Ein System zur Automatisierung der Planung und Programmierung von industriellen Roboterapplikationen

von der
Fakultät für Elektrotechnik und Informationstechnik
der
Universität Dortmund
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften

von
Dipl.-Inform. Bernd Lüdemann-Ravit
Dortmund
2005

Tag der mündlichen Prüfung:	31. August 2005
Hauptreferent:	Prof. Dr.-Ing. E. Freund †
	Prof. Dr.-Ing. U. Schwiegelshohn
Korreferent:	Prof. Dr.-Ing. H. Wörn

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Roboterforschung der Universität Dortmund.

Herrn Prof. Dr.-Ing. E. Freund, dem damaligen Leiter des Instituts, gilt mein besonderer Dank für die stetige Förderung dieser Arbeit, sein großes Interesse daran und die Schaffung ausgezeichneter Rahmenbedingungen, die in hohem Maße zum Gelingen dieser Arbeit beigetragen haben. Ich bedauere sehr, dass ich Herrn Prof. Dr.-Ing. E. Freund aufgrund seines Tods nicht mehr persönlich danken kann.

Herrn Prof. Dr.-Ing. U. Schwiegelshohn, dem Leiter des Instituts für Roboterforschung, danke ich für die freundliche und unkomplizierte Übernahme des Hauptreferats, für sein großes Interesse sowie für seine detaillierten, inhaltlichen Anregungen.

Herrn Prof. Dr.-Ing. H. Wörn, dem Leiter des Instituts für Prozessrechentechnik, Automation und Robotik der Universität Karlsruhe danke ich für die freundliche Übernahme des Korreferats sowie die sehr gute fachliche Unterstützung im Zusammenhang mit dieser Arbeit.

Herrn Prof. Dr.-Ing. E. Handschin, dem Vorsitzenden des Promotionsausschusses der Fakultät für Elektrotechnik und Informationstechnik der Universität Dortmund, möchte ich für die schnelle, unkomplizierte Unterstützung zur Fortführung des Promotionsverfahrens nach dem Tod von Herrn Prof. Dr.-Ing. E. Freund danken.

Mein weiterer Dank gilt allen damaligen Kollegen des Instituts für Roboterforschung. Sie haben durch viele anregende Diskussionen sowie sehr großer Hilfsbereitschaft, einen erheblichen Beitrag zum Gelingen dieser Arbeit geleistet. Des Weiteren möchte ich mich bei Herrn Dipl.-Inform. O. Stern, Herrn Dr.-Ing. D. Pensky, Herrn Dipl.-Ing. A. Hypki, Herrn Dipl.-Ing. M. Seitter sowie meinem Vater Dipl.-Ing. K. Lüdemann-Ravit für die Durchsicht meiner Arbeit und die wertvollen Hinweise bedanken.

Vor allem möchte ich mich bei meiner Frau Diana sowie bei meinen Kindern Moritz und Sven bedanken, ohne deren Unterstützung, Rücksichtsnahme und Geduld diese Arbeit nicht möglich gewesen wäre.

Gärtringen, im September 2005

Bernd Lüdemann-Ravit

Inhalt

Symbole

$\mathbb{R}^n$	n-dimensionaler Raum reeller Zahlen
${}^A\underline{T}_B$	Homogene Transformation von Koordinatensystem A in B
$\overrightarrow{x}$, x_i	Zellenlayoutänderung mit dessen reellwertigen Komponenten x_i
$x_i^{\min}$, $x_i^{\max}$	minimaler bzw. maximaler Wert einer Komponente x_i
n_i	Anzahl der diskreten Werte von x_i
m	Anzahl Zellenlayoutkandidaten
X	Beschränkte Folge der Zellenlayoutkandidaten
Δx_i	Differenz zwischen zwei Vektorkomponenten x_i
$\overrightarrow{x_k}$, x_{ki}	Element der Folge X mit Komponenten x_{ki}
$F(\overrightarrow{x})$, $F(\overrightarrow{x}^*)$, $\overrightarrow{x}^*$	Zielfunktion, Optimum, Optimalstelle
M	Zulässiger Bereich
V, V_i	(Menge der) Steuerknoten i
E^{Init}, $E_{i,j}^{Init}$	(Menge der) Initialisierungskanten zwischen V_i und V_j
E^{Aus}, $E_{i,j}^{Aus}$	(Menge der) Ausführungskanten zwischen V_i und V_j
d_{Mantel}	Zementmanteldicke
E_{Anfang}, E_{Mitte}, E_{Ende}	Begrenzungsebenen
E, E_i	Folge E von Ebenen E_i
$d_{A,B}$	Abstand zweier Ebenen E_A und E_B
d_S^{Mantel}, d_S^{Bahn}	Ebenenabstand bei Mantel- bzw. Bahnberechnung
α	Abtastwinkel
N	Anzahl der Punkte eines Polygons
P	Polygonfolge
$Poly_i$	Polygon als Element der Mengen $P^{\mathrm{Pr}othese}$, P^{Mantel}
p_{ij}	Polygonpunkte eines Polygons $Poly_i$
n_{ij}	Anzahl der Punkte p_{ij} eines Polygons $Poly_i$
$p_i^{Schwerpunkt}$	Schwerpunkt von Polygon $Poly_i$
$Poly^{\mathrm{Si}cherheit}$, $d_{\mathrm{Si}cherheit}$	Sicherheitspolygon, Sicherheitsabstand
d_f, v_f	Vorschub und Vorschubgeschwindigkeit
d_z, v_z	Zustellung und Zustellgeschwindigkeit
r_{Kopf}, $r_{Fräse}$	Radius von Fräskopf bzw. Fräse
$Poly_{ij}^{Orientierung}$	Polygon zur Beschreibung aller Orientierungen für Bahnpunkt p_{ij}
p_{ij}^{Vorzug}, $p_{ij}^{Neigung}$	Punkte zur Festlegung der Fräsrichtung für Bahnpunkt p_{ij}
t_{Ausf}	gemessene Ausführungszeit auf realer Robotersteuerung
t_{Ausf}^C	konstanter Anteil von t_{Ausf}
T_W, t_W, $t_{W,i}$	Gesamtzeit/Einzelzeit für Peripherieinteraktion
t_K, t_{K_i}	Zeit für die Kommunikation mit Peripherie
t_G, $t_{G,i}$	Zeit für die Ausführung der Peripherie
S_i, G_j	Schraube, Gewinde
${}^At_B$	Zeit für Beweging von A nach B

Abkürzungen

AST	Abstrakter Syntaxbaum
BAPS	Bewegungs-und Ablauf-Programmiersprache
BASIC	Beginner's All Purpose Symbolic Instruction Code
CAD	Computer Aided Design
CAN	Controller Area Network
CAR	Computer Aided Robotics
COSIMIR®	Cell Oriented Simulation of Industrial Robots
CT	Computertomografie
E/A	Eingang/Ausgang
EBNF	Extended Backus-Naur Form
eM-OLP™	eM-Workplace-Modul: Open Language Program
GSL	Graphic Simulation Language
ICR	Intermediate Code for Robots
IGES	Initial Graphics Exchange Specification
IRL	Industrial Robot Language
IGRIP®	Interactive Graphical Robotic Instruction Program
IRDATA	Industrial Robot Data
KRC	KUKA Robot Controller
KRL	KUKA Robot Language
LALR(1)	Look ahead, left-to-right scanning of the input, righmost derivation
NC	Numerical Control
MBA4	MELFA BASIC IV
OPC	Open Process Control
PTP	Point-To-Point
RC	Robot Controller
RRS	Realistic Robot Simulation
SPS	Speicherprogrammierbare Steuerung
STL	Stereolithographie-Format
TCP	Tool Center Point
TCP/IP	Transmission Control Protocol/Internet Protocol
TDL	Task Description Language
u. d. N.	unter der Nebenbedingung
UML	Unified Modeling Language
VBA	Visual Basic® for Applications
VRC	Virtual Robot Controller

1 Einleitung

Um die Effizienz bei der Entwicklung von Roboter-Fertigungssystemen zu steigern, setzen Unternehmen zunehmend rechnergestützte Systeme zur grafischen, dreidimensionalen Simulation ein [73]. Diese so genannten CAR-Systeme (Computer Aided Robotics) ermöglichen den Aufbau einer virtuellen Roboter-Fertigungszelle und die Analyse des darin simulierten Verhaltens, ohne dass dazu reale Fertigungskomponenten erforderlich sind [125].

1.1 Problemstellung

Die Haupteinsatzgebiete der CAR-Systeme sind einerseits die Offline-Programmierung der Roboter und andererseits die Planung von Roboter-Fertigungszellen [42, 87]. Unter Verwendung heutiger CAR-Systeme kann ein CAR-Anwender zahlreiche Planungs- und Programmieraufgaben nur mit einem hohen Kosten- und Zeitaufwand lösen [124]. Dies kann den wirtschaftlichen Einsatz eines Industrieroboters in Frage stellen [22, 160].

Bei der Programmierung eines Roboters zur Bearbeitung variantenreicher Produkte muss der CAR--Anwender für die Programmierung jeder einzelnen Variante zahlreiche Schritte manuell wiederholen [11]. Das bedeutet, dass für die zeitintensive Programmierung jeder einzelnen Produktvariante hoch qualifizierte CAR-Anwender zum Einsatz kommen müssen [129], weil die Software zur Roboterprogrammierung immer umfangreicher und komplexer wird [172]. Dem entgegen steht ein Mangel an qualifizierten CAR-Anwendern [42, 86]. Um die Gesamtkosten zur Programmierung aller Varianten zu reduzieren, wird zunehmend gefordert, die notwendigen Roboterprogramme automatisiert zu erzeugen [78, 89], damit zur Programmierung neuer Varianten keine hoch qualifizierten CAR-Anwender nötig sind [63]. Dies kann nur durch Programmgeneratoren gelingen, die für das zu programmierende Produktspektrum maßgeschneidert sind. Durch den notwendigen hohen Automatisierungsgrad ist eine starke Anpassung der Programmgeneratoren an die jeweilige Aufgabe erforderlich [23, 24], um deren spezifische Anforderungen zu berücksichtigen.

Bei der Planung von Roboter-Fertigungszellen ergeben sich komplexe Probleme, besonders wenn ein optimiertes Roboterverhalten in der Fertigungszelle angestrebt wird, z. B. die Bestimmung eines Zellenlayouts, bei dem der Roboter eine gegebene Aufgabe in möglichst kurzer Zeit ausführen soll. Die heutigen CAR-Systeme besitzen zwar eine leistungsfähige grafische Darstellung für eine realistische Bewegungssimulation, aber für die Beantwortung komplexer Fragen in der Planungsphase muss der CAR-Anwender zeitaufwendige manuelle Trial-and-Error-Methoden anwenden [13]. Der CAR-Anwender wiederholt dabei die Schritte Programmerstellung (oder Programmmodifikation), Bewegungssimulation und Ergebnisbewertung so lange, bis ihm ein zufrieden stellendes Resultat vorliegt. Die Qualität der Lösungen hängt in starkem Maße von den Fähigkeiten und der Motivation des CAR-Anwenders ab [14, 154]. In der Praxis wird oftmals die erste Lösung für ein Planungsproblem umgesetzt, weil der zu leistende Aufwand für eine weitere Verbesserung der Lösung zu hoch ist [11]. Mit einem System, das die Schritte während der Planungsphase automatisieren kann, eröff-

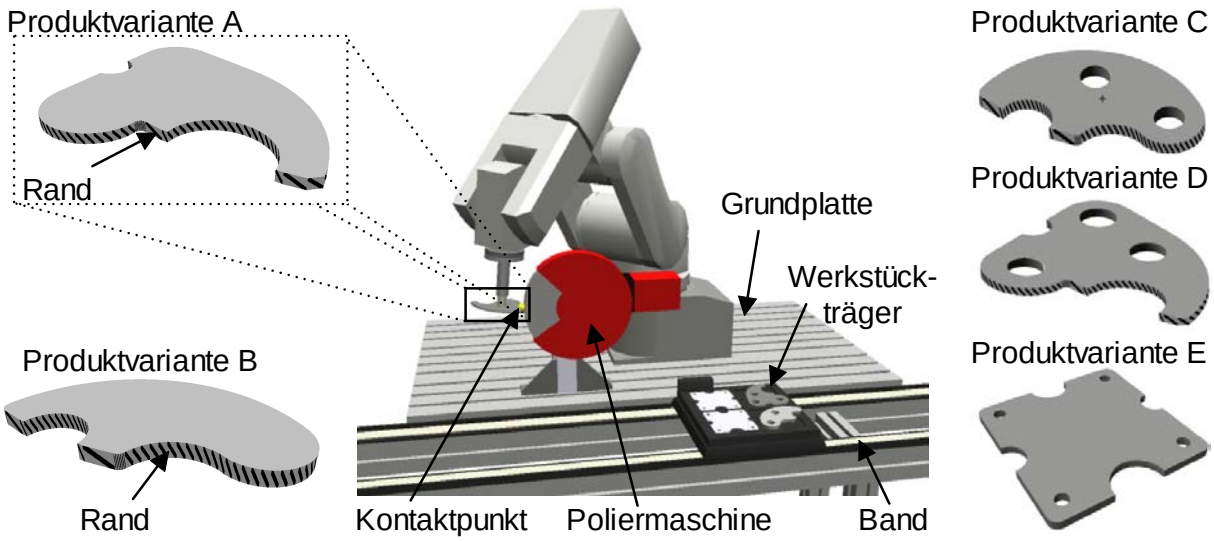


Bild 1.1: Polieren von variantenreichen Produkten

net sich dem CAR-Anwender die Möglichkeit, viele Planungsvarianten auszuprobieren und daraus die für ihn beste Lösung seines Planungsproblems zu ermitteln. Während das System die Varianten automatisch ausprobiert, muss der CAR-Anwender nicht am System anwesend sein. Eine derartige Unterstützung des CAR-Anwenders zur Lösungssuche ist in heutigen CAR-Systemen nur unzureichend anzutreffen. Das Potenzial zur Ermittlung optimierter realisierbarer Lösungen ist bei weitem noch nicht ausgeschöpft [91].

