitgestellt. Obwohl es auch die Möglichkeit bei bestimmten Steuerungsfamilien gibt, auf die Eingangsinformationen durch spezielle Befehle einer SPS direkt zuzugreifen, wird im Folgenden davon ausgegangen, dass die Eingangsinformation für die Dauer eines Programmdurchlaufs als konstant betrachtet werden dürfen. Diese Einschränkung ist für die überwiegende Anzahl der Fälle ohne Bedeutung. Die Eingangsinformation wird als eine Folge von Wahrheitswerten im so genannten Eingangsvektor zusammengefasst, der frühestens im folgenden Programmdurchlauf verändert werden kann. Der Datentyp des Eingangsvektors erlaubt dem SPS-Programm nur lesenden Zugriff. Ausgangsinformationen und variable Werte sind im Ausgangsvektor organisiert. Sie bestehen ebenfalls aus Folgen von zugehörigen Wahrheitswerten und sind sowohl lesend als auch schreibend nutzbar. Typisch für die meisten SPS ist, dass die Ausgangsinformation erst nach vollständiger Bearbeitung des SPS-Programms an die Ausgangschaltung und damit an die Ausgangsklemmen einer SPS weitergeleitet und dann bis zum nächstfolgenden Programmdurchlauf konstant gehalten wird.

Die Aufgabenstellung wird in dieser Arbeit durch das untersuchte SPS-Programm repräsentiert. Den Zusammenhang zur Aufgabenstellung bildet eine Bedeutungsfunktion. Die Bedeutungsfunktion wird durch die Bedeutung einzelner Berechnungsschritte des untersuchten SPS-Programms zusammengesetzt und kann sinngemäß mit den Ausgangsvektoren assoziiert werden. Die Assoziation wird durch die Namensgebung unterstützt. Dennoch darf dabei nicht außer Acht gelassen werden, dass die in Prädikatenlogik formulierten Konstrukte kein Teil eines SPS-Programms sind, selbst dann nicht, wenn die gewollte Ähnlichkeit den Nahebezug herstellt. So könnte etwa anstelle der Bezeichnung „line“ für eine Programmzeile jede beliebige regelkonforme Bezeichnung gewählt werden, ohne dass die Ergebnisse der Berechnungen dadurch beeinflusst werden. Anders ausgedrückt: die Summe der Formulierungen bildet

zwar ein SPS-Programm ab, ist aber keines. Dennoch wird im Folgenden immer wieder auf den Bezug, die Bedeutung bzw. die Bedeutungsfunktion zum SPS-Programm hingewiesen.

Einige weitere allgemeine Begriffe werden noch ergänzend erläutert, um die Lesbarkeit des Folgenden zu verbessern. In dieser Arbeit wird unter Algorithmus eine formale oder informelle Vorschrift zur Berechnung verstanden. Für den Begriff Funktion gilt für eine partielle Funktion, dass nicht alle Elemente über der Zielmenge definiert werden können und für die totale Funktion, dass alle Elemente über der Zielmenge definiert sind. Im Folgenden werden ausschließlich totale Funktionen verwendet.[63]

Auch wenn ein Algorithmus, der Elemente über einer Zielmenge bestimmt, nicht terminiert also unendlich lang weiterrechnet, wird von einer partiellen Funktion gesprochen. Die Eigenschaft des Terminierens ist eine notwendige Bedingung für das vorgestellte Verifikationsmodell. Termination wird dadurch erreicht, dass die einzelnen Programmdurchläufe, die Zyklen, eigenständig und separat untersucht werden.

Eine Funktion gilt als berechenbar, wenn für die betrachtete Funktion ein Algorithmus existiert, der Funktionswerte liefert.

Eine Menge ist rekursiv definiert, wenn ein Algorithmus existiert, der die untersuchten Elemente der Menge, beginnend von einem Startwert, erzeugt. Sie ist rekursiv aufzählbar, wenn ein Algorithmus, der die untersuchten Elemente der Menge schrittweise, beginnend von einem Startwert, erzeugt. Ein Element einer Menge ist entscheidbar, wenn ein Algorithmus existiert, der in einer endlichen Bearbeitungszeit feststellt, ob dieses Element in der untersuchten Menge enthalten ist oder nicht.

Für Zeichenfolgen (*engl. string*) gilt, dass diese bei ihrer Betrachtung als eine feste Anzahl (definiert durch ihre Länge) von willkürlich gewählten Zeichen verstanden werden.

Verifikation mittels PROLOG

Das Ziel dieser Arbeit besteht darin, die vorgestellte Methode zur Verifikation in tatsächlichen Anwendungen zum Einsatz zu bringen. Die Entscheidung ist zugunsten von PROLOG ausgefallen. Auch Norbert Völker [VÖ98] hat SPS-Funktionsbausteine mit Hilfe der Prädikatenlogik untersucht. Die in seiner Arbeit vorgestellte Methode ist extrem komplex und für

[63] Beispiel: Subtraktion innerhalb der Definitionsmenge von natürlichen Zahlen: ist das Resultat kleiner als Null, kann es nicht in natürlichen Zahlen ausgedrückt werden (partielle Funktion). Bei der totalen Funktion wird bei einem Resultat dieser Berechnung kleiner als Null als Resultat Null ausgegeben.

nicht-triviale Programme nur schwer anwendbar. So musste eine eigene Programmiersprache entwickelt und eingeführt werden (ST-). Die Anwendbarkeit auf herkömmliche SPS-Programmiersprachen, wie die in dieser Arbeit genutzte AWL, wurde dort erst gar nicht untersucht. Seine starke Orientierung an einer Typisierung, Verwendung von Prädikatenlogik höherer Stufen und der komplexen Isabelle-Objektlogik HOL macht die in der genannten Arbeit vorgestellte Verifikationsmethode für die tägliche Praxis nur schwer anwendbar.

Hier wurde daher eine Beschränkung auf Prädikatenlogik erster Stufe vorgenommen, wodurch die Komplexität überschaubar bleibt. Außerdem führt diese Vorgangsweise dann unmittelbar zu PROLOG, einer mittlerweile ausgereiften Programmiersprache mit stabilen, sehr mächtigen Implementierungen. Dadurch sind die erstellten Programme vergleichsweise sehr kurz und werden dadurch übersichtlich. In seiner Funktion ist der in [KP12] eingeführte Verifikator leistungsfähig und dennoch kompakt geblieben. Damit kann eine Verifikation auch auf heute herkömmlichen PCs ausgeführt werden.

Nicht unerwähnt soll in diesem Zusammenhang auch bleiben, dass SPS-Programme in der Regel aus der Sicht der Informatik in Struktur und Aufbau direkt und linear sind, ähnlich zu Assemblerprogrammen. Sie haben keine höheren Kontroll- und Ablaufstrukturen oder komplexe Datenstrukturen, sowie in den typischen Anwendungen für Maschinenprogramme keine Rückwärtssprünge. Die einzige besondere Eigenschaft ist die zyklische Ausführung, die aber in der Regel einfach beherrschbar ist und sogar, wie sich in der Folge herausstellen wird, für eine formale Verifikation nicht nur ohne wesentliche Bedeutung sind und diese sogar begünstigt.

Am folgenden Beispiel wird der Begriff Term, nunmehr bereits in der Sprechweise von PROLOG näher erläutert. Die in der Folge diskutierten Programm Konstrukte sind bereits Teile des Verifikators.

Ein Term ist eine allgemeine Form von Objekten, wie diese in PROLOG verwendet werden. Ein Term kann einfache oder strukturierte Objekte enthalten. Beispiel für einen im Verifikator verwendeten Term: **`input_vector([1, 0, 1, 1, 0], 3)`**. Dieser Term ist gleichzeitig ein Faktum, weil er ausschließlich Konstante (so genannte Terminalsymbole) und keine so genannten „don't cares" enthält. Später wird noch auf „offene" Symbole wie don't cares und deren Verwendung genauer eingegangen. Der gezeigte Term stellt einen Eingangsvektor für die Verifikation dar, präzieser ausgedrückt: kann als Eingangsvektor interpretiert werden.

Am folgenden Beispiel wird der Begriff Formel, Regel näher erläutert:

Eine Formel, in PROLOG synonym verwendbar ist der Begriff Regel, ist beweisbar. Sie enthält ableitbare Elemente (Variable) und Konstante. Beispiel für eine später verwendete Formel: **`line(3, u, 2, VE2, VE3)`**. Diese Formel ist der Ausdruck für eine Befehlszeile in SPS-Programmen. Ihre Bedeutung wird in [KP12] genauer diskutiert und ist zum besseren Verständnis hier nur kurz angeführt:

`line` Name des Befehls
`3` Nummer der Programmzeile
`u` PROLOG-Darstellung des UND-Befehls
`2` Position des zugehörigen Parameters im Eingabevektor
`VE2` Variable, die das bisherige Verknüpfungsergebnis enthält
`VE3` Variable, für das Verknüpfungsergebnis nach Ausführung des Befehls

Insbesondere den Variablen, wie im Beispiel **`VE2`** und **`VE3`**, werden erst zur Laufzeit der Verifikation Werte zugewiesen. Die Summe dieser Werte spiegelt die schrittweise Entwicklung des SPS-Programms in jeden Zyklus genau wieder.

Ableitbare Elemente werden von PROLOG bewiesen. Dabei bedeutet Ableiten von Aussagen, dass aus gegebenen Tatsachen (Fakten) und Formeln (Regeln) PROLOG Schlussfolgerungen zieht. Dabei sind die Tatsachen (Fakten) Aussagen über ein PROLOG Element, das als wahr gilt.

Die Elemente in der Prädikatenlogik müssen, im Gegensatz zu anderen Programmiersprachen, nicht explizit deklariert werden. Ein Beispiel für ein im Verifikator verwendetes PROLOG Element ist **`input_vector([1, 0, 1, 1, 0], 3)`**.

Dabei sind:
`input_vector` die syntaktisch korrekte Bezeichnung und
`([1, 0, 1, 1, 0], 3)` die für dieses Element wahre Aussage.

Erfolgt eine Anfrage zum Element **`input_vector`**, wird mit der vordefinierten Aussage geantwortet. Der in der Formel **`line(3, u, 2, VE2, VE3)`** angegebene Parameter **`2`** zeigt auf den Wert das zweite Element im **`input_vector`**, hier besetzt mit dem Wert **`0`** (bzw. den logischen Wert „**`FALSE`**“). Die Bedeutung in Bezug auf ein SPS-Programm wäre die logische „**`UND`**“ (**`u`**) Verknüpfung der Wahrheitswerte an der Stelle **`2`** des Eingangsvektors (hier steht das Ergebnis **`0`** gleichzusetzen mit den Wert logisch „**`FALSE`**“ bereits fest) mit dem Wahrheitswert von **`VE2`** (dem im unmittelbar vorhergehenden Schritt ermittelten Ergebnis) und die Ablage in **`VE3`**. Obwohl die hier hergestellte Assoziation der Wertefolge **`[1, 0, 1, 1, 0]`** zu entsprechenden Zuständen im Speicher einer SPS zusammen mit

der Wechselwirkung zu einer Anweisung im SPS-Programm sehr anschaulich dargestellt ist: PROLOG selbst kennt keine SPS und kann eine derartige Interpretation auch nicht treffen.

Am folgenden Beispiel sollen die definierten formalen Ausdrücke erläutert werden:

Im Verifikator [KP12] werden Terme sowohl als Variable als auch als Konstante genutzt. Beispielsweise werden einzelne Befehlszeilen in der Form **`line(3, u, 2, VE2, VE3)`** dargestellt. Ohne hier im Einzelnen auf die in [KP12] diskutierte Bedeutung einzugehen, sind die dabei in der Klammer angeführten Elemente die oben definierten Terme, wobei die Ausdrücke **`3`**, **`u`**, **`2`** Konstante darstellen, die vorher definiert und bezeichnet werden mussten, die Elemente **`VE2`**, **`VE3`** stellen Variable dar, die mit Werten instanziert und danach bewiesen werden.[64]

Der Ausdruck **`line(3, u, 2, VE2, VE3)`** stellt eine Formel dar. Innerhalb dieser Formel werden den Funktionssymbolen Elemente aus der Menge der Interpretation zugeordnet. In dem betrachteten Ausdruck stehen Elemente in einer Beziehung (im Beispiel **`u`**, **`2`**), die zusammen mit den Variablen (im Beispiel **`VE2`**, **`VE3`**) die Interpretation bestimmen, in diesem Fall gleichzusetzen mit dem Sinn dieser Befehlszeile.

Innerhalb dieser Formel werden Wahrheitswerte benutzt, die einerseits in vorhergehenden Beweisschritten bewiesen worden sind (im Beispiel **`VE2`**), andererseits im aktuellen Bearbeitungsschritt bewiesen werden und in späteren Beweisschritten weiterverwendet werden (im Beispiel **`VE3`**).

Zur Beweisführung für die Variablen (im Beispiel **`u`**, **`2`**, **`VE2`**) müssen Fakten und Regeln aufgestellt sein. **`VE2`** stellt in diesem Beweis die Vorbedingung dar, das Ergebnis, also die Nachbedingung, wird in **`VE3`** dargestellt. Im Beispiel bezeichnet **`2`** die Position einer Variablen in einer Liste.[65] Diese zu beweisen ist insoweit trivial, weil ihr Wert aus der vorgegeben Liste entnommen (instanziert) wird. Die Variable **`VE2`** wurde schon in vorhergehenden Schritten (hier genau im unmittelbar Vorhergehenden) bestimmt und ist deshalb bereits instanziert. Der Beweis wird über jene Variable geführt, die den so genannten Operator repräsentiert (im Beispiel **`u`**). Für den Operator existieren seinerseits Fakten und Regeln. Mit Hilfe dieser Fakten und Regeln wird der Zusammenhang **`u`**, **`2`**, **`VE2`** bewiesen mit dem Ergebnis des Beweises, die

[64] Ohne auf die Form der Schreibweise in PROLOG genauer einzugehen sind Zeichen und Zeichenkombinationen, die mit einer Ziffer oder einem Kleinbuchstaben beginnen immer Konstante (Literale in der Diktion von PROLOG), Zeichen und Zeichenkombinationen, die mit einem Großbuchstaben beginnen immer Variable.

[65] Nebenbei bemerkt: diese Liste selbst zählt ebenfalls zu den Fakten. Die hier genannte Liste wurde als Eingangsvektor bezeichnet.

in der Variablen **VE3** instanziert worden ist, zur weiteren Verwendung für andere Beweisschritte bereitgestellt. Konnte **VE3** instanziert werden, existiert die Nachbedingung und damit konnte der Beweis über die hier diskutierte Anweisung **line(3, u, 2, VE2, VE3)** erfolgreich geführt werden.

Zum obigen Beispiel ist in diesem Sinn ergänzend anzumerken:

Es wurde der Begriff der Interpretation eingeführt. Die Beweisführung in der Prädikatenlogik kennt nur zwei Zustände: beweisbar, gleichzusetzen mit wahr, wenn eine entsprechende Abbildung eines Elements aus der Menge der in der Interpretation[66] enthaltenen Elemente möglich ist, und nicht beweisbar, gleichzusetzen mit falsch, in allen anderen Fällen.

Die Beweisführung wurde im dem oben zugrunde gelegten Beispiel bereits erläutert. Für den Verifikator ist es jedoch erforderlich, dass alle vorkommenden Befehle der SPS interpretierbar und damit beweisbar sind. Dazu muss es für jeden Befehl einen Satz von Tatsachen (Fakten) und Behandlungsvorschriften (Regeln) geben, die diese Beweisführung ermöglichen. Die Summe dieser Fakten und Regeln ist in der Theorie zusammengefasst. Dargestellt sind die Fakten und Regel in Formeln, die zusammengenommen die Wissensbasis, im vorliegenden Fall für den Verifikator bilden. Damit bildet die Wissensbasis die Theorie für die zu lösende Aufgabenstellung.

Die typische Arbeitsweise einer SPS, zuerst die Eingangszustände zu lesen und für die Dauer eines Programmdurchlaufs zwischenzuspeichern, danach das vom Anwender erstellte Programm abzuarbeiten, um zuletzt die daraus ermittelten Ausgangzustände auszugeben, begünstigt das Formulieren der für die Verifikation benötigten Bedingungen[67]. Durch die Anwendung der Methode der induktiven Assertionen werden die so genannten Vorbedingungen für die Verifikation benötigt, die, angewendet auf das Programm, zu den Nachbedingungen führen (vgl. [KP12]).

---

[66] Die Interpretation ordnet aus der Sichtweise der formalen Logik gesehen jedem syntaktisch korrekten Konstrukt einer formalen Sprache seine Bedeutung zu. Für das hier diskutierte Verifikationsmodell ist diese Sichtweise relevant. Im Sprachgebrauch hingegen wird Interpretation als die Auslegung eines Sachverhalts verstanden. Während im erst genannten Fall streng mathematische Zusammenhänge beschrieben werden, die eindeutig sind, könnte es im zweiten Fall auch mehrere Auslegungen geben.

[67] Auch die beispielsweise in den SPS-Familien von Siemens vor einigen Jahren eingeführte Verschiebung des Ablaufes, zuerst die Ausgangszustände der vorhergehenden Berechnung auszugeben, danach Eingänge zu lesen und erst dann das Programm zu bearbeiten, um im Folgezyklus die vorher Ausgangszustände bereitzustellen, ist nur eine Verschiebung der Grenzen. Die Verschiebung wurde getroffen, um beim Neuanlauf einerseits zuerst die über die Dauer einer Abschaltung gespeicherten Ausgangszustände ausgeben zu können, andererseits wegen der Initialisierung.

SPS und Verifikator

Bereits in der Einführung wurde der Begriff eines Pfades erwähnt. Damit ist der Weg, gleichzusetzen mit dem Programmfortschritt bei der Programmbearbeitung, entlang der Befehlskette assoziiert. Als Maschine, die von einem Zustand nach einem Programmschritt in den Folgezustand wechselt, werden im Gegensatz zu anderen Computerprogrammen immer die gleichen Befehlsfolgen durchlaufen. Mögliche Verzweigungen können sich nur auf Grund von Sprungbefehlen ergeben. Im Falle der hier betrachteten sicherheitsgerichteten SPS-Programme sind jedoch nur vorwärts gerichtete Sprünge zulässig, wodurch sich die Befehlsfolge durch Auslassungen besten Falls nur verkürzen kann.

Der Pfad wird durch seine Statusinformationen beschrieben. Dieser Pfad ist im Modell der SPS durch seine Konfiguration K beschrieben. Die Konfiguration enthält die vollständige Information in Bezug auf einen kompletten SPS-Zyklus, in Einzelnen das komplette SPS-Programm und die einem Zyklus zuordenbaren Zustände.

Der Verifikator entwickelt bei einem Programmdurchlauf eine Historie durch die Instanzierung der Zwischenergebnisse. Die instanzierten Werte stammen einerseits aus dem Eingangsvektor und andererseits aus dem Berechnungsfortschritt. Gegeben sei ein SPS-Programm, das durch mehrere Netzwerke beschrieben ist. Die Anzahl der Kommandos K in den Netzwerken seien mit α, β, … bezeichnet. Dann werden vom Verifikator die Werte als Verknüpfungsergebnisse (**VE**) im Programmfortschritt des SPS-Programms der Reihe nach entwickelt. Zuletzt existiert eine Folge:

```
<  VE1,      VE2,…      VEα,
   VEα+1,    VEα+2,…    VEα+β,
   VEα+β+1,  VEα+β+2,…  usw.     >
```

Diese Folge ist eine Folge von Variablenwerten, die für die Dauer einer Berechnung Gültigkeit besitzen. Diese Folge ist hier gleichzusetzen mit den Zuständen aller Berechnungsschritte. Die in einem Netzwerk an erster Stelle stehenden Kommandos sind ausnahmslos Abfragen und stammen daher immer aus Informationsinhalten des Eingangsvektors. Sie sind im oben angeführten Beispiel hervorgehoben.

Im erweiterten Modell existieren neben den Variablen V Zustände Z. Mit den Zuständen sind im weiter oben definierten Modell jene Werte und Wertzuweisungen zu verstehen, wie diese etwa in einem Akkumulator eines Computersystems auftreten. In diesem Sinn ist im Modell

der SPS jedem Kommando ein eigener Akkumulator zugeordnet. Bestimmte Zustände zeichnen sich dadurch aus, dass sie aus dem Eingangsvektor und in weiterer Folge aus dem Eingangsabbild stammen. Als Variable wird dort die Herkunft der Zustandsinformationen bezeichnet, also der Eingangsvektor bzw. sein temporäres Abbild.

Der wesentliche Unterschied SPS-Modell zum Verifikator ist, dass beim Verifikator der jeweils älteste Wert eines Zustands vom zuletzt ermittelten Wert dieses Zustands überschrieben wird. Im SPS-Modell bleiben auch ältere Werte erhalten.

Diskussion

An dieser Stelle kann gefragt werden, welcher Mehrwert aus der Kenntnis des vollständigen SPS-Modells gezogen werden kann. Die zusätzlichen Werte sind für die vertiefte Analyse interessant, haben aber an dieser Stelle keine praktische Bedeutung.