Zur Verdeutlichung der Aufgaben, die der CAR-Anwender mit heutigen Systemen nur unter Aufbringung eines hohen Zeitaufwands lösen kann, zeigt Bild 1.1 eine typische Applikation. Das Beispiel zeigt sowohl das Problem der Programmierung von variantenreichen Produkten als auch Planungsprobleme, deshalb dient es im Verlauf der Arbeit für Erklärungen zum entwickelten System.

Ein Roboter soll in dieser Applikation unterschiedliche Metallplatten (z. B. Produktvarianten A-E) von einem Werkstückträger nehmen und deren schraffierten Rand polieren. Im Verlauf der Roboterprogrammierung für jede einzelne Metallplatte muss der CAR-Anwender viele Roboterstellungen derart festlegen, dass der Roboter den Metallplattenrand entlang des Kontaktpunkts der Poliermaschine führt. Der aufwendige Faktor bei der Programmierung dieser Applikation liegt in der Definition der notwendigen Roboterstellungen, da diese für die einzelnen Varianten größtenteils unterschiedlich sind. Bisher erstellen CAR-Anwender diese Roboterprogramme vollständig manuell oder teilautomatisch. In letzterem Fall muss der CAR-Anwender in der Regel manuelle Modifikationen am Programm vornehmen, bevor er die Roboterprogramme in der realen Roboter-Fertigungszelle einsetzen kann [128]. Die CAR-Systeme berücksichtigen nur in Spezialfällen applikationsspezifische Anforderungen bei der Programmgenerierung. Damit stellen sich folgende Fragen:

- Wie kann man den Gesamtaufwand zur Erstellung der Roboterprogramme reduzieren?
- Wie kann man Personen, die die umfangreichen Kenntnisse eines CAR-Anwenders nicht besitzen, in die Lage versetzen, Programme für neue Metallplatten von ähnlichem Charakter selbstständig zu erstellen?

In der Planungsphase dieser Roboter-Fertigungszelle (Bild 1.1) stellen sich weitere Fragen:

- Wo muss die Poliermaschine auf der Grundplatte positioniert werden, sodass der Roboter den Poliervorgang in möglichst kurzer Zeit und fehlerfrei ausführt?
- Welcher Robotertyp von welchem Hersteller kann diese Aufgabe für eine bestimmte Produktvariante in möglichst kurzer Zeit und fehlerfrei ausführen?
- Welchen Roboter und welche Position der Poliermaschine muss man wählen, wenn ein Mix aus verschiedenen Varianten berücksichtigt werden soll?

Für die Beantwortung solcher Fragen muss der CAR-Anwender viele Planungsvarianten durchprobieren [124]. Jede Variante erfordert ein neues oder modifiziertes Roboterprogramm, dessen Erstellung mit heutigen CAR-Systemen größtenteils manuell erfolgen muss. Der CAR-Anwender muss das Programm in einem Simulationslauf für jede einzelne Planungsvariante ausführen, beobachten und bewerten. Die Programmerstellung und der Simulationslauf sind dabei die beiden wesentlichen Faktoren für den hohen Planungsaufwand.

1.2 Zielsetzung

Vor diesem Hintergrund ist das Ziel dieser Arbeit die Entwicklung eines Systems, mit dem man die notwendigen Schritte zur Programmierung und Planung einer Roboter-Fertigungszelle automatisieren kann. Ein solches System soll folgende Kriterien erfüllen:

1. Das System soll es dem CAR-Anwender ermöglichen, Programmgeneratoren zu erstellen, die Roboterprogramme nach einer von ihm definierten Vorschrift für ein festgelegtes Produktspektrum erzeugen.
2. Mit diesen Programmgeneratoren sollen Personen, die die umfangreichen Kenntnisse eines CAR-Anwenders nicht besitzen, selbstständig Roboterprogramme für neue Produktvarianten erzeugen können.
3. Die generierten Roboterprogramme sollen ohne weitere Konvertierungen auf industriellen Robotersteuerungen ausführbar sein.
4. Das System soll die Programme in unterschiedlichen Robotersprachen erzeugen können, ohne dass der CAR-Anwender diese Sprachen beherrschen muss.
5. Bei der Bewertung von Planungsergebnissen (z. B. Zellenlayout) soll das System Eigenschaften der später eingesetzten Robotersteuerung berücksichtigen können (z. B. Ausführbarkeit, Ausführungszeiten, Fehlverhalten).

6. Das System soll eine Planungsvariante durch die Ausführung des zugehörigen Roboterprogramms auf Umsetzbarkeit prüfen. Nach der Planungsphase soll mit dem Planungsergebnis ein fehlerfrei ausführbares Programm vorliegen, das in der Inbetriebnahmephase einsetzbar ist.

7. Das System soll es dem CAR-Anwender in der Planungsphase ermöglichen, Roboter unterschiedlicher Hersteller gemäß einstellbarer Kriterien zu vergleichen.

8. Der CAR-Anwender soll Roboterprogramme erzeugen können, die nach seinen Kriterien und unter Berücksichtigung möglichst aller applikationsspezifischen Bedingungen optimal sind.

Betrachtet man alle diese Kriterien, so ist der Schlüssel zu deren Erfüllung, dass das System *steuerungsspezifische* Roboterprogramme erzeugt, ausführt und bewertet. Das bedeutet, dass das Roboterprogramm in der herstellerspezifischen Steuerungssyntax vorliegen und auf der entsprechenden industriellen Robotersteuerung ausführbar sein muss. Die Hauptschwierigkeit liegt dabei an der Tatsache, dass es mehrere hundert Roboterprogrammiersprachen gibt und nahezu jeder Hersteller seine eigene Sprache entwickelt hat [70]. Die Versuche, diese zu normieren (z. B. DIN 66312 IRL - Industrial Robot Language [2]), muss man als gescheitert betrachten [148], weil die Roboterhersteller die Normen in ihren Robotersteuerungen nicht umsetzen [129].

Zur Erfüllung der genannten Kriterien können die folgenden Ziele der Arbeit abgeleitet werden:

1. Die Systemstruktur muss so entworfen werden, dass der CAR-Anwender das System auf die sich in der industriellen Praxis ergebenden Anforderungen anpassen kann. In der Inbetriebnahmephase ergeben sich z. B. oft unvorhersehbare Anforderungen. Ausschließlich diese Anpassungsfähigkeit des Systems ermöglicht die Berücksichtigung der applikationsspezifischen Anforderungen bei einem industriellen Robotereinsatz. Nur deren Berücksichtigung ermöglicht es dem CAR-Anwender, einen möglichst großen Anteil des Optimierungspotenzials seiner Applikation zu erschließen.

2. Der vom System zur Verfügung gestellte Basisfunktionsumfang muss festgelegt werden, den der CAR-Anwender bei der Lösungsfindung seiner Planungs- und Programmieraufgaben häufig benötigt. Es muss ein Konzept erstellt werden, wie der CAR-Anwender diese Basisfunktionen für die eigene Lösungsfindung flexibel einsetzen und kombinieren kann.

3. Damit der CAR-Anwender für die Generierung der Roboterprogrammvarianten die spezifische Steuerungssyntax nicht kennen muss, ist ein Konzept zu entwickeln, mit der CAR-Anwender bei der Angabe über den Inhalt der Roboterprogramme möglichst viel Funktionen heutiger industrieller Robotersteuerungen verwenden kann, und trotzdem unabhängig von einer konkreten Steuerung ist. Das bedeutet, dass der CAR-Anwender den Inhalt der Roboterprogramme in einem neutralen Format formuliert, welches das System in die spezifische Syntax einer konkreten Steuerung umwandelt. In dem neutralen Format dürfen nur Funktionen verwendbar sein, die in allen zu betrachtenden Steuerungen verfügbar sind.

4. Die Information über die spezifische Steuerungssyntax muss offen zugänglich sein. Dadurch erfordern Änderungen der Steuerungssyntax (z. B. neue Robotersteuerungsversion) keinen tiefen Eingriff in das System, und der Realisierungsaufwand für die Unterstützung neuer Robotersteuerungen wird minimiert. Dazu ist ein Konzept notwendig, das es ermöglicht, den Aufbau und die Syntax einer Roboterprogrammiersprache zu formulieren.

5. Die Ausführung der steuerungsspezifischen Roboterprogramme in einem Simulationslauf soll es erlauben, die Umsetzbarkeit einer Planungsvariante zu prüfen und Informationen (z. B. Ausführungszeit) zu erhalten. Heutige CAR-Systeme können nur vereinzelt steuerungsspezifische Roboterprogramme direkt ausführen, stattdessen benötigen die CAR-Systeme die Programme in einer eigenen Simulationssprache [87]. Vor einer Ausführung der steuerungsspezifischen Roboterprogramme müssen deshalb die CAR-Systeme diese in die Simulationssprache übersetzen [37]. Die Realisierung solcher Übersetzer ist aufwendig [51, 129]. Um diesen Aufwand zu reduzieren, ist ein weiteres Ziel dieser Arbeit, ein Framework zu realisieren, das die Entwicklung solcher Übersetzer vereinfacht und beschleunigt.

Insgesamt soll das in dieser Arbeit entwickelte System einen Beitrag leisten, den Aufwand sowohl bei der Roboterprogrammierung von variantenreichen Produkten als auch bei der Planung optimierter Roboter-Fertigungszellen zu reduzieren und die möglichen Programmier- und Planungsfehler zu minimieren.

1.3 Aufbau der Arbeit

Nach diesem einleitenden Kapitel findet in Kapitel 2 zunächst eine Beschreibung und Beurteilung heutiger CAR-Systeme statt. Dabei werden die betrachteten Systeme unterschieden in Systeme, die für eine spezifische Fertigungstechnik (z. B. Schweißen, Montage) entwickelt wurden, und in Systeme, die unabhängig von einer Fertigungstechnik arbeiten. Danach erfolgt eine Betrachtung bisheriger Ansätze zur Lösung von Problemen bei der Planung und Optimierung von Roboter-Fertigungszellen.

Mit den in Kapitel 2 erzielten Erkenntnissen wird in Kapitel 3 eine Anforderungsanalyse durchgeführt, aus der sich die Anforderungen an ein System zur Generierung, Ausführung und Bewertung von Programmvarianten ableiten und formulieren lassen.

Kapitel 4 erläutert das Systemkonzept und skizziert kurz die notwendigen Systemmodule. Es wird gezeigt, wie das Systemkonzept die Anforderungen aus Kapitel 3 umsetzt und damit insgesamt erfüllt. Das Kapitel dient außerdem als Übersicht für die Kapitel 5-8 und führt in die Terminologie des Systemkonzepts ein.

Kapitel 5 erklärt den zentralen Bestandteil des Systemkonzepts, den in dieser Arbeit so genannten *Generierungsplan*, der für die Lösung eines konkreten Problems notwendig ist. Für den Entwurf der Generierungsplansprache werden die Kriterien formuliert und die getroffenen Entwurfsentscheidungen erläutert.

Kapitel 6 stellt die systeminterne Repräsentation zur Ausführung des Generierungsplans (*Steuergraph*) dar. Daran schließt sich die Beschreibung der Architektur des Systemmoduls *Generierungssteuerung* an, das die Pläne ausführt.

Kapitel 7 analysiert die Programmierkonzepte heutiger industrieller Robotersteuerungen. Aus diesen Erkenntnissen wird einerseits die *steuerungsneutrale Programmspezifikation* entwickelt, mit der der CAR-Anwender den Inhalt der Roboterprogramme ohne Kenntnis der Steuerungssyntax festlegt, und andererseits die *Syntaxdefinition* entworfen, die die Information über eine konkrete Steuerungssyntax enthält. Daran schließt sich die Beschreibung des Systemmoduls *Programmsynthese* an, das aus der Programmspezifikation und einer Syntaxdefinition ein steuerungsspezifisches Roboterprogramm erzeugt.

Kapitel 8 beleuchtet die Ausführung der steuerungsspezifischen Roboterprogramme in einem Simulationslauf. Im Einzelnen betrachtet Kapitel 8 das Framework zur Erstellung der Übersetzer von steuerungsspezifischen Roboterprogrammen, die Prüfung der Roboterprogramme und die Rückkopplung von Simulationsergebnissen. Letzteres erfordert die Bewertung der Roboterprogramme und eine Möglichkeit, daraus eine Verbesserung zu erzielen. Dazu werden bekannte Optimierungsverfahren für den hier notwendigen Einsatz beurteilt, ausgewählt und erweitert.