Anpassungen

In der IEC 61131 Teil 3 sind SPS-Programmsprachkonzepte definiert. Nahezu alle Hersteller von SPS, die gleichzeitig auch die zugehörige Programmierumgebung ihren Nutzern bereitstellen, halten sich an diese Definitionen. Dennoch gibt es Unterschiede in der Auslegung, die sich auf die Programmerstellung auswirken.

Soll die Verifikation für SPS-Programme durchgeführt werden, müssen bereits bei den vorbereitenden Schritten zur Verifikation derartige Unterscheidungen in Schreibweise und Bezeichnung der Kommandos (Operatoren und Operanden) bekannt sein. Eine Zuordnung von SPS-Befehlen zu entsprechenden Befehl im Verifikator muss bereits im Vorfeld getroffen werden. Ein typisches Beispiel für Varianten in der Bezeichnung ist der Ladebefehl in den Schreibweisen **„L“** bzw. **„LD“**.

Eine weitere Fragestellung betrifft den Befehlssatz bzw. jene Teilmenge der Kommandos aus dem Befehlssatz einer SPS, die dem Verifikator bekannt sind. Im Verifikator in [KP12] ist zunächst nur eine Teilmenge, die wichtigsten Grundbefehle einer SPS, implementiert worden. Entsprechende Erweiterungen können dort noch notwendig werden.

Anpassungen und Erweiterungen sind sowohl beim Verifikator selbst als auch bei den zugehörigen Analyse und Hilfsprogrammen notwendig. Aus Effizienzgründen ist die Formalisierung des Quellprogramms in Fakten und Formeln für den Verifikator in Hilfsprogramme ausgelagert.

Vorgehensweise

Die Vorgehensweise bei der Verifikation eines SPS-Programms mit dem Verfikator nach [KP12] erfordert einige Arbeitsschritte, die in der Folge zusammengefasst dargestellt sind. Ausgangspunkt ist ein vom Programmierer entwickeltes SPS-Programm. Da Editoren handelsüblicher SPS meist den Quelltext nicht in der benötigten Form zur Verfügung stellen, müssen zuerst vorbereitende Schritte gesetzt werden.

Zu Beginn werden Funktionen des Programmeditors des SPS-Programms genutzt. Es wird ein Programmausdruck des zu untersuchenden SPS-Programms angefertigt und auf eine Datei umgeleitet. Sie enthält den Quell-Code für die nachfolgende Untersuchung. Ein dabei entstandenes List-File muss insoweit nachbearbeitet werden, weil typische Programmausdrucke von Dokumentationen zusätzliche Informationen enthalten, wie Überschriften, Netzwerkbezeichnungen und Nummerierungen, Seitenbezeichnungen, Programmkommentare, symbolische Adressierungen und andere. Da vielfach unterschiedliche SPS-Sprachkonzepte durch das Umschalten von SPS-Programmeditoren ermöglicht werden, muss entsprechend vorgenommenen Grundeinstellungen des Programmiersystems der Programmausdruck in AWL erfolgen. Es sind nicht nur die Sprachkonzepte allein zu beachten, sondern auch die im Quellprogramm verwendete Art der Adressierung. Symbolische Adressen kann der Verifikator nicht verarbeiten, daher müssen diese durch die entsprechenden Eingangs- bzw. Ausgangsadressen ersetzt werden. In der Regel kann dies bereits im Programmeditor der SPS vorgewählt werden.

Je nach den Möglichkeiten, die für derartige Dokumentationsfunktionen vom Hersteller der SPS bereits vorgesehen worden sind, muss eine für die untersuchte SPS zugeschnittene Vorverarbeitung stattfinden. Diese Aufgabe kann automatisiert durch ein Hilfsprogramm erfolgen, das bereits auch Anpassungen, wie diese vorher diskutiert worden sind, vornehmen kann.

Der eigentliche Verifikationsschritt beginnt mit dem Start von PROLOG. Das bereits in PROLOG übertragene Programm wird mit dem Befehl „**`consult([Name des Programms]).`**“ geladen. Die benötigten zulässigen Eingangs- und Ausgangsvektoren müssen eingegeben werden. Mit dem Befehl „**`verify_all1.`**“ wird der Verifikationsvorgang durchgeführt. Dabei werden einerseits alle möglichen Eingangsvektoren auf die zugehörigen Ausgangsvektoren abgebildet, andererseits deren Korrektheit festgestellt (vgl. [KP12]).

Darstellung von Ergebnissen

Werden zunächst keine Vorgaben in Bezug auf die Vektoren getroffen, erzeugt der Verifikator die benötigten Permutationen der Eingangsvektoren selbst. Danach wird für jeden so erzeugten Eingangsvektor der passende Ausgangsvektor erzeugt und verifiziert. Im Fall fehlender Vorgaben ist die Verifikation in jedem Fall erfolgreich, weil ein mit „don't cares" bereitgestellte Ausgangsvektor immer erfüllbar ist. Es ist sofort klar, dass in diesem Fall nachträglich festgestellt werden muss, ob der derart erzeugte Ausgangsvektor der Aufgabenstellung entspricht oder nicht.[68]

Betrachtet wird als einfaches Beispiel ein triviales SPS-Programm, in dem zwei Eingänge (**E1, E2**) logisch **UND** verknüpft sind. Der Befehl „**verify_all1.**" liefert als Ergebnis:

```
 E1  E2  A        E1  E2  A
[0, 0, 0] ==> [0, 0, 0]  Klasse 1
[0, 0, 1] ==> [0, 0, 0]  Klasse 1
[0, 1, 0] ==> [0, 1, 0]  Klasse 1
[0, 1, 1] ==> [0, 1, 0]  Klasse 1
[1, 0, 0] ==> [1, 0, 0]  Klasse 1
[1, 0, 1] ==> [1, 0, 0]  Klasse 1
[1, 1, 0] ==> [1, 1, 1]  Klasse 1
[1, 1, 1] ==> [1, 1, 1]  Klasse 1.
Yes
```

Der Verifikator hat zuerst im Eingangsvektor die don't cares durch **0** oder **1** ersetzt und entsprechende Kopien der Variationen vorbereitet. Die angegebene Klasse resultiert aus der Vorgabe von AWLPRO. Für jeden Programmdurchlauf steht ein Eingangsvektor bereit, die der Reihe nach verifiziert werden.

Anstelle der Verifikation des gesamten Programms können auch Tupel von Vektoren verifiziert werden. Die Eingabe „**verify([Eingangsvektor], [Ausgangsvektor])**." liefert entweder die Antwort **Yes** bei erfolgreicher Verifikation, sonst **No**. [69]

---

[68] Aus einem ursprünglich mit „don't cares" bereitgestellten Eingangsvektor wird die Menge aller möglichen Eingangsvektoren erzeugt. Dies entspricht, bezogen auf ein reales System, die Untersuchung des Programms auf alle möglichen Eingangskombinationen, also aller möglichen Testfälle.

[69] Der Verifikator löst das SPS-Programm formal und gibt die Lösungen im Ausgangsvektor an. Weil der Ausgangsvektor beliebige Zustände annehmen durfte, ist auch die Verifikation erfolgreich. Der Verifikator zeigt das erzeugte Abbild des Ausgangsvektors in der Form **[Eingangsvektor] ==> [Ausgangsvektor]** an. Wird ein reales System betrachtet, kann diese Abbildung mit „wie hätte das System auf die repräsentierte(n) Eingabe(n) reagiert" umschrieben werden. So kann

Emulation als Teilverifikationsschritt

In vielen Arbeiten wird bereits von erfolgreicher Verifikation eines SPS-Programms gesprochen, wenn vergleichsweise kurze oder Programme mit geringer Komplexität verifiziert worden sind. Die Verifikation eines Programms, das sich über mehrere Netzwerke erstreckt, bedeutet, dass schnell eine Grenze erreicht wird, bei dem die Anzahl der zeitgleich zu betrachtenden Variablen einerseits für die eingesetzten Werkzeuge zu groß wird, andererseits das Erstellen der notwendigen Bedingungen etwa zur Bestimmung der Ein- und Ausgangsvektoren zu aufwendig erscheint.

Definitionsgemäß müssen bereits vor der Verifikation die Tupel Eingangsvektor - Ausgangsvektor bekannt sein. Dies ist bei einer relativ geringen Komplexität etwa auf Netzwerkebene noch vergleichsweise einfach handhabbar.

Die Definition von Eingangs- und Ausgangsvektoren ist mitunter erschwert, besonders dann, wenn ein SPS-Programm eine große Anzahl von Ein- und Ausgängen nutzt. Die Möglichkeit, das Programm in Funktionsgruppen zu modularisieren, ist eine Möglichkeit diese Anzahl zu verringern. Die Grundlage zur Segmentierung erschließt sich aus der Programmanalyse und wurde bereits genauer diskutiert. Im Folgenden wird die Möglichkeit untersucht, nur die Eingangsvektoren festzulegen, um die Reaktion des SPS-System auf diese festzustellen.

Eine SPS fungiert als eine Maschine zur Erzeugung von Zustandsübergängen nach bestimmten Vorschriften, dem SPS-Programm. Bei der Verifikation wird gefragt, ob die Erzeugenden, im gegeben Fall die Eingänge, immer die korrekte Ausgangsinformation liefern. Diese Ausgangsinformation steht aber erst im letzten Arbeitsschritt vor dem finalen Vergleichsschritt zur Verfügung. Der hier diskutierte Vorschlag lautet, die Verifikation genau vor dem letzten Schritt zunächst zu unterbrechen, mit der Fragestellung, was kann auf diese Art und Weise erreicht werden.

Die Antwort ist sehr einfach: mit dem Bereitstellen des Eingangsvektors kann der Verifikator bereits arbeiten. Er wird im Ablauf des SPS-Programms solange fortschreiten, bis sein Ende erreicht ist. Der abschließende Vergleich mit dem Ausgangsvektor kann auf Grund des Fehlens dieser Information nicht durchgeführt werden.

---

jeder erzeugte Ausgangsvektor abgelesen und verglichen werden. Ob die Annahme richtig ist, dass das nachträgliche Feststellen der Korrektheit einfacher als das Bestimmen der Abbilder **`[Eingangsvektor] ==> [Ausgangsvektor]`** ist, wird die Praxis zeigen. Begründet wird diese Annahme durch den Umstand, dass mögliche Fehlfunktionen, die durch den Programmablauf bedingt sind, auf Grund der Aufgabenstellung schnell erkannt werden können.

Dennoch hat der Verifikator bis zu diesem Schritt die Entwicklung des Ausgangsvektors bereits berechnet, er hat praktisch mit der gegebenen Eingangsinformation das gesamte SPS-Programm emuliert. Bis hierher kann der Ablauf automatisiert erfolgen. Die errechneten Ergebnisse können ausgegeben werden.

Mit anderen Worten ausgedrückt: der Verifikator hat auf die Anfrage, welche Reaktion erzeugt der untersuchte Eingangsvektor mit dem passenden Ausgangsvektor geantwortet.

Auf Grund der Eindeutigkeit des Vorgangs, wird auch bei wiederholter Anfrage immer das zugehörige Endergebnis erzeugt werden. Verallgemeinert gilt, dass dieser Vorgang auch für alle anderen Eingangsvektoren durchgeführt werden kann.

Diskussion

Im Normalfall wird bei der Verifikation geprüft, ob eine vorher definierte Belegung durch Wahrheitswerte für den Ausgangsvektor erreicht wird oder nicht. Grundsätzlich wird die Bestimmung vor dem Verifikationsschritt durchgeführt. Es ist zumindest ein Schritt zur Programmüberprüfung, mögliche Reaktionen des SPS-Programms auf Grund der Ausgangsbelegung durch die Emulation vor der eigentlichen Inbetriebnahme einer Anlage bestimmen zu lassen. Im weitesten Sinn kann eine derartige Vorgehensweise als gezieltes Testen von Funktionalitäten in Form der Simulation von Betriebszuständen einer Anwendung gesehen werden.

Eine weitere Möglichkeit, diese Option zu nutzen, ist z.B. mit Hilfe eines Generators alle Eingangsvektoren bestimmen zu lassen um daraus alle durch das Programm erzeugbaren Ausgangsbelegungen zu gewinnen. Genutzt werden können diese Daten in zweierlei Hinsicht: kritische oder sich ausschließende Zustände, die vorher definiert worden sind, können durch Filtermechanismen lokalisiert werden. Durch entsprechende Maßnahmen im SPS-Programm sind erkannte Schwächen oder Fehler behebbar. Weiters kann durch Sortierung die Zuordnung, welche Eingangsvektoren zu bestimmten Ausgangsvektoren geführt haben, die Klassifizierung automatisch vorgenommen werden. Damit ist die weiter oben diskutierte Simulation auch auf zufällige Betriebszustände eines Systems ausweitbar und mit dem entsprechenden Aufwand bis hin zur Simulation aller Zustände.

Wenn es auch rein formal gesehen nicht als Verifikation im herkömmlichen Sinn verstanden werden darf: das Endergebnis, ein überprüftes SPS-Programm, steht am Ende dieser Vorgehensweise ebenfalls zur Verfügung.

Ergänzend ist dazu anzumerken, dass diese Vorgehensweise dem Testen eines SPS-Programms nahekommt. Definitionsgemäß wird beim Testen das Verhalten von Eingangs- und

Ausgangspaaren (die weiter oben genannten Tupel der Ein- uns Ausgangsvektoren) überprüft. Dazu werden Eingangsfolgen erzeugt und beobachtet, ob das Verhalten eines untersuchten Systems den Spezifikationen entspricht. Als Möglichkeiten wird dabei unterschieden, die Tests an der realen Steuerung durchzuführen oder Teststellungen simulativ an einem Rechner durchzuführen.

Bei der Simulation wird mit Hilfe eines Modells das Verhalten einer Steuerstrecke untersucht, oder z.B. mit Hilfe des Verifikators automatisch oder teilautomatisch durch vorgegebene Eingangskonfigurationen das Verhalten der Steuerung überprüft.

Duale Arbeit Verifikator - SPS

Verifikation unter Einbeziehen der Hardware einer SPS könnte durch folgende Anordnung erfolgen. Fig. 48 zeigt das Prinzip. Die Hardware mit dem laufenden SPS-Programm wird in den Verifikationsprozess mit einbezogen. Der Datenfluss ist durch die Pfeile symbolisiert. Der Zähler gibt dabei die Eingangsvektoren vor, die einerseits dem Verifikator zugeführt werden, andererseits als Eingangssignale der SPS zur Verfügung gestellt werden. Durch Vergleich der Ausgangsvektoren können Unterschiede festgestellt werden und weitere Untersuchungen vorgenommen werden.

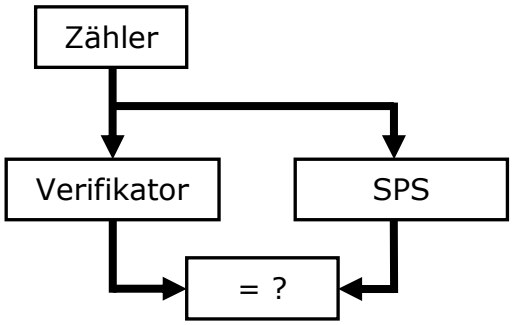


Fig. 48: Anordnung, Prinzip

Der als Signalgenerator vorgesehene Zähler besitzt Ausgänge für die SPS. Diese erhält dabei vom Zähler nur jene Eingangsinformation, die tatsächlich an die Hardware angeschlossen werden können. Zu jeder Eingangsbelegung variiert der Signalgenerator die im Eingangsvektor enthaltenen Ausgangsinformationen. Diese werden dem Verifikator als Startbedingung zugeführt und die Ergebnisse mit jener der SPS in Relation gesetzt.

Programmanalyse und Verifikation

Aus der Sicht der Bedienung eines Computers können Programme in solche, die transformationell und solche, die reaktiv arbeiten eingeteilt werden (vgl. [HP85]). Eine SPS arbeitet re-

aktiv in Kombination des SPS-Programms und seiner angeschlossenen Hardware. Bezogen auf einzelne Programmzyklen ist Programmverhalten transformationell. Aus diesem Grund werden in dieser Arbeit die Programme als Transformationen von Statusinformationen betrachtet, genauer alle möglichen Transformationen der Statusinformationen von Anfangszuständen in die zugehörigen Endzustände. Aufgrund der Tatsache, dass eine SPS nur eine einzelne Transformation von Eingaben in Ausgaben innerhalb eines einzelnen Programmzyklus ermöglicht, können diese Transformationen unabhängig berechnet werden. Damit kann eine SPS als ein temporär statisches System für einzelne Zyklen betrachtet werden.

Verifikation

Die SPS selbst als abstrakte Interpretation nutzt die Hauptgruppen Organisationssystem, Ein-Ausgabesystem und die Bearbeitung des Anwenderprogramms. Das Organisationssystem ist vergleichbar mit einem Betriebssystem in einem Rechner. Es sorgt für die zyklische Bearbeitung, Kommunikation zu weiteren Komponenten der SPS und zur Ansteuerung der physikalischen Ein- und Ausgänge des Ein- Ausgabesystems.

Durch das Organisationssystem werden die Variablen der Ein- und Ausgänge zyklisch bereitgestellt und dem SPS-Programm zur Verfügung gestellt. Die als Ein- bzw. Ausgangsvektoren bezeichneten Abbilder der Zustände sind temporär statisch, deshalb wird in der Folge von einem temporär statischen System gesprochen, bei dem die Eingangsvektoren die Prämissen für einen Verifikationsdurchlauf bilden.

Die Verifikation ist aus der Sicht eines temporär statischen Systems die Zusammenfassung von überprüften einzelnen Zyklen, womit das Programmverhalten des gesamten reaktiven Systems überprüft wird. Ein Programm ist verifiziert, wenn zu einer Spezifikation die Semantik des SPS-Programms die Spezifikation erfüllt. Im Falle der Nichterfüllung liefert der Analysator Informationen, um die Ursachen für mögliche Fehler darzustellen.

Der in [KP12] vorgestellte Verifikator untersucht als eine abstrakte Interpretation ein in Prädikatenlogikformeln transformiertes SPS-Programm. Die abstrakte Interpretation der Spezifikation ist durch die internen und externen Zustände der Programmelemente gegeben.[70]

[70] Obwohl theoretisch der Verifikator auch komplexere Programme verifizieren kann und die Anzahl von Variablen nicht im Voraus beschränkt ist, sind seiner Nutzbarkeit durch die Limitierung auf noch überblickbare Variationen der Variablen Grenzen gesetzt. Testläufe mit einen Testprogramm mit bis zu ca. 14 Variablen wurden beschrieben (vgl. [KP12]). Das in dieser Arbeit verfolgte Konzept der Segmentierung bricht diese Grenzen auf.

Der Verifikator ist ein Programm, das auf einem Computer ausführbar ist. Bezüglich seiner Komplexität kann festgestellt werden, dass bei ***n*** Bit-Variablen (Eingänge, Ausgänge, Merker) 2^n Berechnungsschritte durchgeführt werden müssen. Daraus resultiert das bekannte „*state explosion problem*“, das es zu mildern gilt. Aus diesem Grund wurden die Überlegungen zur Zerlegung des SPS-Programms in kleine Einheiten, funktionelle Abschnitte, vorgenommen.

Die angewendete Technik der Verifikation vergleicht eine formale Programmbeschreibung bzw. die formale Spezifikation eines Programms mit einem formalisierten SPS-Programm. Die formale Programmspezifikation erfolgt in mathematischer Schreibweise. Damit werden auf präzise Art und Weise das System und sein Verhalten beschrieben. Die beste Möglichkeit für formale Spezifikationen ist die Verwendung boolescher Ausdrücke. Die Spezifikation für [*I, Q*] [71] sagt informell: Wenn *I* erfüllt ist und der Programmdurchlauf erfolgt ist, dann ist *Q* erfüllt.

In der Folge wird Termination eines Programms als die Termination aller möglichen Programmdurchläufe mit unterschiedlichen Eingangskonfigurationen definiert. Sei Termination für einen Programmdurchlauf gegeben und eine Boolesche Variable h für „halt“ eingeführt, kann die Spezifikation als einfacher Boolescher Ausdruck $I \Rightarrow h \wedge Q$ angegeben werden. Damit wird eine einzelne Transformation der Statusinformationen in einem Zyklus des SPS-Programms beschrieben. I definiert dabei eine bestimmte Eingangskonfiguration und Q die zugehörige Ausgangskonfiguration. Dabei seien $I_{0,1...n}$ mit $I_0 \neq I_1 \neq ...I_n$ eine Gruppe verschiedener Eingangskonfigurationen mit Q als der gemeinsamen zugehörigen Ausgangskonfiguration.

Im Detail wurde diskutiert, dass diese Darstellung durch die Bildung von so genannten Klassen weiter präzisiert wird. Bei der Bildung von Verifikationsbedingungen wird durch die Klassifizierung eine stärkere Zuordnung von Vor- und Nachbedingungen erreicht. In Bezug auf die Vorbedingung für die spätere Verifikation bedeutet dies eine Stärkung, weil dann in Abhängigkeit von jeder betrachteten Klasse c gelten muss: $I_{0,1...n}(c) \Rightarrow h \wedge Q(c)$.