Kapitel 9 beschreibt Applikationen, die das weite Einsatzspektrum des Systems zeigen. Für die Programmierung von variantenreichen Produkten handelt es sich um den Einsatz des Systems in der Medizinrobotik und in der Porzellan verarbeitenden Industrie. Zur Beantwortung von Planungsfragen wird das System zur Optimierung einer exemplarischen Fertigungsstraße eingesetzt.

Kapitel 10 fasst die wichtigsten Ergebnisse dieser Arbeit abschließend zusammen.

2 Stand der Technik

In diesem Kapitel werden heutige Systeme für die Roboterprogrammierung und aktuelle Verfahren für die Optimierung des Robotereinsatzes betrachtet. Dabei wird eine Bewertung der Systeme und Verfahren hinsichtlich deren Anpassbarkeit zur Berücksichtigung von applikationsspezifischen Anforderungen und hinsichtlich deren Möglichkeit zur Generierung von Roboterprogrammvarianten vorgenommen. Besonderes Augenmerk wird auf die Erzeugung und Verwendung steuerungsspezifischer Roboterprogramme gelegt, weil nur dadurch gewährleistet werden kann, dass das Programmier- oder Planungsergebnis in der realen Roboter-Fertigungszelle umsetzbar ist.

2.1 Systeme zur Offline-Programmierung von Robotern

Wegen der starken Abhängigkeit der Roboterprogrammierung von der eingesetzten Fertigungstechnik (z. B. Schweißen, Lackieren) [23] werden neben den CAR-Systemen, die unabhängig von einer spezifischen Fertigungstechnik sind, auch Systeme betrachtet und beurteilt, die speziell auf eine solche Technik zugeschnitten sind.

2.1.1 Kommerzielle CAR-Systeme

In den letzten Jahren entstanden viele kommerzielle Programmiersysteme, die sich wie folgt einteilen lassen: Es existieren einerseits Systeme, die weder von einem Roboterhersteller noch von einer spezifischen Fertigungstechnik abhängig sind. Andererseits existieren Programmiersysteme, bei denen die Roboter eines spezifischen Herstellers oder eine spezifische Fertigungstechnik oder beides im Vordergrund stehen.

Herstellerunabhängige und fertigungstechnikunabhängige Programmiersysteme

Die Marktführer bei CAR-Systemen und damit die wichtigsten Vertreter sind die Systeme eM-Workplace™ (ehemals ROBCAD™) aus der eMPower™-Produktreihe [158] der Firma Tecnomatix Technologies Ltd. und das System IGRIP® der Firma DELMIA [36].

Beide Systeme ermöglichen die Roboterprogrammierung für unterschiedliche Hersteller. Beide Systeme verfolgen den Ansatz, die Roboter in einer eigenen Simulationssprache zu programmieren. Im System eM-Workplace™ heißt die Sprache TDL (Task Description Language) und im System IGRIP® heißt sie GSL (Graphic Simulation Language).

Neben den Simulationssprachen ist es möglich, die Roboterbewegung grafisch-interaktiv unter Zuhilfenahme von Dialogen zu erstellen. Der CAR-Anwender gibt die Roboterbewegung als Bahnen, d. h. Folgen von Bahnstützpunkten am Bildschirm, vor. Jeder Bahnstützpunkt stellt Attribute zur Verfügung, denen der CAR-Anwender Werte zuweist. Die Attribute lassen sich unterscheiden in steue-

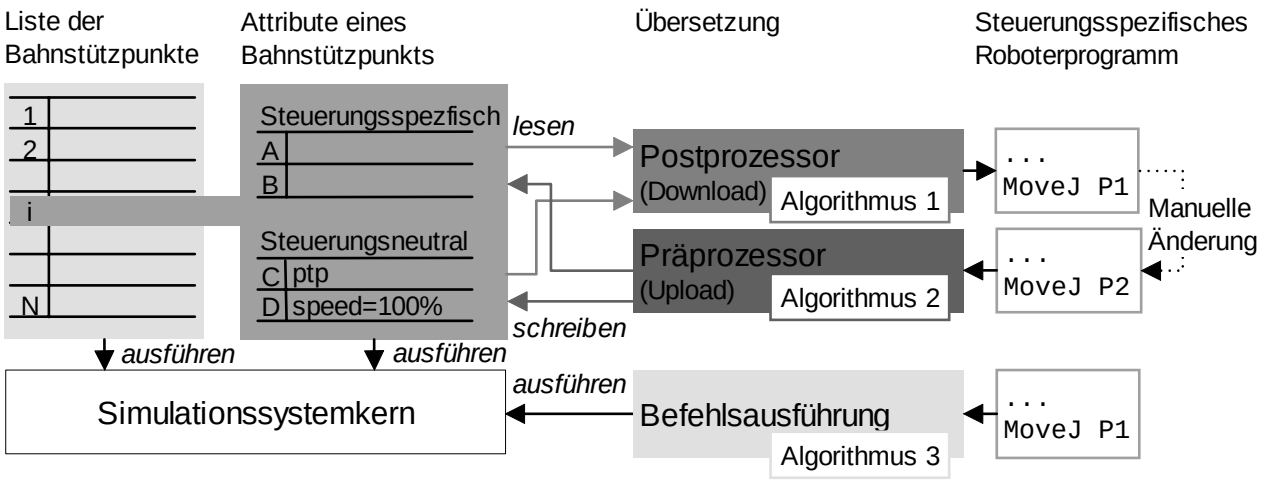


Bild 2.1: Steuerungsspezifische Programme in eM-Workplace™und IGRIP®

rungsneutrale und steuerungsspezifische Attribute [67, 129]. Die steuerungsneutralen Attribute enthalten alle Angaben für die Roboterausführung in einem Simulationslauf, z. B. die Interpolationsart und die Geschwindigkeit, mit der der Roboter den zugehörigen Bahnstützpunkt anfahren soll, das Setzen von Ausgängen oder Wartezeiten. Die steuerungsspezifischen Attribute enthalten die für das steuerungsspezifische Programm darüber hinaus benötigten Angaben, z. B. die Befehlssyntax.

Für die Erzeugung der steuerungsspezifischen Programme kommen in eM-Workplace™ und IGRIP® Postprozessoren zum Einsatz, die dazu die Attributwerte der Bahnstützpunkte lesen. Die Postprozessoren benötigen für jede Robotersprache einen Algorithmus zur Übersetzung (Bild 2.1). Der Vorteil dieses Postprozessoransatzes ist, dass bei Austausch des Roboters eines anderen Herstellers, der CAR-Anwender die Programmierung nicht wiederholen muss. Für den umgekehrten Weg existieren Präprozessoren, die das steuerungsspezifische Programm parsen und die entsprechenden Bahnstützpunkte mit den zugehörigen Attributwerten schreiben. In eM-Workplace™ sind diese Prozessoren in dem Modul eM-OLP™ als Perlskript implementiert. Eine Besonderheit bietet das System eM-Workplace™, indem es möglich ist, über eine interne Schnittstelle (Controller Modeling Language) einzelne Roboterbefehle auszuführen, z. B. Bewegungsbefehle [112]. Ein weiteres Perlskript kann damit ein steuerungsspezifisches Programm parsen und einzelne Roboterbefehle über die Controller Modeling Language während des Simulationslaufs ausführen.

Die Anwender dieser CAR-Systeme beklagen eine auffallend lange Dauer für die Erstellung der Programme, die sie in der Roboter-Fertigungszelle einsetzen wollen. Sie begründen dies mit den erforderlichen manuellen Änderungen an den von den Postprozessoren erstellten Roboterprogrammen. Die Anwender dieser CAR-Systeme müssen in der Regel Programmbefehle manuell ergänzen und können die so geänderten Roboterprogramme erst in der realen Roboter-Fertigungszelle testen [128].

Ein weiteres Problem ist die vordefinierte Struktur der steuerungsspezifischen Roboterprogramme, die die Attribute implizit vorgeben. Man kann die Werte der Bahnstützpunkte in der Roboter-Fertigungszelle verändern und die so veränderten Programme zurück in das System laden. Syntaktische Änderungen an den Programmen, wie sie in der Praxis üblich sind, können dazu führen, dass der Präprozessor die Programme nicht zurück in das System laden und dort ausführen kann. Problema-

tisch sind z. B. die Verwendung von Variablen, Kontrollanweisungen, Unterprogrammaufrufen und Ausdrücken [129]. Der Präprozessor kann, sofern es sein Parser zulässt, Programmanweisungen für steuerungsspezifische Funktionen als Attributwert in die Stützpunktliste schreiben. Die Befehlsfunktion bleibt aber im Simulationslauf unberücksichtigt [67]. Dasselbe gilt auch für Ausführung von Einzelbefehlen, die nicht in der Controller Modeling Language enthalten sind [112]. Eine umfassende Ausführung von steuerungsspezifischen Programmen ist in diesen Systemen somit nicht möglich.

Problematisch ist auch die Tatsache, dass die Roboterprogrammierung gute Kenntnisse über den Postprozessor erfordert, denn die Simulationssprache TDL und GSL sind in der Regel mächtiger als die Programmiersprache der Robotersteuerungen [129]. Ebenso ist die aufwendige Suche und Korrektur bei fehlerhaften Eintragungen in den Attributen und fehlerhaften oder unvollständigen Perlskripten der Prozessoren ein bekanntes Problem [121]. Solche Fehler werden in der Regel erst in der realen Roboter-Fertigungszelle entdeckt und können großen Schaden verursachen.

Beide Systeme stellen Programmiermodule für das Punktschweißen, das Bahnschweißen und das Lackieren zur Verfügung (eM-Spot™, eM-Arc™, eM-Paint™ im System eM-Workplace™ [158] und UltraSpot™, UltraArc™, UltraPaint™ im System IGRIP® [36]). Bei IGRIP® ist ein weiteres Programmiermodul UltraFinish™ für die Fertigungstechniken Entgraten, Schleifen und Polieren vorhanden. Diese Programmiermodule unterstützen den CAR-Anwender bei der Erzeugung der Bahnstützpunkte (Bild 2.1) für ihre jeweilige Fertigungstechnik, z. B. die Einhaltung eines vom CAR-Anwender angegebenen Winkels zwischen Brenner und Naht beim Bahnschweißen [121].

Keines der beiden Systeme ermöglicht dem CAR-Anwender eine Generierung steuerungsspezifischer Programme mit allen erforderlichen Programmbefehlen, sodass die Programme nicht ohne weitere syntaktische Änderungen auf der Steuerung einsetzbar sind [135, 140]. Die Philosophie der Programmiermodule ist die manuelle oder teilautomatische Erzeugung von Bahnstützpunkten unter Einhaltung von fertigungstechnischen Restriktionen. Der CAR-Anwender muss vor der Programmerzeugung die Stützpunkte erstellen und jedem Stützpunkt fertigungstechnische Informationen zuordnen, z. B. den Zustand der Schweißzange beim Punktschweißen oder die erforderliche Schweißspannung beim Bahnschweißen [128]. Er muss für die Programmierung einer neuen Produktvariante jeweils einen Großteil der Bahnstützpunkte und die darin enthaltenen fertigungstechnischen Informationen neu erstellen oder korrigieren. Folglich ist für die Programmierung einer neuen Produktvariante ein qualifizierter CAR-Anwender erforderlich. Ein weiterer Nachteil der Programmiermodule besteht in der fehlenden Änderbarkeit und Erweiterbarkeit, um die Module für andere Fertigungstechniken einsetzen zu können [34]. Insgesamt kann man festhalten, dass die automatische Generierung steuerungsspezifischer Programmvarianten unter Berücksichtigung applikationsspezifischer Anforderungen, die der CAR-Anwender festlegt, nicht möglich ist.

Weitere wichtige Vertreter von herstellerunabhängigen CAR-Systemen auf dem Markt sind: das System Grasp der Firma BYG Systems Ltd. [28], das System CimStation™ Robotics der Firma Silma/Adept Technology Inc. [149], das System COSIMIR® der Firma EF-Robotertechnik GmbH und des Instituts für Roboterforschung [116], das System Workspace5™ der Firma Flow Software Technologies [44], das System MOSES der Firma AUTOCAM Informationstechnik GmbH [12], das System MSM 2102 der Firma SL-Automatisierungstechnik GmbH [151], das System KISMET der Firma

Hans Wälischmiller GmbH, das System Robot3D der Firma FFSoft, das System Ropsim der Firma Camelot und das System EASY-ROB™ der Firma EASY-ROB [40].

Das System Grasp und das System CimStation™ Robotics sind von ihrem Programmierkonzept her vergleichbar mit den bereits betrachteten Systemen eM-Workplace™ und IGRIP® und besitzen dieselben Vor- und Nachteile [28, 87, 169].

Das System COSIMIR® verfolgt, im Gegensatz zu den bisher betrachteten Systemen, den Ansatz, die Roboter nicht in einer Simulationssprache zu programmieren, sondern in der jeweiligen steuerungsspezifischen Sprache [46, 135, 161]. Der Vorteil besteht darin, dass das geprüfte Programm mit dem auf der Steuerung ausgeführten Programm identisch ist. Die Gefahr, dass ein Postprozessor oder ein Präprozessor fehlerhaft ist, besteht nicht. Der CAR-Anwender kann steuerungsspezifische Eigenschaften (z. B. maximale Anzahl von Roboterpositionen) bereits in der Planungsphase berücksichtigen. Ein weiterer Vorteil besteht darin, dass man die Programme ohne syntaktische Einschränkungen von der Steuerung zurück in das System COSIMIR® laden und dort ausführen kann. Ein Nachteil ist der hohe Aufwand für die Unterstützung einer spezifischen Robotersprache [51, 129].

Für das System COSIMIR® ist ein Programmiermodul für die Fertigungstechniken Entlacken verfügbar [135]. Der CAR-Anwender kann durch die Angabe von fertigungstechnischen Parametern, z. B. maximal zulässiger Abtrag, steuerungsspezifische Programme erzeugen. Dabei ist es möglich, Programme für Produktvarianten erzeugen zu lassen [135]. Allerdings sind für einen Einsatz der Programme auf einer realen Robotersteuerung in der Regel weiterhin manuelle Modifikationen notwendig, da die Struktur der erzeugten Programme starr vorgegeben ist. Somit ist ein erfahrener CAR-Anwender erforderlich. Des Weiteren kann der CAR-Anwender die Vorschrift zur Berechnung der Roboterprogramme nicht verändern, sodass neue Anforderungen, z. B. der Einsatz einer anderen Fertigungstechnik, nicht berücksichtigt werden können.

Ein weiterer Vertreter, der für die Roboterprogrammierung direkt die steuerungsspezifische Sprache einzelner Hersteller (ABB, Motoman) verwendet, ist das System Workspace5™ [44]. Es existieren Programmiermodule für die Fertigungstechniken Punkt- und Bahnschweißen sowie das Lackieren. Eine Besonderheit bietet das System Workspace5™ durch die Integration von Visual Basic® for Applications (VBA) für kundenspezifische Anpassungen des Simulationsverhaltens von Fertigungskomponenten (z. B. neue Roboterkinematiken, kundenspezifische Werkzeuge) [11]. Eine Generierung von Roboterprogrammvarianten für unterschiedliche Hersteller ist mit dem System Workspace5™ nicht möglich, weil man für die Programmierung eines Roboters ein herstellerspezifisches Programmiermodul verwenden muss. Ein herstellerunabhängiges Programmiermodul existiert nicht [103].

Das System MOSES der Firma AUTOCAM Informationstechnik ist in das CAD-System AutoCAD® [5] integriert. Für MOSES existieren Programmiermodule mit verschiedenem Automatisierungsgrad für diverse Fertigungstechniken [22, 23, 59, 140]. Zur vollautomatischen Programmgenerierung existieren Programmiermodule für das Profilschneiden von Normprofilen im Schiffbau (MOSES-Profil) und das Plasmaschneiden an Blechen (MOSES-Blech). Des Weiteren existieren Programmiermodule zur teilautomatischen Programmierung für Plasmaschneidaufgaben an 3D-Bauteilen (MOSES-Cut), für Schneidaufgaben an Rohren (MOSES-Rohr), für die Beschickung von

Biegepressen (MOSES-Press) und für das Bahnschweißen (MOSES-Arc). Der CAR-Anwender kann die Rechenvorschriften in diesen Programmiermodulen nicht ändern, um applikationsspezifische Anforderungen zu berücksichtigen. Auf den Programmaufbau hat der CAR-Anwender erst dann Einfluss, wenn die Programmiermodule die Programme ausgegeben haben. Das System erzeugt die Roboterprogramme in der internen Simulationssprache MPL [21, 140], die vergleichbar mit der Robotersprache VAL-II ist [25]. Die Prüfung der generierten Programme auf Ausführbarkeit durch den Roboter erfolgt mithilfe eines Simulationslaufs der Programme in der Syntax von MPL. Bevor das Programm in der Roboter-Fertigungszelle einsetzbar ist, muss ein Postprozessor das Programm konvertieren [21]. Dadurch bleiben steuerungsspezifische Eigenschaften unberücksichtigt und Fehler im Postprozessor findet man erst, wenn man das Programm auf der realen Robotersteuerung einsetzt.

Innerhalb des Systems MSM 2102 ist es ebenfalls möglich, steuerungsspezifische Roboterprogramme auszuführen [151]. Allerdings sind die Befehlssätze der Sprachen stark eingeschränkt. Programmiermodule sind nicht verfügbar, sodass eine automatische Programmgenerierung fehlt.

Die Systeme KISMET, Robot3D und Ropsim beruhen ebenfalls auf dem Ansatz, das Roboterprogramm in einer neutralen Sprache zu programmieren und für die Erzeugung der steuerungsspezifischen Programme Postprozessoren einzusetzen. Somit können Letztere erst in der realen Roboter-Fertigungszelle getestet werden. Das System Ropsim bietet die Möglichkeit, Bahnstützpunkte aus CAD-Daten der Werkstücke zu erhalten. Bei variantenreichen Produkten muss der CAR-Anwender aber den Programmiervorgang jeweils wiederholen.

Das System EASY-ROB™ ist ein Robotersimulationssystem, in dem der CAR-Anwender den Roboter in der internen Simulationssprache ERPL (EASY-ROB™ Programming Language) programmiert. Postprozessoren für die gängigen Hersteller sind nicht verfügbar. Somit kann das System EASY-ROB™ keine steuerungsspezifischen Programme ausführen. Programmiermodule werden über das im Folgenden betrachtete System FAMOS robotic® zur Verfügung gestellt [29, 40, 135].

Im Folgenden werden noch herstellerunabhängige Offline-Programmiersysteme betrachtet, die keine Möglichkeit zur Simulation bieten. Zu den wichtigsten Vertretern gehören das System RobotWorks [11, 32] der Firma Compucraft Ltd. und das System FAMOS robotic® [29] der Firma carat robotic innovation GmbH. Beide sind als werkstückorientierte Programmiersysteme ausgelegt. Der Anwender erstellt Roboterbahnen interaktiv unter Zuhilfenahme von Geometriedaten des Werkstücks.

Das System RobotWorks ist in das CAD-System SolidWorks® [5] integriert und bietet dadurch den Vorteil der komfortablen Bearbeitung von Geometriedaten. Das System kann lediglich Stützpunktelisten für die Roboterbahnen erstellen. Der CAR-Anwender kann keine ausführbaren Programme erstellen [11], die Bewegungsbefehle, Befehle zur Festlegung von Bahngeschwindigkeiten oder Befehle für die E/A-Ansteuerung der Werkzeuge enthalten.

Das System FAMOS robotic® besitzt als einziges System die Möglichkeit, kundenspezifische Anforderungen bei der Berechnung der Roboterbahnen zu berücksichtigen. Mit diesem System FAMOS robotic® wurde z. B. ein angepasstes System für die Schuhfertigung (DEScom®) realisiert. Das System FAMOS robotic® erzeugt die steuerungsspezifischen Programme mithilfe von Postprozessoren. Die Prüfung auf Ausführbarkeit der Roboterprogramme findet durch das System EASY-ROB™ statt.

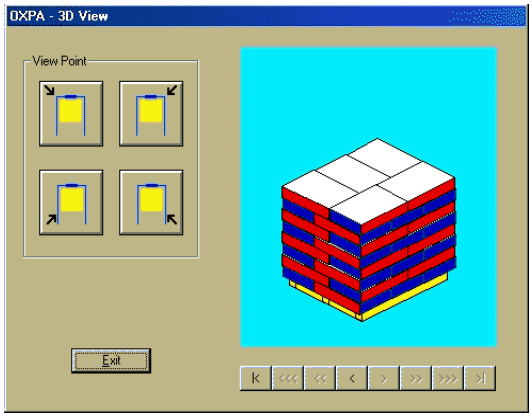


Bild 2.2: Grafisch-interaktive Benutzeroberfläche bei OXPA (Quelle: Columbia/Okura, Kanada)

Das System FAMOS robotic® gibt die Programme in der systeminternen Syntax von EASY-ROB™ aus. Somit wird nicht das steuerungsspezifische Programm geprüft. Dessen Fehler findet man erst beim Einsatz auf der realen Robotersteuerung.

Man kann festhalten, dass der CAR-Anwender für die Programmierung variantenreicher Produkte sowohl im System FAMOS robotic® als auch im System RobotWorks die Schritte für die Erstellung des Programms (FAMOS robotic®) oder der Stützpunktliste (RobotWorks) manuell wiederholen muss. Des Weiteren ist die Ausführbarkeit der steuerungsspezifischen Programme aufgrund der fehlenden Möglichkeit diese zu simulieren nicht gewährleistet.

Fertigungstechnikabhängige Roboterprogrammiersysteme

Das Offline-Programmiersystem Roboplan NT der Firma NIS GmbH ist spezialisiert auf die Fertigungstechnik Bahnschweißen [114, 115]. Dieses System enthält eine wissensbasierte Komponente, die die Brennerbewegung und die Schweißparameter für eine festgelegte Schweißnaht ermittelt und daraus das Roboterprogramm mit entsprechenden Transferbewegungen erzeugt. Eine besondere Eigenschaft dieses Systems ist die Möglichkeit zur Programmierung von Schweißaufgaben für Baugruppen aus dem Handelsschiffbau. Das bedeutet, dass die Programmierung von Produktvarianten für dieses Einsatzgebiet möglich ist. Die eingesetzte Vorschrift zur Berechnung der Programme kann der CAR-Anwender aber nicht ändern [135]. Deshalb und wegen des Bezugs zum Bahnschweißen ist der Einsatz für andere Fertigungstechniken nicht möglich [140].

Weitere Systeme, die auf die Programmierung von Schweißapplikationen beschränkt sind, sind das System ROBCON 2000 der Firma World Wise Technologies Inc. und das System act™/weld der Firma Alma [11] und das System ROBOMAX von Robotics Laboratory [127]. Die Programmiermöglichkeiten sind vergleichbar mit dem Programmiermodul eM-Arc™ des Systems eM-Workplace™ oder mit dem Programmiermodul UltraArc des Systems IGRIP®. Mit dem System act™/weld kann der CAR-Anwender die erzeugten, steuerungsspezifischen Roboterprogramme vor dem Einsatz auf der Robotersteuerung nicht prüfen. Bei beiden Systemen muss der CAR-Anwender für eine Programmierung von Produktvarianten zahlreiche manuelle Schritte für jede Produktvariante wiederholen.

Zur Programmierung von Palettieraufgaben existieren ebenfalls Systeme, die sich allerdings auf Roboter eines Herstellers beschränken. Dies sind z. B. das System VisualPallet® der Firma Motoman, das System PalletTool® der Firma Fanuc Robotics [134] und das System OXPA der Firma Columbia/Okura LLC. Das Besondere an OXPA ist die Integration von Programmiersystem und Robotersteuerung für einen Palettierroboter. Aus Sicht des Anwenders gibt es bei dieser Robotersteuerung keine Roboterprogramme. Über eine grafisch-interaktive Benutzeroberfläche teilt man dem Roboter die zu erledigende Aufgabe mit (Bild 2.2). Dabei kann man u. a. spezielle Griffe, Reihenfolgen und Fügerichtungen vorgeben. Der Anwender kommt nicht mit der Steuerungssyntax in Kontakt. Diese Art der Programmierung ist nur durch den starken Bezug zum Anwendungsbereich möglich.

Neben diesen Systemen gibt es zahlreiche Individuallösungen, d. h. spezielle Entwicklungen für einen Problembereich und in der Regel für einen Kunden, z. B. [9], [90], [111], [173]. Nachteilig ist der eingeschränkte Einsatzbereich und an der mangelnden Anpassungsfähigkeit für andere Applikationen.

Herstellerabhängige Simulations- und Programmiersysteme

Die bekanntesten Vertreter von herstellerabhängigen CAR-Systemen sind: das System RobotStudio™ der Firma ABB Flexible Automation GmbH [7], die Systeme KR SIM und KUKA Sim der Firma KUKA Roboter GmbH [93, 94], das System INVISION der Firma intro Industrieautomation GmbH [80], das System V_CAT der Firma Stäubli Unimation Inc. [156], das System ROTSY und das System MotoSim der Firma Motoman Inc., das System SimPRO der Firma FANUC Robotics, das System Nachi Robot Simulator der Firma Nachi-Fujikoshi, das System Pana Robot DTPS der Firma Panasonic Factory Automation und das System VisualStacker der Firma Salvagini.

Die Unterstützung zur Programmerzeugung ist vergleichbar mit den herstellerunabhängigen CAR-Systemen, z. B. existieren in RobotStudio™ Funktionen zur Erzeugung von Bahnstützpunkten und für SimPRO existieren Module für das Bahnschweißen (WeldPRO™) und das Lackieren (PaintPRO™).

Für alle Systeme gilt, dass die Programmierung an die herstellerspezifische Sprache gebunden ist. Der Vorteil dieser Systeme besteht in der Möglichkeit zur Berücksichtigung steuerungsspezifischer Merkmale bereits in der Planungsphase der Roboter-Fertigungszelle. Ein weiterer Vorteil besteht in der Fähigkeit, die steuerungsspezifischen Programme nach deren Veränderung in der Roboter-Fertigungszelle zurück in das CAR-System laden zu können. Bei einigen Systemen (z. B. KR SIM, INVISION) wird die reale Robotersteuerung samt Hardware an das Simulationssystem gekoppelt [80, 93]. Andere Hersteller koppeln ausschließlich die Steuerungssoftware an ihr CAR-System (z. B. RobotStudio™, KUKA Sim). Die Vorteile, die sich durch die Verwendung der realen Robotersteuerung oder der realen Steuerungssoftware ergeben, bestehen in der Verwendung derselben Bahnplanungsalgorithmen, der nahezu vollständigen Verwendung aller Steuerungsfunktionen und der Erkennung von fehlerhaften Steuerungsfunktionen in einem Simulationslauf. Bei der Kopplung der realen Robotersteuerung ergibt sich der Nachteil der hohen Hardwarekosten und der aufwendigen Handhabung beim täglichen Gebrauch (z. B. lange Zeiten für das Laden der Programme in die Steuerung). Ein weiterer Nachteil bei allen Systemen besteht in der fehlenden Möglichkeit, Roboter unterschiedlicher Hersteller für einen konkreten Einsatz zu programmieren und zu vergleichen.