Aussage: Während genau eines Programmzyklus einer SPS existiert nur genau ein Zustandsübergang.
Begründung: Vor dem Start des Programms werden die Eingangsinformationen ausgelesen und eingefroren. Deshalb ist vom Beginn des Zyklus weg die zugehörige Ausgangsinforma-

[71] In der internationalen Schreibweise werden anstelle von *E* für Eingänge bzw. *A* für Ausgänge *I* für Inputs und *Q* für Outputs verwendet. Die eher ungewöhnliche Bezeichnung *Q* anstelle von *O* soll die Verwechslung mit der Ziffer 0 verhindern.

tion bereits gegeben. Dies schließt ein, dass für eine bestimmte Eingangskonfiguration (Eingangsvektor) nur eine zugehörige Ausgangsinformation (Ausgangsvektor) existiert. □

Aussage: Alle Zustandsübergänge sind voneinander unabhängig.
Begründung: Da für jeden Zyklus nur ein Zustandsübergang existiert, sind Zustandsübergänge umkehrbar eindeutig in Bezug auf einen bestimmten Zyklus. Daher kann jeder Zustandsübergang unabhängig von anderen betrachtet werden. □

Diskussion

Fig. 49 zeigt das Prinzip der SPS-Funktion für den Fall korrekter Programme. Eingangszustände bewirken vordefinierte Ausgangszustände nach der Ausführung des Programms. Sprungbefehle z.B. überspringen Teile des Programms während eines Zyklus und verringern so nur die Anzahl der erforderlichen Zwischenbefehlsschritte. Unterschiedliche Eingangszustände können in dieselben Ausgangszustände führen. Unterschiedliche Eingangszustände, die in einen Zyklus zum selben Ausgangszustand führen, gehören zu bestimmten Klassen von Zuständen.

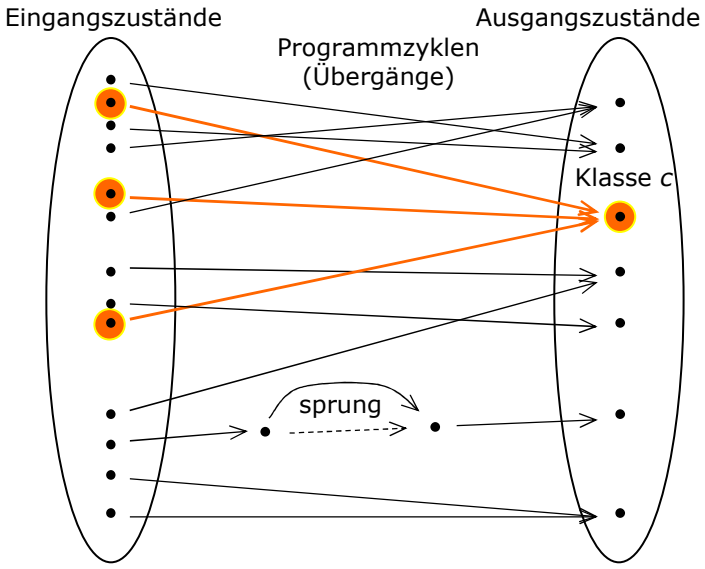


Fig. 49: Prinzip der SPS-Operation

In Fig. 49 sind keine Eingangs- und Ausgangszustände enthalten, die zu Fehlfunktionen der SPS führen. Ausgangszustände für Fehlfunktionen dürfen niemals erreicht werden. Ausgangszustände, die besonders in sicherheitsrelevanten Anwendungen auftreten, können in einer eigenen Klasse c_{malf} zusammengefasst werden. Diese Art von Klasse fasst programmtechnisch vordefinierte Fehlfunktionen (malfunction) bzw. Fehlreaktionen eines Systems zu-

sammen und könnte in SPS-Programmen dazu genutzt werden, automatisch Notabschaltungen auszulösen.

Da die einzelnen Übergänge der Programmzustände voneinander unabhängig sind, kann für die Verifikation eine beliebige Reihenfolge herangezogen werden.

Formalisierung des SPS-Programms

Ein SPS-Programm wird nach festen Regeln ausgeführt. Die eigentliche Programmabarbeitung erfolgt Schritt für Schritt. Anweisungen werden nach ihrer Position im SPS-Programm „bottom down“ abgearbeitet. Beginnend mit einem Test des Zustandsbits der ersten Anweisung beginnt das Programm Zwischenergebnisse zu erzeugen, und fährt damit fort, solange Zuweisungen an Ausgänge erfolgen. Mit der letzten Zuweisung wird der Programmzyklus beendet. Die Beendigung eines Programmzyklus erzeugt ein Übertragen der Ausgangsinformationen an die Ausgänge.

Aussage: jeder Programmzyklus ist umkehrbar eindeutig.
Begründung: Ein SPS-Programm wird innerhalb eines Zyklus schrittweise ausgeführt. Wenn Sprungbefehle zulässig sind, existieren verschiedene Pfade durch das Programm. Zu Beginn eines Zyklus sind jedoch bereits alle Bedingungen für die Abarbeitung festgelegt. Daher sind die Sprungbedingungen bekannt und in diesem Zyklus unveränderlich. Daraus folgt, dass der Pfad für die Programmausführung für einen gegebenen Zyklus umkehrbar eindeutig ist. Daraus folgt unmittelbar, dass nur eine Definition von Zustandsübergängen des SPS-Programms pro Zyklus möglich ist. □

Diskussion

Prinzipiell kann jede SPS-Programmiersprache oder ein entsprechender Dialekt formalisiert werden. Für die Behandlung der berechneten Werte ist die Reihenfolge der Anweisungen von Bedeutung. In Anweisungsliste kann diese Reihenfolge direkt ermittelt werden. Normalerweise kann bei fast allen SPS-Systemen ein Programm ohne weitere Schwierigkeit in eine Anweisungsliste überführt werden. Jede AWL-Anweisung folgt einem starren Schema. Sie besteht aus einem Anweisungsteil und der Adresse eines Operanden. Zu Beginn der Ausführung eines Befehls wird ein Anfangswert entweder als Ergebnis der vorherigen Anweisung oder aus einem Zwischenergebnis geholt. Nach Ausführung der Anweisung wird ein neues Ergebnis erzeugt. Die Ausführung jedes Zyklus und daher des SPS-Programms für eine bestimmte vorgegebene Interaktion mit seiner Umgebung ist eine Folge von Zuständen. Diese

Zustände werden in diskreten Zeitintervallen beobachtet, der so genannten Zykluszeit [72], beginnend von einem Initialzustand und danach weiter von Zustand zu Zustand, indem atomare Programmschritte oder Transitionen ausgeführt werden und enden in einem regulären letzten oder im schlimmsten Fall fehlerhaften Zustand.

In einem SPS-Programm existiert nur ein bestimmter letzter Schritt. Der letzte Schritt erzeugt einen Endzustand. Indem das SPS-Programm Anweisung für Anweisung ausgeführt wird, führt jeder Anfangszustand in einen zugehörigen Endzustand. Anweisungen sind die atomaren Operationen eines SPS-Programms. Ein SPS-Programm wird formalisiert, indem seine Anweisungen formalisiert werden.

Genauer gesagt wird für die Formalisierung von Anweisungen benötigt:
- ihre Position im Programm,
- Rechenregeln,
- Werte der benötigten Operanden,
- Zwischenergebnisse vor der Ausführung der Anweisung und
- schließlich das Ergebnis der Berechnung.

Für die Formalisierung von SPS-Programmen in Prädikatenlogik erster Stufe (PL1) [73] wird die Befehlsfolge in Formeln umgewandelt und mit Zeilennummern versehen. Die Positionen der Formeln entsprechen dabei genau der Befehlsfolge im SPS-Programm. Für die einzelnen Anweisungen werden Rechenregeln abhängig von SPS-Befehlen verwendet. Zeiger in Wertearrays bezeichnen die Werte der aufgerufenen Operanden, Zwischenergebnisse vor der Ausführung der Anweisung und das Ergebnis der Berechnung nach der Ausführung der Anweisung.

AWL-Programme werden als nummerierte Listen von Programmzeilen dargestellt. Jede Zeile beschreibt eine Funktion, die Variable in Werte überführt. Eine Programmausführung kann als endliche Folge von Programmzuständen betrachtet werden, die immer wieder wiederholt wird. Solche endlichen Folgen werden unabhängig betrachtet, als Übergange (*transfer)* $\tau = t_0\ t_1\ .....\ t_n\ .\ t_0$ oder Übergangsschritte bezeichnet. Ein Übergang geht von einem Anfangszustand des Programms aus und jeder Folgezustand ergibt sich aus der Ausführung einer einzelnen atomaren Aktion des vorherigen Zustands.

---

[72] Als Zykluszeit wird die Dauer eines kompletten Programmdurchlaufes bezeichnet, einschließlich des Lesens und Schreibens der Statusinformationen. Eine weitere kurze Zeitspanne wird für Überprüfungen und Verwaltung z.B. der Generierung des Zeittakts benötigt. Die typische Zykluszeit einer SPS beträgt in Abhängigkeit von der Länge des Anwenderprogramms einige Millisekunden.

[73] Prädikatenlogik erster Stufe (PL1, first-order logic)

Ein Beispiel für eine Anweisung in PL1:

line (line_nb , plc_command, pos, temp_nb-1, temp_nb)

Diese Formel ist der Ausdruck einer Anweisung in einem SPS-Programm. **line_nb** (Zeilennummer) ist dabei die Position der SPS-Anweisung im SPS-Programm, **plc_command** (die Repräsentation einer Berechnungsregel) die spezielle Anweisung an dieser Position, **pos** (Position) zeigt auf den Parameter im aktuellen Eingangszwischenvektor, der verwendet wird, **temp_nb-1** (Zwischenergebnis) das vorläufige Zwischenergebnis der vorigen Anweisung, **temp_nb** das vorläufige Zwischenergebnis nach Ausführung der aktuellen Anweisung. Dieses Ergebnis wird für die weitere Berechnung in der nächsten Anweisung verwendet bzw. im Ausgangsvektor abgelegt (Fig. 50). Ableitbare Elemente werden bewiesen. Ableiten aus den Zeilenanweisungen bedeutet, Schlüsse aus gegebenen Fakten (facts) und Formeln (rules) zu ziehen. Die Fakten sind Aussagen über PL1 Elemente, die als wahr betrachtet werden.