2.1.2 Systeme aus dem Forschungsbereich

In den letzten Jahren wurde eine große Anzahl von wissenschaftlichen Systemen zur Roboterprogrammierung entwickelt, bei denen eine spezifische Fertigungstechnik im Vordergrund steht. Diese werden im Folgenden betrachtet. Eine Unterscheidung zwischen herstellerunabhängigen Systemen und herstellerabhängigen Systemen wie im vorangegangenen Abschnitt erfolgt hier nicht, weil die Systeme in der Regel für Roboter eines Herstellers entwickelt wurden.

Programmiersysteme für das Bahnschweißen

Grundlegende Forschungsarbeiten über die Roboterprogrammierung für das Bahnschweißen stammen von HAMMES [64], LIEBENOW [98], ZABEL [174] und SCHMID [141]. Sie entwickelten werkstattorientierte Programmiersysteme, die auf Basis von Makros den Programmieraufwand reduzieren. Die vordefinierten Makros repräsentieren Schweißbilder, d. h. charakteristische Verläufe von Schweißnähten, die der Anwender für eine konkrete Programmieraufgabe zusammenstellt. Ein Makro entspricht einer beliebigen Sequenz an Programmbefehlen. Der CAR-Anwender kann die Makros ändern. Dabei beruhen die zur Verfügung gestellten Makros auf einem eingeschränkten Werkstückspektrum. Mithilfe solcher Makros ist es möglich, eine große Anzahl von praxisrelevanten Schweißaufgaben zu programmieren [128, 131]. LUSZEK [102] und ZANDER [19, 153] stellen Funktionen für die Schweißablaufplanung und für den Sensoreinsatz vor, um die Qualität der Roboterprogramme zu erhöhen und den Programmierer von Routinearbeiten zu entlasten. HOLLENBERG [71] ermöglicht erstmals die Parametrierung der Makros mit Werten aus CAD-Daten. Um die Anwendbarkeit der Makros auf das Werkstückspektrum zu erhöhen, entwickelte HOLLENBERG ein Verfahren, um topologische Gemeinsamkeiten in den CAD-Daten unterschiedlicher Baugruppen zu vergleichen. Damit hat sein System die Fähigkeit, Roboterprogramme für bereits programmierte Baugruppen im Handelsschiffbau auf neue Baugruppen zu übertragen. Allerdings ist die zulässige Struktur der CAD-Daten starr und eingeschränkt [140]. Durch diesen Ansatz ist es möglich, Roboterprogramme für Bauteilvarianten ohne großen Aufwand für Schweißaufgabe automatisch zu generieren. Die Arbeiten von HOLLENBERG beeinflussten die Systementwicklung von Roboplan NT (S. 12).

QUARTIER [131] stellt das System MacroWeld vor, das ebenfalls auf dem Einsatz von Makros beruht. Die Makros kann man mit Werten aus den CAD-Daten der Werkstücke parametrieren.

HARTFUSS [66, 142] stellt eine wissensbasierte Komponente zur Roboterprogrammierung vor. Diese Komponente enthält Expertenwissen für Schutzgasschweißaufgaben im Stahlhochbau. Damit kann man auf einem spezifischen Werkstückspektrum verschiedene Bereiche der Schweißprozess- und Schweißfolgeplanung durchführen.

HUMBURGER [78] beschreibt einen umfassenden Ansatz, der das schweißtechnische Wissen eines Facharbeiters in Form einer Blackboard-Architektur [37] modelliert. Für den beispielhaften Systemeinsatz stellt er die Programmierung von Bahnschweißaufgaben vor. Dabei gehen überwiegend die Arbeiten von LIEBENOW [98] und LUSZEK [102] ein.

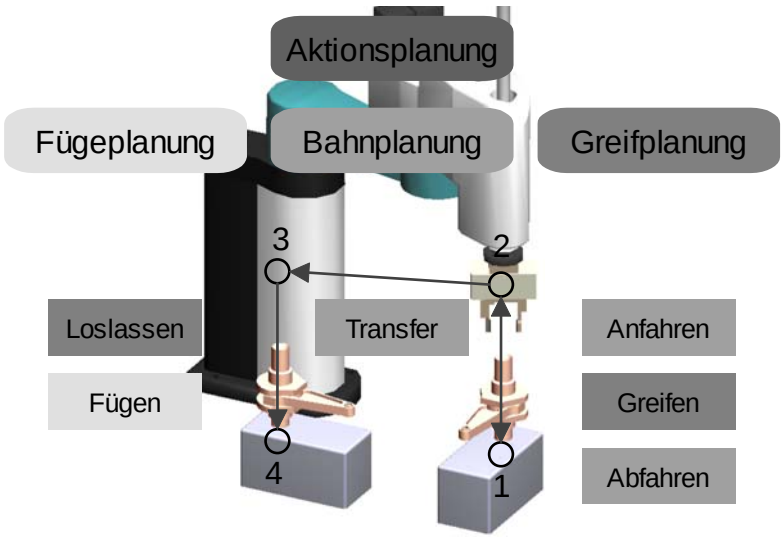


Bild 2.3: Teilprobleme bei Handhabungs- und Montagevorgängen

Einen anderen Ansatz verfolgt PEPER [128]. Sein System PROARC [117] soll den Facharbeiter mit seinem nur aufwendig zu modellierenden Wissen von der Programmierung entlasten, indem es eine strukturierte Modellierung der Schweißaufgabe ermöglicht. Dabei beruht die Programmgenerierung auf einem so genannten Schweißplans. PEPER schränkt das Werkstückspektrum und die Beschreibungsmöglichkeiten der Schweißaufgabe ein, um die Systemkomplexität zu verringern.

Die neuesten Programmiersysteme für das Roboterschweißen setzen Techniken der visuellen Programmierung ein, z. B. das System ROBICON [143] oder das System PIN von DAI [33] und KAMPKER [82, 83]. Letzteres verwendet ebenfalls Makros, die es dem CAR-Anwender als Symbole darstellt. Der CAR-Anwender programmiert die Schweißaufgabe durch Anordnen dieser Symbole. Hinter jedem Symbol verbirgt sich eine textuelle Beschreibungsform der Makrofunktion.

Programmiersysteme für die Montage

Zahlreiche Forschungsvorhaben beschäftigen sich mit der automatisierten Roboterprogrammierung für Handhabungs- und Montagevorgänge. Dabei tauchen komplexe Teilprobleme auf, für die es wiederum zahlreiche Forschungsarbeiten gibt. Bei der Programmierung von Handhabungs- und Montagevorgängen muss man die Problembereiche der Aktionsplanung, der Bahnplanung (Grobbewegungen), der Greifplanung und der Planung von Fein- und Fügebewegungen betrachten (Bild 2.3). Die Aktionsplanung erzeugt eine Beschreibung der Montageaufgabe durch eine Folge von Anweisungen auf hohem Abstraktionsniveau (implizite Roboterprogrammierung). Die Greifplanung ist für die Bestimmung eines Griffs am Werkstück unter Berücksichtigung von Erreichbarkeit, Stabilität und Werkstücklage beim Öffnen des Greifers verantwortlich. Die Bahnplanung plant eine kollisionsfreie Anfahr-, Transfer- und Abfahrbewegung durch den Raum. Die Fein- und Fügebewegungen liefern unter Einsatz von Sensorik eine Bewegung, sodass der Roboter das Werkstück am Zielort ablegt.

Keines der beschriebenen Systeme war in der Lage in einer realen Fertigungsumgebung automatisch Programme für einen Montageroboter zu generieren [78]. Weitere Systeme werden u. a. in OLSCHEW-

SKI [120], FREUND [45], REINISCH [132], WECK UND PEPER [167], HECK [85], NNAJI [76, 119], WEEKS [168], ONORI [41, 121, 122], KUGELMANN [91] und MOSEMANN [108] beschrieben. Es existieren viele Forschungsarbeiten bezüglich der kollisionsfreien Bahnplanung von Transferbewegungen, u. a. bei LOZANO-PEREZ [100], HÖRMANN [74], GLAVINA [60], ROSSMANN [137], HENRICH [68] und ROKOSSA [135]. Auf dem Gebiet der Planung von Fein- und Fügebewegungen existieren u. a. Arbeiten von LAUGIER [96], REINISCH [132] und HÖRMANN UND WERLING [75]. Auf dem Gebiet der Greifplanung sind die Arbeiten von HUCK [77], STETTER [155] und RÖHRDANZ [133] zu erwähnen. Einen Gesamtüberblick liefert SHIMOGA [147]. Das Gebiet der Aktionsplanung wird u. a. in LEVI [97], HUCK [77], PARK [126] und KERNEBECK [85] untersucht.

Eine Programmgenerierung eines konkreten Montagevorgangs erfordert Vereinfachungen, da sonst die Komplexität der einzelnen Probleme nicht mehr beherrschbar ist [78]. Die Komplexität für die Lösung der einzelnen Problembereiche hängt von zahlreichen Faktoren ab (z. B. Roboterkinematik, Greifertyp, Repräsentation der Werkstückgeometrie). Gerade diese Vereinfachungen können dazu führen, dass applikationsspezifische Anforderungen nicht mehr umsetzbar sind.

Programmiersysteme für das Entgraten

Charakteristisch für die Programmierung eines Entgratvorgangs ist die Aufteilung in zwei Phasen und der Einsatz von Sensorik. In der ersten Phase fährt der Roboter zunächst eine programmierte Sollkontur ab, auf der man Grate erwartet. Bei dem Abfahren der Sollkontur erfassen Sensoren die Grate. Dabei kommt der Roboter als Messmaschine zum Einsatz [129]. In der zweiten Phase fährt der Roboter die korrigierte Bahn ab, um die Grate zu beseitigen. Diese zweite Bahn wird aus den sensorisch erfassten Graten und weiteren Parametern für die Beschreibung des Entgratvorgangs berechnet (z. B. Materialeigenschaften). Bisherige Arbeiten auf diesem Gebiet stammen u. a. von BOLEY [26], BASTERT [15], HAMURA UND MIZUNO [65], LIU [99], JUNG [81] und PERSOONS [129]. Die Arbeiten unterscheiden sich in der Art und Weise, wie die Sollkontur ermittelt wird, welche Sensorik zum Einsatz kommt, und wie die Bahn zu Gratbeseitigung bestimmt wird. Für die Programmierung der Sollkontur wird entweder das Teach-In-Verfahren [26, 65, 99] verwendet, oder die Sollkontur wird aus CAD-Daten berechnet [15, 81, 129]. Für Letzteres ist in der Regel ein Abgleich mit einem entgrateten Musterwerkstück notwendig [81]. Zum Einsatz kommende Sensorik sind taktile Sensoren [15, 26], Kraft-/Momentensensoren [65, 81, 129] und Bildverarbeitungssysteme [99]. Die Bestimmung der Bahn zur Gratbeseitigung erfolgt jeweils über eine eigene, systeminterne Rechenvorschrift (z. B. WAGNER [164]). In PERSOONS [129] findet sich ein Überblick über unterschiedliche Rechenvorschriften. Sie unterscheiden sich u. a. in der Geometrie des Entgratwerkzeugs, der geometrischen Repräsentation der Grate und der eingesetzten Sensorik zur Gratermittlung.

Sowohl die Programmierung der Bewegung zum Abfahren der Sollkontur als auch die Berechnung der Bahn zur Gratbeseitigung erfordern einen hohen Programmieraufwand. Hinzu kommt, dass die Rechenvorschrift zur Gratbeseitigung in der Regel applikationsspezifisch ist, z. B. aufgrund des eingesetzten Entgratwerkzeugs. Eine Anpassung der Rechenvorschrift ist bei den betrachteten Systemen nicht (z. B. WAGNER [164]) oder nur eingeschränkt möglich (z. B. PERSOONS [129]).

Programmiersysteme für das Lackieren

Für die Generierung von Roboterprogrammen für das Lackieren wurden ebenfalls Programmiersysteme entwickelt. Dazu zählen u. a. die Arbeiten von IMAM [79], ROKOSSA [135] und PERSOONS [129]. Auch beim Lackieren dient zur Berechnung der Roboterbahnen eine entsprechende Vorschrift, deren Eingangsparameter der Anwender geeignet setzen muss [78]. Weitere Eingangsdaten für die verwendeten Rechenvorschriften sind u. a. die CAD-Daten des zu bearbeitenden Werkstücks und ein Modell der Lackierpistole.

Auch bei diesen Systemen ist es nicht möglich, die Rechenvorschrift für applikationsspezifische Anforderungen anzupassen.

Programmiersysteme für weitere Fertigungstechniken

GRUBE [61] führte grundlegende Arbeiten für das Schleifen durch. Das Ergebnis ist eine Rechenvorschrift, die für die Roboterprogrammierung einsetzbar ist. GRUBE untersuchte Einflussgrößen, z. B. Eigenschaften des Schleifbands und der Werkstückgeometrie. GRUBE und verifizierte diese Einflussgrößen empirisch. Die erzeugten Programme enthalten im Wesentlichen die Sollbahnen des robotergeführten Werkstücks an einer Bandschleifmaschine. Die Ausführbarkeit des Programms ist nicht gewährleistet, da das System keine kinematischen Restriktionen (z. B. Erreichbarkeit) prüft [135].

Für das Brennschneiden entwickelte RUDLOFF [140] und für das Plasmaschneiden entwickelte FUNDER [59] ein entsprechendes Programmiersystem. Ein weiteres Programmiersystem, das speziell für Profile im Schiffbau einsetzbar ist, stammt von BICKENDORF [22, 23]. Diese wissenschaftlichen Arbeiten beeinflussten die Entwicklung des Systems MOSES (S. 10). Dabei kann ein Anwender dieser Programmiersysteme die von den Systemen verwendeten Rechenvorschriften, die zur Berechnung der Roboterbahnen führen, nicht ändern, um spezifische Anforderungen zu berücksichtigen.

Für Klebeprozesse entwickelte PETRY [130] ein Programmbaukasten, der die Möglichkeit bietet, über Makros den Klebvorgang zu programmieren. Hierbei identifiziert PETRY für das Kleben spezifische Bahnabschnitte und stellt dem Programmierer Makrobibliotheken zur Verfügung.

ROSELL ET AL. [136] entwickelten ein Programmiersystem für das Polieren. Die Eingangsgrößen für die Programmerzeugung sind u. a. die Sollbahn samt Geschwindigkeiten am Werkstück, die Sollkräfte entlang der Bahn, die zu verwendende Polierstation und ein geometrisches Zellenmodell. Das Ziel ist die Generierung eines optimierten Roboterprogramms bezüglich der folgenden Kriterien: der Greifpunkt am Werkstück, die Richtung der Bahn, die Roboterkonfiguration für jeden Bahnverlauf und die Reihenfolge der einzelnen Bahnverläufe. Dabei verwendet das System für die Bestimmung der Transferbewegungen vereinfachende Annahmen. Die Ausgabe des Systems ist ein Programm in der Syntax von V+ [8] für die Durchführung der Polieraufgabe auf einem Stäubli Roboter RX 90.

Ein Offline-Programmiersystem für das Profilieren von Reifen durch einen Roboter stellt DOU [38] vor. Die Bewegungsprogramme werden dabei auf Basis von CAD-Daten und zusätzlichen Parametern für den Profiliervorgang teilautomatisch generiert. Die eingesetzte Rechenvorschrift zur Generierung der Programme ist starr und der Anwender kann diese nicht ändern.

Ein System zur Generierung von Roboterprogrammen für die Bestückung von Leiterplatinen stellen VETORAZZI und TELLES [163] vor. Das System entnimmt aus CAD-Daten der Platinen und weiteren Dateien ausschließlich die Koordinaten der Bahnstützpunkte. Kollisionsbetrachtungen finden nicht statt. Das generierte Programm besteht nur aus einer Hauptroutine und enthält die Befehle für die notwendigen Pick-and-Place-Bewegungen. Weitere Steuerungsfunktionen können nicht in das Programm aufgenommen werden.

Ein System für die Programmierung von Großrobotern zur Reinigung von Verkehrsflugzeugen stellt MEISSNER [105] vor. Dabei löst das System Probleme bezüglich der Bahn-, Konfigurations- und Bewegungsplanung. Der Anwender kann zwischen verschiedenen Reinigungsstrategien wählen.

Die in diesem Abschnitt vorgestellten Systeme sind alle auf ein bestimmtes Anwendungsgebiet zugeschnitten. In Bezug auf eine Roboterprogrammierung für variantenreiche Produkte muss der Anwender die Schritte zur Programmerstellung manuell wiederholen. Applikationsspezifische Anforderungen können nur eingeschränkt berücksichtigt werden.

Fertigungstechnikunabhängige Programmiersysteme

Wissenschaftliche Arbeiten auf dem Gebiet der fertigungstechnikunabhängigen Programmiersysteme beruhen auf der Programmiermethode "Programming by Demonstration", z. B. MYERS [109, 110], FRIEDRICH ET. AL. [55] und ROKOSSA [135]. Dabei macht der Anwender die zu programmierende Bewegung in einer realen oder virtuellen Umgebung vor. Die Systeme zeichnen die Bewegung auf und versuchen diese durch Roboterbefehle anzunähern [110]. Schließlich fassen die Systeme die gefundenen Befehle in einem Roboterprogramm zusammen. Vorteilhaft ist die einfache Bedienbarkeit bei der Programmierung der Roboterbewegung, weil keine Kenntnis der Robotersprache erforderlich ist [110]. Benötigt das Programm aber steuerungsspezifische Funktionen, so ist eine manuelle Modifikation der Programme erforderlich. Dies führt zu einer mangelnden Flexibilität, wenn die Programmierung wiederholt werden muss [55], z. B. für variantentenreiche Produkte. Außerdem können fertigungstechnische Restriktionen, z. B. das Einhalten eines Anstellwinkels beim Bahnschweißen, schlechter berücksichtigt werden als bei einer Berechnung der Bahnstützpunkte.

2.2 Verfahren für die Optimierung des Robotereinsatzes

Bei der Planung von Roboter-Fertigungszellen ergeben sich komplexe Probleme, insbesondere wenn ein optimiertes Verhalten angestrebt wird. In diesem Abschnitt werden Planungsprobleme und Lösungsverfahren diskutiert und bezüglich ihrer Umsetzbarkeit und Zuverlässigkeit beurteilt. Für eine erfolgreiche Umsetzung muss das Lösungsverfahren einerseits die Fähigkeit besitzen, applikationsspezifischer Anforderungen zu berücksichtigen und andererseits muss es eine Lösung mit einem steuerungsspezifischen Roboterprogramm prüfen.

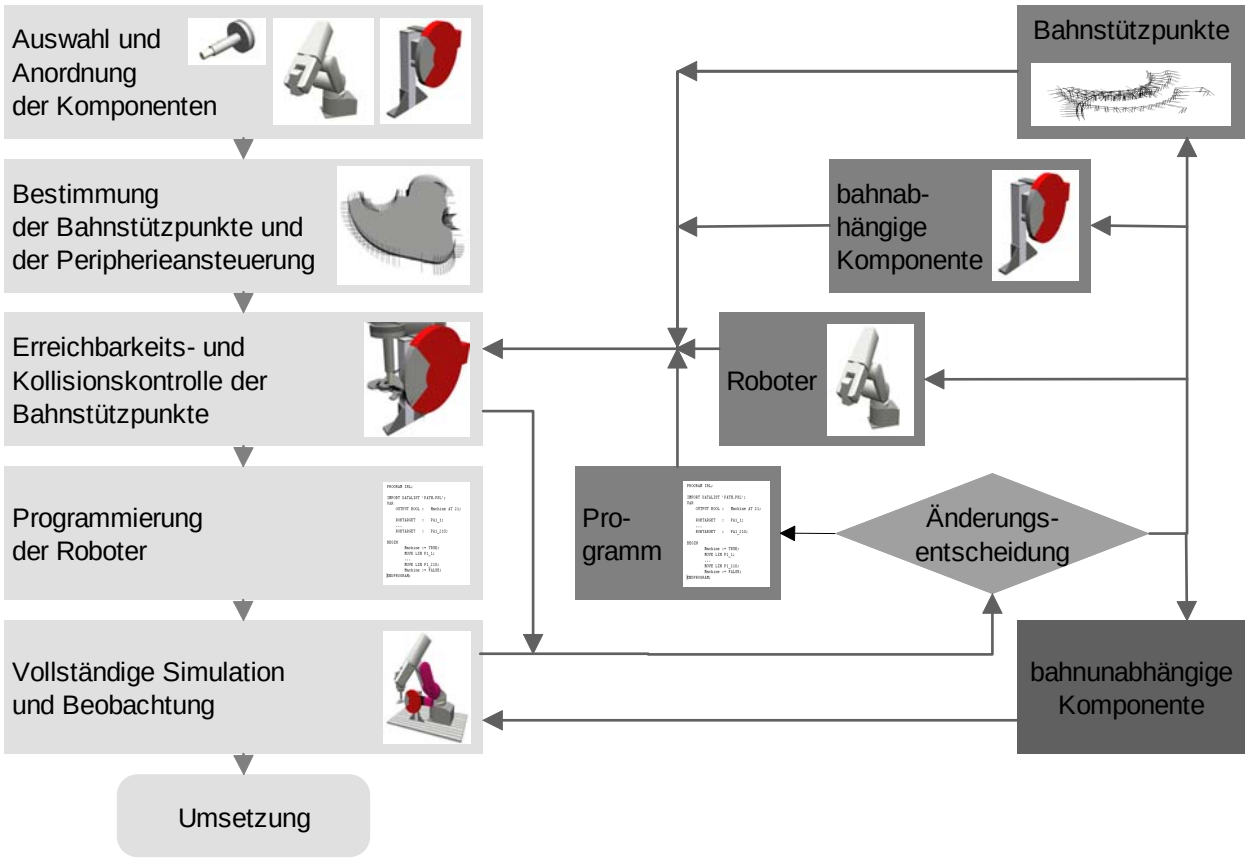


Bild 2.4: Vorgehensweise bei der Planung und Optimierung mit heutigen CAR-Systemen

2.2.1 Verfahren in kommerziellen CAR-Systemen

Eine umfassende Zellenplanung oder Zellenoptimierung (z. B. die Auswahl und Platzierung von Fertigungskomponenten) mithilfe heutiger kommerzieller CAR-Systeme ist gekennzeichnet durch eine manuelle iterative Vorgehensweise [20, 101] (Bild 2.4).

Der CAR-Anwender sucht zuerst für seine applikationsspezifischen Anforderungen geeignete Fertigungskomponenten (z. B. Roboter, Werkzeug) aus. Dabei ist er auf seine Erfahrung angewiesen. Der CAR-Anwender muss bei der Roboterauswahl u. a. die Erreichbarkeit der anzufahrenden Bahnstützpunkte, die einzuhaltende Zykluszeit und die maximalen Kosten für den Robotereinsatz berücksichtigen. Für die ausgewählten Komponenten erstellt der CAR-Anwender dann ein anfängliches Layout.

Die Roboterbahnen sind in der Regel implizit durch das zu bearbeitende Werkstück vorgegeben (z. B. die Bahn entlang eines Metallplattenrands in Bild 1.1, S. 2). Der CAR-Anwender gibt diese Bahnen als Folge von Stützpunkten vor. In Abhängigkeit von der einzusetzenden Fertigungstechnik und der Werkstückgeometrie kann die Zahl der Stützpunkte hoch sein (z. B. Polieren, Schleifen, Wasserstrahlschneiden) [11]. Bedingt durch die große Anzahl von Stützpunkten, die zusätzlich Fertigungsrestriktionen, z. B. der Anstellwinkel beim Bahnschweißen, einhalten müssen, verursacht deren Bestimmung einen hohen Aufwand. Eine weitere Schwierigkeit für den CAR-Anwender ist, sich die Lage der Bahnstützpunkte vorzustellen, wenn der Roboter das Werkstück entlang eines feststehenden Werkzeugs führen soll.

Nach der Bestimmung der Bahnstützpunkte muss der CAR-Anwender diese auf Erreichbarkeit durch den Roboter und auf Kollisionsfreiheit des Roboters mit seiner Umgebung prüfen. Sind alle Bahnstützpunkte kollisionsfrei erreichbar, kann der CAR-Anwender die Roboter programmieren. Dies geschieht überwiegend in der spezifischen Sprache des Simulationssystems [87]. Die Programme enthalten neben den Bewegungsbefehlen auch Befehle zur Ansteuerung anderer Fertigungskomponenten (z. B. Greifer, Werkzeuge oder Sensoren). Nachdem der CAR-Anwender die Programme erstellt hat, muss er diese in einem Simulationslauf testen, um Ausführungsfehler zu entdecken und Informationen über die Qualität der Ausführung und damit der Planung zu erhalten. In der Praxis wird die Qualität eines Programms und einer Planungsvariante in der Regel durch die Ausführungszeit bestimmt. Der CAR-Anwender muss den Simulationslauf am Bildschirm beobachten und feststellen, wann und wo Optimierungsbedarf besteht. Anschließend muss er entscheiden, wie er durch Änderungen an der Roboter-Fertigungszelle und an den Programmen die Ausführung optimieren kann. Ein Simulationslauf kann je nach Applikation zeitintensiv sein und erfordert vom CAR-Anwender eine hohe Konzentrationsfähigkeit. Der Grad der Verbesserung an der zu planenden Roboter-Fertigungszelle, die aufgrund des Simulationslaufs vorgenommen wird, hängt stark von den individuellen Beobachtungsfähigkeiten und Erfahrungen des CAR-Anwenders ab.