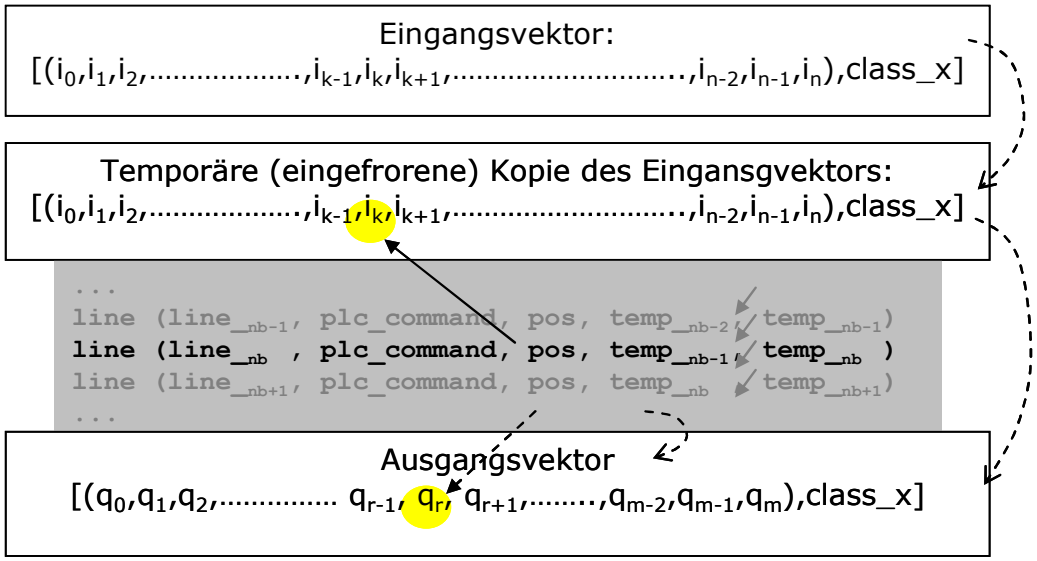


Fig. 50: Prinzip der Ausführung eines einzelnen Verifikationsschrittes

Eine terminierende Ausführung einer unendlichen Sequenz wird erreicht, indem der Endzustand wiederholt wird. In dieser Betrachtung einzelner Schritte gilt daher als Termination für jeden der Übergänge das Erreichen des zugehörigen Endzustandes zu einem gegebenen Anfangszustand, der als Initialwert fungiert.

Sei $I_k^{\sim}$ eine Kopie der Initialwerte, genannt Eingangsvektor I_k. Dann bewirkt ein Eingangsvektor ein Ergebnis in Form eines Ausgangsvektors $I_k^{\sim} \xrightarrow{\tau} Q_k^{\sim}$.

Diskussion

Fig. 50 kann auch als allgemeines Schema einer SPS betrachtet werden. Sie zeigt die typische Betriebsweise, bei der zuerst die Eingangsinformation gelesen, dann das Programm ausgeführt und anschließend die Ausgangsinformation geschrieben wird. In der Anweisung entspricht **line_nb** dem Programmzähler, **plc_command, pos** dem Befehlssatz und **temp_nb-1, temp_nb** den CPU-Registern.

Bereits an diese Stelle ist erkennbar, dass die Ähnlichkeit von formalisierten Programmzeilen gegenüber einem in AWL-Programm Sprachelementen erzeugtem SPS-Programm gegeben ist. Der Aufwand ein SPS-Programm zu formalisieren wird dadurch stark verringert und ist automatisiert möglich. Das auf diese Weise formalisierte SPS-Programm hat damit seine ursprüngliche Bedeutung insoweit verloren, als der formalisierte SPS-Befehl eine mögliche Interpretation seiner ursprünglichen Bedeutung darstellt. Der transformierte SPS-Befehl ist damit in eine Menge mathematisch beschreibbarer Objekte zusammengefasst, wobei diese Objekte aus Atomen, Fakten, Formeln und Regeln bestehen, die formallogisch beweisbar sind.

Dieser Weg wird auch in [KP12] verfolgt. Die Grenze für die Berechenbarkeit ist eng verbunden mit der Leistungsfähigkeit des verwendeten Rechensystems, der Überschaubarkeit der benötigten Eingangsparameter zusammen mit der Analyse der gewonnenen Ergebnisse.

Testen mit dem Verifikator

Der Verifikator kennt mehrere Betriebsarten. Unter anderem können Referenzen für die Zustände auch völlig offen gelassen werden. Zugrunde gelegt wird ein einfaches SPS-Programm. Die logische Formel $\mathbf{a} \wedge \neg \mathbf{b} \vee \mathbf{c} = \mathbf{d}$ wird in ein SPS-Programm abgebildet und die zugehörige „Wahrheitstabelle“ dadurch erzeugt, dass alle möglichen Kombinationen von Wahrheitswerten für die Referenzen **a**, **b**, **c**, **d** als „freie“ Variablen durch Expandieren gebildet werden. Diese werden der Reihe nach bearbeitet und als Zuordnung von Anfangswert und dem entsprechenden Zielwert ausgegeben. Die für die Verifikation eingeführte „Klasse“ ist in diesem Beispiel nicht von Interesse und kann ignoriert werden (vgl. [KP12]).

Das SPS-Programm wird entsprechend der in [KP12] beschriebenen Vorgehensweise in prädikatenlogische Formeln umgeformt und liegt dann in der folgenden Form vor (Auszug):

```
...
program(Zi, Zo) :-
    assert(temp_vector(Zi)),
    assert(stack([])),
    line(  1,   1,   1,    _,  VE1),
```

```
    line(  2, ukl,   _,  VE1,  VE2),
    line(  3,  ln,   2,  VE2,  VE3),
    line(  4,   o,   3,  VE3,  VE4),
    line(  5, klz,   _,  VE4,  VE5),
    line(  6,   =,   4,  VE5,    _),
    line(  7, end,   _,    _,    _),
    temp_vector(Zo),
    retractall(temp_vector(_)),
    retractall(stack([])),
    !,
    (stack(X) -> (retractall(stack(X)), !, fail) ; true).
...
1 ?- verfy_all1.
Correct to: "verify_all1"? yes

[0,0,0,0] ==> [0,0,0,0]  Klasse 1
[0,0,0,1] ==> [0,0,0,0]  Klasse 1
[0,0,1,0] ==> [0,0,1,0]  Klasse 1
[0,0,1,1] ==> [0,0,1,0]  Klasse 1
[0,1,0,0] ==> [0,1,0,0]  Klasse 1
[0,1,0,1] ==> [0,1,0,0]  Klasse 1
[0,1,1,0] ==> [0,1,1,0]  Klasse 1
[0,1,1,1] ==> [0,1,1,0]  Klasse 1
[1,0,0,0] ==> [1,0,0,1]  Klasse 1
[1,0,0,1] ==> [1,0,0,1]  Klasse 1
[1,0,1,0] ==> [1,0,1,1]  Klasse 1
[1,0,1,1] ==> [1,0,1,1]  Klasse 1
[1,1,0,0] ==> [1,1,0,0]  Klasse 1
[1,1,0,1] ==> [1,1,0,0]  Klasse 1
[1,1,1,0] ==> [1,1,1,1]  Klasse 1
[1,1,1,1] ==> [1,1,1,1]  Klasse 1
true.

2 ?-
```

Nach der Anfrage an das System werden die offenen Referenzen durch Wahrheitswerte ersetzt und als „Ausgangsvektoren“ ausgegeben. Angemerkt sei, dass die Eingangsvektoren (hervorgehoben) mit deren Eingangszuständen Anfragen an den Algorithmus erzeugen mit der Fragestellung: „Welche Ausgangszustände entsprechen diesen Eingangszuständen?“. In Tab. 36 sind die Zuordnungen entsprechen der Wahrheitswerte zusammengefasst.

a	b	c	d		Anmerkungen zur Ausgabe der Ergebnisse
0	0	0	0	←	1. und 2. Zeile: werden auf das nebenstehende Ergebnis verifiziert
0	0	1	0	←	3. und 4. Zeile: werden auf das nebenstehende Ergebnis verifiziert
0	1	0	0	←	5. und 6. Zeile: werden auf das nebenstehende Ergebnis verifiziert
0	1	1	0	←	7. und 8. Zeile: werden auf das nebenstehende Ergebnis verifiziert
1	0	0	1	←	9. und 10. Zeile: werden auf das nebenstehende Ergebnis verifiziert
1	0	1	1	←	11. und 12. Zeile: werden auf das nebenstehende Ergebnis verifiziert
1	1	0	0	←	13. und 14. Zeile: werden auf das nebenstehende Ergebnis verifiziert
1	1	1	1	←	15. und 16. Zeile: werden auf das nebenstehende Ergebnis verifiziert

Tab. 36: Wahrheitswerte des Ausdrucks **a ∧ ¬b ∨ c = d**

Die tatsächliche Anzahl von offenen Referenzen, die verarbeitet werden können, sind im Wesentlichen von den Systemeinstellungen von PROLOG und vom Speicher des Rechnersystems abhängig, auf dem der Testlauf durchgeführt wird. Mit den getroffenen Vorgaben konnten zum Beispiel mit einem Standard-PC unter 64-bit Windows 7 mit 4 GB Hauptspeicher maximal 12 offene Referenzen aufgelöst werden. Grund dafür ist, dass die unter PROLOG zum Einsatz kommende Tiefensuche mit Backtracking sehr speicherintensiv ist und außerdem jedes zusätzliche Bit den Lösungsraum verdoppelt.[74]

Diskussion

Die Nutzung von Wahrheitswerten für Referenzen als „freie“ Variable bedeutet, dass als Ergebnis gezeigt wird, dass ein SPS-Code genau das bewirkt, was er bewirkt. Damit ist jedoch noch nicht gesagt, dass ein so gefundenes Ergebnis den Vorgaben einer Aufgabenstellung entspricht. Für eine reale Verifikation muss daher die Zielgröße, gleichzusetzen mit dem vorbestimmten Ausgangsverhalten nach Beendigung der SPS-Zyklen, in Relation mit dem durch den Algorithmus gefundenen Ergebnissen gesetzt werden.

Das vorbestimmte Ausgangsverhalten selbst fordert der Auftraggeber in seiner Spezifikation der Aufgabenstellung, das Team der Programmierer ergänzt die in der Regel informellen Spezifikationen und setzt diese in die Funktionalität um. Kein Außenstehender kennt die Anforderungen an die Funktion so gut, wie der genannte Personenkreis. Es ist daher unmittelbar einleuchtend, dass auch die Vorgaben für eine Verifikation genau aus diesem Personenkreis stammen sollten. Es kann damit sichergestellt werden, dass nur das tatsächlich Erforderliche, das „Gewollte“ durch das SPS-Programm erreicht wird.