Wird aufgrund des Simulationslaufs eine Änderung an der zu planenden Roboter-Fertigungszelle erforderlich, so muss der CAR-Anwender entscheiden, wie und wo er Änderungen am Roboterprogramm vornehmen, oder wie und welche Fertigungskomponente er verschieben oder sogar austauschen muss.

Es lassen sich folgende Typen von Fertigungskomponenten unterschieden:

1. *Bahnabhängige Komponenten*: Dies sind Komponenten, deren Orte die Roboterbahn direkt beeinflussen (z. B. Poliermaschine in Bild 1.1, S. 2). Durch eine Verschiebung oder einen Austausch einer bahnabhängigen Komponente wird eine erneute Prüfung der Erreichbarkeit und der Kollisionsfreiheit der Bahnstützpunkte notwendig. Gegebenenfalls muss der CAR-Anwender die Roboterprogramme anpassen und einen erneuten Simulationslauf durchführen. Der Roboter ist ein Spezialfall einer bahnabhängigen Komponente, weil dessen Austausch oder dessen Verschiebung in der Regel große Umplanungen und eine Neuprogrammierung zur Folge hat.

2. *Bahnunabhängige Komponenten*: Diese Komponenten beeinflussen die Roboterbahn nicht unmittelbar. Solche Komponenten sind z. B. Hindernisse, die durch ihre Verschiebung oder Beseitigung zu einer fehlerfreien Ausführung führen können. Eine Anpassung der Roboterprogramme ist nicht erforderlich, aber der CAR-Anwender muss einen erneuten Simulationslauf durchführen, um die fehlerfreie Ausführbarkeit zu gewährleisten.

Die Anzahl der Iterationen bis zu einem akzeptablen Ergebnis kann man nicht vorhersagen. Eine in der Praxis durchgeführte, zeitintensive und manuelle Bestimmung der einzelnen Bahnstützpunkte führt in der Regel dazu, dass eine anfänglich gewählte Planungsvariante (z. B. Zellenlayout) festgelegt bleibt, da der Aufwand zur Optimierung zu hoch ist. Dieser Aufwand wird durch eine erneute Anpassung an den Bahnstützpunkten und Roboterprogrammen und durch erneute, zeitintensive Simulationsläufe verursacht [11]. Somit wird vorhandenes Optimierungspotenzial oftmals nicht genutzt.

Automatisierte Unterstützung für Planungsprobleme

Kommerzielle Systeme bieten wenige Verfahren für eine automatisierte Lösungssuche für Planungsaufgaben. Es existieren Ansätze für eine automatisierte Unterstützung für die Problembereiche [11]:

- Roboterstandortoptimierung
- Beseitigung von Zwischenpunkten
- Werkzeugauswahl

Roboterstandortoptimierung

Für die Unterstützung bei der Layouterstellung von Roboter-Fertigungszellen bieten kommerzielle CAR-Systeme (z. B. eM-Workplace™ [158], IGRIP® [36], CimStation™ [150], Workspace5™ [44]) die so genannte Autoplace-Funktion [13]. Diese Funktion ermöglicht dem CAR-Anwender die Prüfung einer festen Zahl von Roboterstandorten und benötigt als Eingangsgröße die vom Roboter zu durchfahrende Bahn, die der CAR-Anwender durch eine Folge von Bahnstützpunkten festlegt. Des Weiteren muss der CAR-Anwender ein Raster vorgeben, das die zu prüfenden Roboterstandorte festlegt. An jedem Roboterstandort prüft die Autoplace-Funktion, ob die Bahnstützpunkte kollisionsfrei erreichbar sind. Anschließend erhält der CAR-Anwender für jeden geprüften Roboterstandort die Angabe, ob der Roboter die Bahn durchfahren kann. Die Autoplace-Funktion hat folgende Nachteile:

1. Die Autoplace-Funktion prüft die kollisionsfreie Erreichbarkeit nur an einzelnen Bahnstützpunkten, sodass die Möglichkeit besteht, dass der Roboter die Bahn zwischen zwei Bahnstützpunkten nicht (kollisionsfrei) durchfahren kann. Für eine zuverlässige Aussage diesbezüglich ist ein Simulationslauf notwendig.
2. Die Ausführungszeit für einen Standort erfordert einen zugehörigen Simulationslauf [13].
3. Die Prüfung auf Ausführbarkeit erfolgt nicht für die eingesetzte Robotersteuerung, d. h. es wird kein steuerungsspezifisches Roboterprogramm ausgeführt.
4. Die Autoplace-Funktion berücksichtigt keine Vorgänge, bei denen der Roboter ein mitgeführtes Werkstück oder Werkzeug während der angegebenen Bahn wechselt.

Insgesamt kann man festhalten, dass der CAR-Anwender durch die Autoplace-Funktion darin unterstützt wird, verschiedene Roboterstandorte zu prüfen. Allerdings berücksichtigt die Autoplace-Funktion keine applikationsspezifischen Anforderungen (z. B. Werkstückwechsel, Produktvarianten, Peripherieansteuerung). Ein Teil der von der Autoplace-Funktion geprüften Roboterstandorte muss der CAR-Anwender mithilfe zeitintensiver Simulationsläufe erneut prüfen und beobachten, um Aussagen über eine fehlerfreie Ausführbarkeit und die Ausführungszeit an einem Roboterstandort zu erhalten.

Beseitigung von Zwischenpunkten

In Programmiermodulen für das Punktschweißen, z. B. bei eM-Spot™ von Tecnomatix [158] oder bei UltraSpot™ von DELMIA [36] (S. 9), trifft man auf eine Funktion, die nicht benötigte Zwischenpunkte einer gegebenen Roboterbahn beseitigt. Der CAR-Anwender gibt die Roboterbahn als Folge von Stützpunkten vor und kennzeichnet, an welchen Stützpunkten der Roboter einen Schweißpunkt setzen muss. Des Weiteren kennzeichnet er, welche Stützpunkte der Roboter zum Anfahren und Abfahren der Schweißpunkte benötigt. Das Programmiermodul beseitigt Bahnstützpunkte aus der angegebenen Stützpunktfolge, sodass die resultierende Roboterbahn alle Schweißpunkte enthält, ohne dass dabei Kollisionen auftreten, wenn der Roboter die Bahn durchfährt. Die für die Ermittlung der resultierenden Bahn durchgeführten Prüfungen sind vergleichbar mit den Prüfungen, die das System während der Ausführung der Autoplace-Funktion macht. Damit gelten dieselben Nachteile.

Eine Unterstützung für den CAR-Anwender zur Optimierung der Reihenfolge, sodass der Roboter die Schweißpunkte in möglichst kurzer Zeit setzt, bietet kein kommerzielles Programmiermodul [13].

Werkzeugauswahl

Eine automatisierte Werkzeugauswahl unterstützen nur wenige CAR-Systeme, z. B. unterstützt das Modul eM-Spot™ der Firma Tecnomatix [158] die Auswahl einer Punktschweißzange. Das Modul eM-Spot™ durchsucht eine Datenbank mit Standardschweißzangen und mit vom Kunden konstruierten Schweißzangen nach folgenden Kriterien:

- Werkstückgeometrie
- Blechdicke
- Schweißpunktinformationen
- Weitere Schweißparameter

Das CAR-System flanscht automatisch das ausgewählte Werkzeug an den Roboter. Der CAR-Anwender muss anschließend Erreichbarkeits- und Kollisionsprüfungen, sowie und Simulationsläufe und manuelle Änderungen wiederholen, bis er ein akzeptables Planungsergebnis erzielt hat.

2.2.2 Verfahren aus dem Forschungsbereich

Verfahren zur Optimierung des Robotereinsatzes aus dem Forschungsbereich wurden für folgende Problemklassen [13] entwickelt. Diese werden anschließend erläutert und bewertet.

- Bahnoptimierung
- Layoutoptimierung

- Werkzeugauswahl
- Werkzeuggeometrie
- Reihenfolgeoptimierung
- Konfigurationsoptimierung

Bahnoptimierung

Der Bahnoptimierung wurde in den letzten Jahren viel Aufmerksamkeit beigemessen [14]. Das Problem ist, eine Bahn zu finden, sodass der Roboter von einer Start- zu einer Zielstellung in möglichst kurzer Zeit gelangt. Mit der Problemlösung beschäftigten sich u. a. HEIM [67], ZHA [175] und ZHANG [176]. Problematisch ist die praktische Umsetzung der optimalen Bahn auf heutigen industriellen Robotersteuerungen, weil diese diesbezüglich weiterentwickelt werden müssten [67]. Aus diesem Grund wird dieses Problem nicht weiter betrachtet.

Layoutoptimierung

Für die Layoutoptimierung im Zusammenhang mit Montageaufgaben, die ein Roboter vom Typ SCARA ausführt, hat PARK [126] ein Verfahren entwickelt, um die Ausführungszeit zu minimieren. Auf Basis vorher festgelegter möglicher Zuordnungen von Standorten zu Fertigungskomponenten und einer fest definierten Montagereihenfolge werden für alle Kombinationen die Verfahrwege und darauf aufbauend die Verfahrzeiten bestimmt. Die Anzahl der möglichen Kombinationen kann über einfache Regeln eingeschränkt werden. PARK schätzt die benötigten Ausführungszeiten mithilfe eines einfachen Modells einer Robotersteuerung. Demzufolge führt er keine umfassenden Simulationsläufe durch. Somit ist eine fehlerfreie Ausführung nicht gewährleistet, z. B. bezüglich Kollisionen, und die für die Optimierung verwendeten Ausführungszeiten sind Schätzungen.

SCHWINN [145] stellt einen Ansatz zur Bestimmung eines optimalen Roboterstandortes vor. Für die Berechnung der Zielfunktion, die einen Roboterstandort bewertet, gibt der Anwender eine fest vorgegebene Folge von Roboterstellungen an. Die Bewegungsdauer des Roboters schätzt SCHWINN mithilfe eines ebenfalls einfachen Steuerungsmodells. Darin lässt SCHWINN nur PTP-Bewegungen zu. Des Weiteren berücksichtigt das Verfahren keine Kollisionen. Das Steuerungsmodell liefert analytisch berechnete Kenngrößen wie Summe der Achsvariablendifferenzen, Maximum der Achsvariablendifferenzen und Ausführungszeit. Für die Minimierung der Zielfunktion setzt SCHWINN numerische Parameteroptimierungsverfahren ein.

WOENCKHAUS [170, 171] hat einen wesentlich allgemeineren Ansatz entwickelt (Bild 2.5), da er nicht nur den Roboterstandort sondern das gesamte Zellenlayout optimiert. Dabei repräsentieren die Eingangsvariablen der Zielfunktion jeweils einen Freiheitsgrad der veränderbaren Komponentenstandorte. Ein Zielfunktionswert wird mithilfe eines Simulationslaufs bestimmt und kann je nach Applikation entweder die Ausführungszeit oder die Summe der auftretenden Winkeldifferenzen des

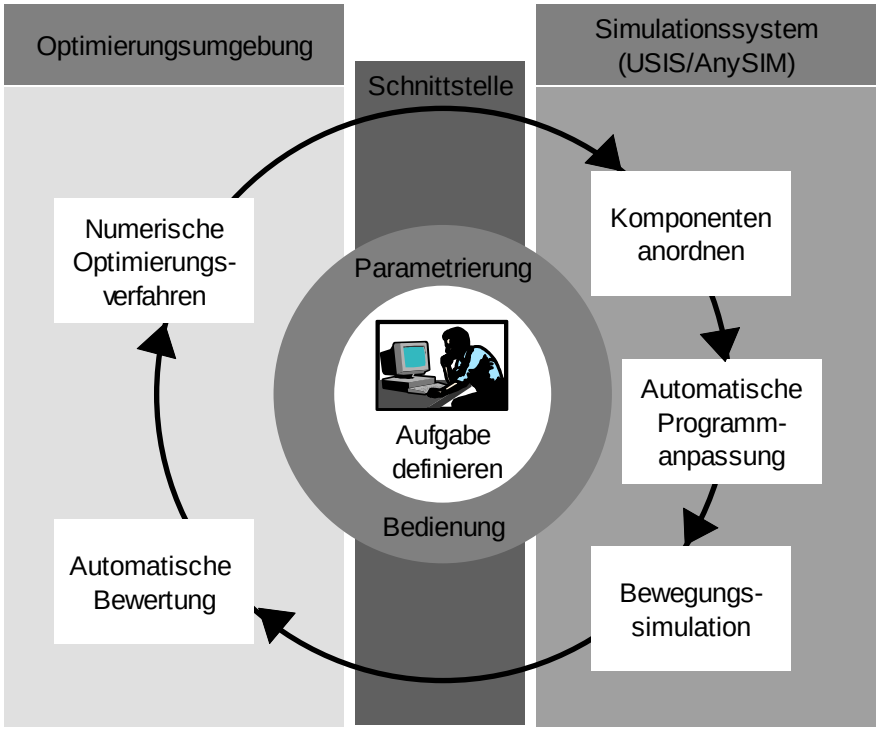


Bild 2.5: Aufbau des Systems für die Layoutoptimierung nach WOENCKHAUS [170]

Roboters sein. WOENCKHAUS berücksichtigt Kollisionen mithilfe von Straffunktionen, die er zur Zielfunktion addiert. Das Roboterprogramm, das zur Bestimmung des Zielfunktionswerts notwendig ist, wird nicht in der steuerungsspezifischen Sprache ausgeführt, sodass WOENCKHAUS keine steuerungsspezifischen Merkmale während des Simulationslaufs berücksichtigen kann. Der Einsatz einer realen Robotersteuerung zur Bestimmung möglichst genauer Ausführungszeiten ist somit ebenfalls nicht möglich. Die im Roboterprogramm auftretenden Befehle muss der CAR-Anwender vor einer Layoutoptimierung fest vorgegeben. Das System kann keine Befehle während des Optimierungslaufs ergänzen (z. B. Befehl für eine Ausweichbewegung). Einen Austausch von Fertigungskomponenten während des Optimierungslaufs sieht WOENCKHAUS ebenfalls nicht vor.