Ansteuerung der Ventile „Schere“ im Baustein „Stampfen“ (FC38)

Die Ansteuerung der Ventile „Schere“ aus dem SPS-Programmteil FC38 wird als weiteres Beispiel untersucht. Um das Programm dem Verifikator in seiner derzeit vorliegenden Form anzupassen, musste es dem darin implementierten Programm-Dialekt geringfügig angepasst werden (vgl. [KP12]). Das war jedoch nicht ganz ausreichend, weil das Hilfsprogramm die Adressierung von Ein-, Ausgängen und Merkern einschränkt. Die Adressierung der Adress-Byte ist auf einstellige Werte begrenzt, es sind dort nur Adressen mit Byte 0 bis Byte 9 vorgesehen. Für das Fragment im Testprogramm (Quelle) sind die entsprechenden Korrekturen vorgenommen wurden (Tab. 37).

[74] Die Verifikation als Werkzeug bei der Programmerstellung und zum Festlegen von Teststellungen erfordert, dass die Anzahl der zu beobachteten Variablen nicht zu groß werden darf, um die Überblickbarkeit nicht zu gefährden. Deshalb erscheint es, die Anzahl auf zunächst 12 „Unbekannten“ zu belassen, als angemessener Kompromiss.

Quelle	Repräsentation in PROLOG

```
L   M 36.4
UN  M 36.5
O   M 36.7
U   E 14.5
U   M 0.2
UN  E 14.7
U   E 14.4
S   A 40.4
L   M 36.5
U   E 15.2
ON  M 0.2
ON  E 14.5
O(
L   M 37.0
U   E 15.2
)
UN  E 15.0
R   A 40.4
L   E 14.7
U   E 14.5
UN  M 36.5
UN  M 37.0
UN  E 15.1
U   E 14.4
S   A 40.6
L   M 36.5
O   M 37.0
U   E 14.7
ON  E 14.5
UN  E 15.2
R   A 40.6
```

```
program(Zi, Zo) :-
    assert(temp_vector(Zi)),
    assert(stack([])),
    line(  1,   l,   8,    _,  VE1),
    line(  2,  un,   9,  VE1,  VE2),
    line(  3,   o,  10,  VE2,  VE3),
    line(  4,   u,   2,  VE3,  VE4),
    line(  5,   u,   7,  VE4,  VE5),
    line(  6,  un,   3,  VE5,  VE6),
    line(  7,   u,   1,  VE6,  VE7),
    line(  8,   s,  12,  VE7,  VE8),
    line(  9,   l,   9,  VE8,  VE9),
    line( 10,   u,   6,  VE9, VE10),
    line( 11,  on,   7, VE10, VE11),
    line( 12,  on,   2, VE11, VE12),
    line( 13, okl,   _, VE12, VE13),
    line( 14,   l,  11, VE13, VE14),
    line( 15,   u,   6, VE14, VE15),
    line( 16, klz,   _, VE15, VE16),
    line( 17,  un,   4, VE16, VE17),
    line( 18,   r,  12, VE17, VE18),
    line( 19,   l,   3, VE18, VE19),
    line( 20,   u,   2, VE19, VE20),
    line( 21,  un,   9, VE20, VE21),
    line( 22,  un,  11, VE21, VE22),
    line( 23,  un,   5, VE22, VE23),
    line( 24,   u,   1, VE23, VE24),
    line( 25,   s,  13, VE24, VE25),
    line( 26,   l,   9, VE25, VE26),
    line( 27,   o,  11, VE26, VE27),
    line( 28,   u,   3, VE27, VE28),
    line( 29,  on,   2, VE28, VE29),
    line( 30,  un,   6, VE29, VE30),
    line( 31,   r,  13, VE30,    _),
    line( 32, end,   _,    _,    _),
    temp_vector(Zo),
    retractall(temp_vector(_)),
    retractall(stack([])),
    !,
    (stack(X) -> (retractall(stack(X)), !, fail) ; true).


% E 14.4  1  … Eingänge
% E 14.5  2
% E 14.7  3
% E 15.0  4
% E 15.1  5
% E 15.2  6
% M 0.2   7  … Merker (dieser ist immer EIN siehe Spalte 7)
% M 36.4  8
% M 36.5  9
% M 36.7 10
% M 37.0 11
% A 40.4 12  … Ausgänge
% A 40.6 13

 input_vector([_, _, _, _, _, _, 1, _, _, _, _, _, _], 1).
output_vector([_, _, _, _, _, _, 1, _, _, _, _, _, _], 1).

% Position …   1  2  3  4  5  6  7  8  9  10 11 12 13
```

Tab. 37: Quelle und Testprogramm für PROLOG

Kategoriesierung „bad states in result“

Systembedingt sind, wie weiter oben festgestellt, mit der genutzten Testumgebung max. 12 freie Variable im Verifikationsprozess handhabbar. Weil der Merker **M 0.2** die Funktion als Startbedingungen (Rundtisch ist in Position, Motore sind eingeschaltet) für den Ablauf freigibt oder sperrt, wurde er mit logisch EIN vorgegeben.

Es werden im Beispielprogramm in Summe 4.096 mögliche Programmzyklen der SPS überprüft. Die Ausgabe erfolgt in einer Zuordnung, die tabellarisch dargestellt werden kann. Tab. 38 zeigt einige dieser Übergänge als Beispiele,

Eingangsvektor		Ausgangsvektor	Anmerkung
		↓ ↓	
[0,0,0,1,0,1,**1**,0,0,0,1,0,0]	==>	[0,0,0,1,0,1,**1**,0,0,0,1,**0,0**]	… Typ 1
[0,0,0,1,0,1,**1**,0,0,0,1,0,1]	==>	[0,0,0,1,0,1,**1**,0,0,0,1,**0,1**]	… Typ 2
[0,0,0,1,0,1,**1**,0,0,0,1,1,0]	==>	[0,0,0,1,0,1,**1**,0,0,0,1,**1,0**]	… Typ 3
[0,0,0,1,0,1,**1**,0,0,0,1,1,1]	==>	[0,0,0,1,0,1,**1**,0,0,0,1,**1,1**]	… Typ 4

Tab. 38: Ausschnitt der PROLOG Ausgabe

Die Lesart für Tab. 38 lautet: die Konfiguration der Zustände im Eingangsvektor (linker Block) erzeugt am Ende des Programmdurchlaufs die „Ergebnis-Konfiguration“ (rechter Block). Das jeweils siebente Element stellt den Zustand des oben genannten Merkers **M 0.2** dar. Das Hauptinteresse ist jedoch, welche Zustände die Ausgänge einnehmen können. Es sind bei zwei Ausgängen 4 Typen von Ausgangskonfigurationen möglich. Es haben sich für Ausgangskonfigurationen nach Typ 1 in Summe 1.774 verschiedene Zyklen ergeben, nach Typ 2 waren es 846 Zyklen, nach Typ 3 waren es 946 Zyklen, nach Typ 4 waren es 539 Zyklen und in Summe 4.096 Zyklen.

Definiert man laut der Aufgabenstellung in einer Kategoriesierung von Programmübergängen „bad states in result“, trifft das im Untersuchungsfall auf die Programmübergänge vom Typ 2 zu, weil solange die Schere nicht nach vorne bewegt worden ist (Ausgang **A 40.4**) dürfen die Schneidblätter der Schere nicht geschlossen werden (Ausgang **A 40.6**). Im Wesentlichen sagt diese Feststellung nur aus, dass die betroffenen Zyklen genauer zu betrachten sein werden. Die entsprechenden Eingangskonfigurationen sind mögliche Testfälle, die genauer untersucht werden können.

Zu bedenken ist, das bis hierher erst ein Teilaspekt im SPS-Programm untersucht worden ist. Gefunden wurden „kritische“ Konfigurationen, die theoretisch vorkommen können. Das bedeutet jedoch nicht zwangsweise, dass ein Programmfehler vorliegt, sondern nur, dass ein Programmfehler nicht ausgeschlossen werden kann. Erst wenn die weiteren Untersuchungser-

gebnisse zeigen, dass solche Konfigurationen auftreten können, sind programmtechnische Maßnahmen notwendig.

Ergänzungen im Verifikator

Gegenüber dem in [KP12] beschriebenen PROLOG Programm des Verifkators ist der genutzte Befehlssatz um einige neu hinzu gekommenen Kommandos angepasst worden. Das bedeutet, dass auch die Hilfsprogramme für den Transfer der Anweisungsliste in PROLOG als auch die Verifikationsalgorithmen erweitert werden müssen.

Eränzungen im Prolog Verifikator: Neue Kommandos

Positive Flankenbildung „edge up“

```
/* Funktion: Impulsbildung beim Einschalten

   Ist das Verknüpfungsergebnis logisch EIN wird bei der positiven Flanke in
   einem Zwischenmerker das Verknüpfungsergebnis in einem Flankenmerker
   gespeichert. War der Wert des Flankenmerkers logisch NULL, wird das
   Verknüpfungsergebnis für die nachfolgenden Kommandos auf logisch EIN
   ausgegeben. War der Wert des Flankenmerkers logisch EIN, wird das
   Verknüpfungsergebnis für die nachfolgenden Kommandos mit logisch NULL
   ausgegeben.
   Opi zeigt auf den Wert des Zwischenmerkers (Flankenmerker)
*/
line(_, fp, Opi, VEalt, VEalt) :-
    temp_vector(Z),
    retract(temp_vector(Z)),
    neg(VEalt, X),
    insert(Opi, Z, Z1, X),
    assert(temp_vector(Z1)).
```

Negative Flankenbildung „edge down“

```
/* Funktion: Impulsbildung beim Ausschalten

   Ist das Verknüpfungsergebnis logisch Null wird bei der negativen Flanke in
   einem Zwischenmerker das Verknüpfungsergebnis in einem Flankenmerker
   gespeichert. War der Wert des Flankenmerkers logisch EIN, wird das
   Verknüpfungsergebnis für die nachfolgenden Kommandos auf logisch EIN
   ausgegeben. War der Wert des Flankenmerkers logisch NULL, wird das
   Verknüpfungsergebnis für die nachfolgenden Kommandos mit logisch NULL
   ausgegeben.

   Opi, Alt, Neu
   0,   0,   0.
   1,   0,   1.
   1,   1,   0.
   0,   1,   0.

   Opi zeigt auf den Wert des Zwischenmerkers (Flankenmerker)
*/
line(_, fn, Opi, VEalt, VEneu) :-
```

```
    temp_vector(Z),
    nth1(Opi, Z, Optemp),
    fneg(Optemp, VEalt, VEneu).
    assert(temp_vector(Z1)).

fneg(A, B, C) :- free_variables(A, [_]), free_variables(B, [_]),
free_variables(C, [_]).

fneg(A, B, C) :- free_variables(A, [_]), B = 0, !, C = 0.
fneg(A, B, _) :- free_variables(A, [_]), B = 1.
fneg(A, B, C) :- free_variables(B, [_]), A = 0, !, C = 0.

fneg(0, 0, 0).
fneg(1, 0, 1).
fneg(1, 1, 0).
fneg(0, 1, 0).
```

Einschaltverzögerung (Timer)

```
/* Funktion: Zeitverzögerung nach dem Einschalten

   Ist das Verknüpfungsergebnis logisch EIN wird der einschaltverzögerte
   Timer gestartet und sein Ausgang für die Verifikation so behandelt, als
   sei er bereits abgelaufen. Ist der Wert des Verknüpfungsergebnis logisch
   NULL, wird der einschaltverzögerte Timer wieder ausgeschaltet.

   Opi zeigt auf den Wert logischen Wert des Timers
*/
line(_, se,  Opi, VEalt, VEalt) :-
    temp_vector(Z),
    retract(temp_vector(Z)),
    insert(Opi, Z, Z1, VEalt),
    assert(temp_vector(Z1)).
```

Ausschaltverzögerung (Timer)

```
/* Funktion: Zeitverzögerung nach dem Ausschalten

   Ist das Verknüpfungsergebnis logisch EIN wird der ausschaltverzögerte
   Timer gestartet. Sein Ausgang wird sofort gesetzt. Ist das
   Verknüpfungsergebnis logisch AUS wird der Zeitablauf gestartet und nach
   Ablauf der Zeit wird der Timer ausgeschaltet.
   Für die Verifikation wird die Ausschaltverzögerung so behandelt, dass
   diese nur durch Änderung im Prozessabbild ausgeschaltet werden kann.

   Opi zeigt auf den Wert logischen Wert des Timers
*/
line(_, sa,   _, 0, 0).
line(_, sa, Opi, 1, 1) :-
    temp_vector(Z),
    retract(temp_vector(Z)),
    insert(Opi, Z, Z1, 1),
    assert(temp_vector(Z1)).
```

Neues Netzwerk

```
/* Funktion: Setzen des Verknüpfungsergebine auf logisch EIN.
*/
line(_, net,   _, 1, 1).
```

Schlussbemerkungen

Das Arbeiten mit dem Verifikator erfordert eine gewisse Praxis. In jedem Programmteil existieren bestimmte Eingangszustände, die in der Form von Wertetabbellen für die zu erwartenden Anfangs- und Endzustände paarweise aufgestellt werden müssen.

In Bezug auf das SPS-Programm bedeuten solche Wertepaare Zustandsübergänge. Um den Überblick auch bei einer großen Zustandsmenge zu behalten, werden die beteilgten Zustände der Reihe nach gruppiert. Für die Definition dieser Zustandsübergänge werden immer initiale und finale Zustände des untersuchten Programmabschnitts zu berücksichtigen sein. Da die im Programm genutzten Programmadressen für diese Zustände nicht notwendigerweise in einer geordneten Form vorliegen, können auch größere Lücken in der Adressierung vorkommen.

Andererseits werden bei der Programmierung von SPS-Programmen üblicherweise auch Zuordnungslisten für die verwendeten Signalgeber (Eingänge und Ausgänge) sowie auch für die intern verwedeten Merker definiert. In der komprmierten Form werden ihre Zuordnung zu bestimmten Programmteilen durch die Analyse bestimmt (vgl. Anhang C: Analyse durch Hilfsprogramme). Ziel wird es sein, die Anzahl der unbestimmten Referenzen für die zu untersuchenden Zustände gering zu halten, um die Übersichtlichkeit zu erhöhen. Im Gegensatz dazu steht, dass möglichst umfangreiche Programmteile in einem Zug überprüft werden können, um das Verständnis der wechselsitigen Wirkung besser darzustellen.

Anhang E: Lebenslauf

Geburtsdatum				17.1.1942
Geburtsort	Bad Vöslau			
Vater	Dipl.-Ing. Walter Kucera			
Mutter	Dietlinde Koppitsch			
Volksschule	vom	1.9.1949	bis	5.7.1952
Realschule Wien 1 (R1 Schottenbastei)	vom	1.9.1952	bis	1.6.1960
Matura				1.6.1960

Universitätsstudien, Abschlüsse

Nicht abgeschlossenes Studium der Elektrotechnik als Werkstudent an der Technischen Hochschule Wien	vom	WS 1960/61	bis	SS 1967
Verleihung zur Führung der Standesbezeichung „Ingenieur“ (Ing.)	Dez. 1972			
Studium der Elektrotechnik (wissenschaftlicher Hochschulstudiengang) an der FernUniversität Hagen (D)	vom	WS 1994/95	bis	SS 2001
Diplomurkunde des Fachbereiches Elektrotechnik	April 2000			
Diplomurkunde des Fachbereiches Elektrotechnik und Informationstechnik	Oktober 2001			
Weiterführende Studien im Fachbereich Informatik, Bachelor-Prüfung Informatik über Betriebssysteme an der FernUniversität Hagen (D)	August 2002			
Dissertation an der TU-Wien (Prof. Kopacek), Doktorat in technischen Wissenschaften (Maschinenbau)	Juli 2004			
Dissertation an der Universität Kassel (Prof. Böhm), Doktorat der Ingenieurwissenschaften (Maschinenbau)	April 2013			

Berufliche Tätigkeit

Elektroingenieur, Projektingenieur, Konstruktionsleitung, Projektbearbeitung,	vom	3.7.1967	bis	30.9.1975
Geschäftsführer	vom	30.9.1975	bis	31.7.1980
Selbständige Gewerbeausübung (diverse Gewerbeberechtigungen und Konzessionen) Technisches Büro für Elektrotechnik, Elektroinstallationen, nicht protokolliertes Einzelunternehmen (KMU) Tätigkeitsprofil**:** Automatisierung und Steuerungstechnik mit elektronischen Steuerungen, SPS und Systemlösungen, Prozessvisualisierungen, Leittechnik, Planungsarbeiten (Ausschreibungen), Inbetriebnahmen und Servicearbeiten	vom	1.1.1980	bis	31.12.2006
Geschäftsführender Gesellschafter der Fa. M-Software Mikroprozessorprogrammierung Ges.m.b.H. (KMU) Tätigkeitsprofil**:** Maschinen und Anlagenautomatisierung, Steuerungstechnik, Antriebstechnik für Gewerbe und Industrieunternehmen	vom	1.6.1983	bis	31.12.2015
Vertragslehrer an der HTLuVA Wien 5, Spengergasse	vom	WS 2000/01	bis	SS 2006
Lektor an der fh campus-favoriten bzw. FH Campus Wien	vom	WS 2000/01	bis	5.12.2012
FH-Prof. an der FH Campus Wien	seit	6.12.2012	bis	laufend
Lehrbeauftragter an der Universität Kassel	seit	SS 2014	bis	laufend

Publikationen

Kucera, G.: „Automatisierung in der Produktion. Steuerung von Fertigungsautomaten“, Buch, FH Campus Wien, 2017, ISBN 987-3-902614-35-3
Korak G., **Kucera** G.: „Optisches Positionserfassungssystem“, Tagungsband zur 4. Tagung Innovation Messtechnik: Neuheiten in der Sensorwelt und in der Messelektronik sowie deren Anwendungsgebiete, Konferenzbeitrag, Peer Reviewed Paper, 2015, ISBN-10: 3844035605
Korak G., **Kucera** G.: „Optical Tracking System “, International Journal of Electronics and Telecommunications, 2015, eISSN: 2300-1933
Korak G., **Kucera** G.: „Entwicklung eines optischen Positionserfassungssystems zur Robotersteuerung“, Beitrag in einem Tagungsband, Peer Reviewed Paper, 2015, ISSN 2411-5428
Sandtner H., **Kucera** G., Unterweger U., Geyer S., Korak G., Zellner G.: „Verbesserung der Adhäsionseigenschaften für Aufschmelzverfahren von thermoplastischen Kunststoffen“, Beitrag in einem Tagungsband, Peer Reviewed Paper, 2014, ISBN 978-3-9503491-9-1
Kucera, G.: „Programmanalyse von SPS-Programmen als Teilschritt für die automatisierte Verifikation“, Konferenzbeitrag, Tagungsband zum 7. Forschungsforum der österreichischen Fachhochschulen, Seite 47-54, FH Vorarlberg GmbH, 2013, ISBN: 978-3-86573-709-0
Paulis, H., **Kucera**, G.: „Verifikation von SPS-Programmen“, sps ipc drives, Elektrische Automatisierung, Systeme und Komponenten, Internationale Fachmesse und Kongress, Nürnberg, 26. – 28. November 2013, Konferenzbeitrag, Tagungsband, S.73-80, Veranstalter: Mesago Messemanagement GmbH, Stuttgart, VDE VERLAG GMBH, ISBN 978-3-8007-3560-0
Kucera, G.: „Verbesserung der Eigenschaften von Stampfverbindungen für Kohlebürsten“, Fortschrittsberichte aus der Produktionstechnik, Dissertation, Shaker Verlag, ISBN 978-3-8440-1994-0, 2013
Sandtner, H. **Kucera**, G. Unterweger, U. Geyer, S. Korak, G. Dötsch H., Ledermann, B.: „AVIT - Einfaches Anlernen von Industrierobotern an produktionsrelevante Bewegungsabläufe mittels magnetischer Trackingsysteme“, Konferenzbeitrag, Tagungsband zum 7. Forschungsforum der österreichischen Fachhochschulen, FH Vorarlberg GmbH, 2013, ISBN 978-3-86573-709-0
Kucera, G. Paulis, H.: „Schritte zur Verifikation von SPS-Programmen“, Buch, FH Campus Wien, ISBN: 978-3-902614-24-7, 2012
Kucera, G.: „Elektrische Signalbeeinflussung in technischen Prozessen“, Buch, FH Campus Wien, ISBN: 987-3-902614-23-0, 2012
Kucera, G., Paulis, H., Petz, A., Oberpertinger, R.: „Einführung für das Erstellen wissenschaftlicher Arbeiten auf dem Gebiet der Technik“, Buch, FH Campus Wien, ISBN: 987-3-902614-25-4, 2012
Kucera, G.: „Die wesentlichen Prozessparameter zur maschinellen Herstellung von Stampfkontakten für Kohlebürsten, Dissertation, TU Wien, 2004
Kucera, G.: „Ertüchtigung einer Mikrocontroller Plattform zum Einsatz in sicherheitsgerichteten Anwendungen“, Diplomarbeit im Fachbereich Elektrotechnik und Informationstechnik, FernUniversität Hagen, 2001

Kucera, G.: „Entwicklung einer speicherprogrammierbaren Steuerung“, Diplomarbeit im Fachbereich Elektrotechnik, FernUniversität Hagen, 2000