Einen vergleichbaren Ansatz verfolgen BARRAL ET. AL. [13, 14] für die Bestimmung eines optimierten Roboterstandorts, indem sie Simulationsläufe zur Bestimmung der Zielfunktionswerte einsetzen. Die Optimierungsverfahren beruhen auf zufallsbasierten Methoden des Simulated-Annealing.

Einen anderen Ansatz zur Bestimmung eines Layouts verfolgt LÜTH [101]. Er plant ausgehend von einer Menge an Fertigungskomponenten und einer Menge von möglichen Bahnverläufen die Komponenten sukzessive in die Roboterzelle ein. Dazu sind beim Einfügen einer neuen Komponente möglicherweise größere Umplanungen erforderlich. Für die kollisionsfreie Bewegungsplanung setzt LÜTH den kartesischen Konfigurationsraum von HÖRMANN [74] ein. Die Bewertung erfolgt ausschließlich anhand der Bewegungsbahnen, z. B. Summe oder Maximum der Achsvariablendifferenzen. Vorgänge entlang der Bahn, z. B. Greifen von Werkstücken, werden nicht betrachtet. Es wird kein Roboterprogramm erzeugt und damit auch kein Simulationslauf zur Prüfung und Ausführungszeitermittlung durchgeführt. Die Bestimmung eines optimalen Layouts bezüglich minimaler Taktzeit ist deshalb nicht möglich. Des Weiteren bleiben steuerungsspezifische Eigenschaften in seinem Verfahren eben-

falls unberücksichtigt. LÜTH führt an, dass die Planung einer Roboter-Fertigungszelle stark abhängig vom Roboter, den einzusetzenden Fertigungskomponenten und der vom Roboter auszuführenden Aufgabe ist. Aus diesem Grund kann der CAR-Anwender den Planungsprozess selbst programmieren. Den Einsatz von applikationsspezifischen Heuristiken zur Lösung derartiger komplexer Planungs- bzw. Optimierungsaufgaben betrachtet LÜTH als unvermeidlich.

Werkzeugauswahl

In den Fertigungsstraßen im Rohbau der Automobilhersteller setzen Roboter an den Karossen mehrere hundert Schweißpunkte. Bei der Neuplanung von solchen Fertigungsstraßen bietet sich die Verwendung vorhandener Schweißzangen an. Dazu muss aus einer Menge von existierenden Schweißzangen und einer Menge von Schweißpunkten eine Untermenge von Schweißzangen bestimmt werden, sodass die Kosten für die Zangen insgesamt minimal sind. PASHKEVIC [127] hat z. B. eine eigene Heuristik für die Lösungssuche dieses kombinatorische Optimierungsproblem entwickelt. Diese Heuristik setzt voraus, dass es für jeden Schweißpunkt eine Schweißzange gibt, die den Schweißpunkt setzen kann. Diese Voraussetzung ist für die Praxis unzutreffend, da gerade für die Zuordnung zwischen Schweißpunkt und Schweißzange viele roboterspezifische Bedingungen zu beachten sind (z. B. muss der Roboter den Schweißpunkt erreichen). Die Einhaltung solcher Bedingungen kann man nur durch einen Simulationslauf auf Basis von steuerungsspezifischen Programmen gewährleisten.

Werkzeuggeometrie

Die Optimierung der Werkzeuggeometrie hat zum Ziel, offene geometrische Größen (z. B. Länge) derart festzulegen, dass die Abläufe in der Roboter-Fertigungszelle bezüglich einer Zielfunktion optimal sind (Bild 2.6).

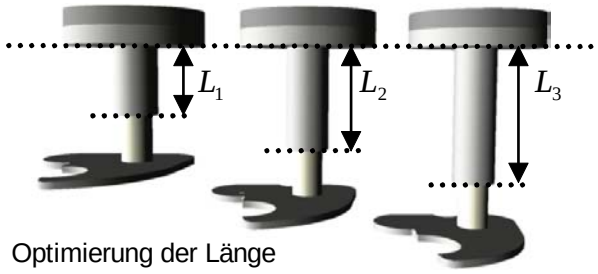


Bild 2.6: Beispiel zur Optimierung der Werkzeuggeometrie

Eine Veränderung der Werkzeuggeometrie hat unmittelbar eine Veränderung der Bewegungsbahn des Roboters zur Folge. SCHWINN [145] bewertet z. B. die Werkzeuggeometrie für eine spezifische Aufgabe über verschiedene analytisch berechnete Kenngrößen (z. B. Maximum der Achsvariablendifferenzen) analog zu seiner Roboterstandortoptimierung (S. 23). Es gelten die dort genannten Nachteile.

Reihenfolgeoptimierung

Verfahren zur Reihenfolgeoptimierung haben zum Ziel, die Reihenfolge für eine Bearbeitungsaufgabe (z. B. Punktschweißen, Bahnschweißen, Montage) derart festzulegen, dass die Abläufe in der Roboter-Fertigungszelle gemäß einem bestimmten Kriterium (z. B. Ausführungszeit) optimal sind.

BARRAL ET. AL. stellen ein Verfahren zur Optimierung der Reihenfolge zum Setzen der Schweißpunkte durch den Roboter vor [13]. Das Verfahren minimiert die Ausführungszeit, die der Roboter für den Schweißvorgang benötigt. Dabei bilden BARRAL ET. AL. in ihrem Verfahren das Reihenfolgeproblem auf ein Rundreiseproblem (Traveling-Salesman-Problem) ab. Die jeweiligen Verfahrzeiten zwischen zwei beliebigen Schweißpunkten ermittelt das Verfahren durch Simulationsläufe. Für die Lösung des Rundreiseproblems setzen BARRAL ET. AL. in ihrem Verfahren Methoden des Simulated-Annealing ein. Das Verfahren ist starr, sodass die Berücksichtigung von in der Praxis üblichen An- und Abfahrpunkten oder der Einsatz einer anderen Bewegungsart (zirkular, PTP) zwischen zwei Punkten nicht durchführbar ist. Dies gilt auch für das Verfahren von DUBROWSKY ET. AL. [39], der die Ausführungszeit zwischen zwei Punkten über die euklidische Distanz schätzt.

Vergleichbare Verfahren stellen RUBINOVITZ ET. AL. [139] und HARTFUSS [66] zur Bestimmung der Bearbeitungsreihenfolge von Schweißnähten beim Bahnschweißen vor. Beide Verfahren bestimmen neben der Reihenfolge der Schweißnähte gleichzeitig auch die Bearbeitungsrichtung einer Schweißnaht. Dazu bauen sie einen Graphen auf, bei dem die Knoten den Anfangs- und Endpunkten der Schweißnähte entsprechen. Eine Kante entspricht der Roboterbewegung von einem Anfangs- oder Endpunkt einer Schweißnaht zu einem anderen Anfangs- oder Endpunkt. Für die Kantenbewertung setzen sie in Abhängigkeit von der Kante die Bearbeitungszeit einer Schweißnaht von deren Anfangspunkt zu deren Endpunkt oder die Zeit für eine Transferbewegung vom Endpunkt einer Schweißnaht zum Anfangspunkt einer anderen Schweißnaht ein. Die Verfahrzeiten schätzen RUBINOVITZ ET. AL. und HARTFUSS analytisch ab. Für die Lösung des Rundreiseproblems setzen beide jeweils ein heuristisches Suchverfahren ein. Einen ähnlichen Ansatz verwendet auch DAMSBO [35], der zur Bewertung einer Bahn deren Länge einsetzt, und zur Lösung des Reihenfolgeproblems auf evolutionäre Algorithmen zurückgreift.

Ansätze zur Optimierung der Montagereihenfolge von Bauteilen finden sich u. a bei LEVI [97], FROMMHERZ [56] und PARK [126]. Diese Ansätze reduzieren die Anzahl aller Montagefolgen entweder durch einen UND/ODER-Graphen (LEVI), durch Regeln (PARK) oder durch einen Vorranggraphen (FROMMHERZ). Um die bestmögliche Montagefolge zu finden, setzen die Ansätze heuristische Suchverfahren auf der Menge der reduzierten Montagefolgen ein.

SHENG ET. AL. [146] bestimmen eine Reihenfolge mit minimaler Ausführungszeit von Messpunkten zur Prüfung von Oberflächen im Automobilbau ebenfalls durch Lösung des Rundreiseproblems. Zur Reduktion des Rechenaufwands bilden sie Teilmengen der anzufahrenden Messpunkte und lösen das Rundreiseproblem für jede Teilmenge. Anschließend wird die optimale Reihenfolge zum Anfahren der Teilmengen bestimmt.

Konfigurationsoptimierung

Wegen ihrer Kinematik können heutige Industrieroboter einen Punkt samt Orientierung im Raum, ein so genanntes Frame, mit mehreren Achsstellungen erreichen. Für die Identifikation einer eindeutigen Achsstellung gibt man neben dem Frame die so genannte Konfiguration an [57]. Unterschiedliche Konfigurationen bei einem Frame verursachen unterschiedliche Bewegungsbahnen zum Erreichen des Frames. Die Aufgabe der Konfigurationsoptimierung ist die Bestimmung einer Konfiguration für jedes Frame auf einer Roboterbahn, sodass die Gesamtbewegung optimal wird.

SCHWINN [145] stellt dazu einen Lösungsansatz vor, der auf ähnlichen Prinzipien beruht, die er bei der Optimierung des Roboterstandorts (S. 23) oder der Werkzeuggeometrie (S. 25) verwendet. SCHWINN lässt nur PTP-Bewegungen zu und bewertet die Gesamtbewegung durch verschiedene Kenngrößen (z. B. Summe oder Maximum der Achsvariablendifferenzen, Ausführungszeit). Die Bestimmung der Kenngröße Ausführungszeit erfolgt über eine analytische Schätzung und nicht über einen Simulationslauf. Für die Minimierung der Kenngrößen setzt SCHWINN auch hier numerische Parameteroptimierungsverfahren ein.

Insgesamt kann man festhalten, dass das von SCHWINN entwickelte System ROBOPT das einzige System ist, bei dem die Kombination verschiedener Optimierungsprobleme (Roboterstandort, Werkzeuggeometrie, Konfigurationsoptimierung) möglich ist. Der Anwender spezifiziert dabei über die Sprache AUTOOPT das applikationsspezifische Optimierungsproblem. Der Vorteil dieses Ansatzes ist die Aneinanderreihung mehrere Optimierungsprobleme und die Wiederholbarkeit für ähnliche Probleme. Allerdings können nur vordefinierte Lösungsansätze aneinander gereiht werden. Das benötigte Bewegungsprogramm ist während der Optimierung nicht veränderbar und lässt ausschließlich PTP-Bewegungen zu. In der Praxis benötigte Befehle wie Linearbewegungen oder Befehle für eine E/A-Kommunikation sind nicht möglich. Auch muss man nach der Beendigung des Optimierungslaufs das optimale Programm manuell in die Steuerungssyntax konvertieren.

2.3 Fazit und Handlungsbedarf

Eine Aufwandsreduktion für die Roboterprogrammierung wird seit längerer Zeit mithilfe entsprechender Systeme und Lösungsansätze angestrebt. Die Schwierigkeiten einer automatischen Programmgenerierung liegen u. a. an den starken Abhängigkeiten von den verwendeten CAD-Daten, der eingesetzten Robotersteuerung, der Fertigungstechnik und weiteren applikationsspezifischen Anforderungen. Zahlreiche Programmiersysteme unterstützen den CAR-Anwender bei der Erzeugung von Roboterprogrammen durch interaktive Mechanismen. Im Hinblick auf die Programmierung zahlreicher Produktvarianten müssen die CAR-Anwender zur Programmierung einer neuen Produktvariante jedoch viele Programmierschritte manuell wiederholen, sodass nur erfahrene CAR-Anwender die Programmierung übernehmen können. Unterstützende Automatismen zur Programmgenerierung, die der CAR-Anwender gemäß seinen Anforderungen anpassen kann, sind kaum vorhanden.

Außerdem zeigt sich, dass die erzeugbaren Roboterprogramme in ihrer Struktur starr sind, sodass man diese nach ihrer Erzeugung manuell modifizieren muss. Dies erfordert immer die Kenntnis der

steuerungsspezifischen Sprache. Bei in der Praxis üblichen Änderungen an den Anforderungen oder bei neuen Produktvarianten muss man fast alle durchgeführten Programmierschritte und Tests manuell wiederholen. Dies hat einen großen Aufwand zur Folge. Des Weiteren kann man Fehler in den Algorithmen der Postprozessoren erst auf der Robotersteuerung finden. Solche Fehler können große Schäden in der Roboter-Fertigungszelle verursachen.

Bei der Optimierung des Robotereinsatzes sind in kommerziellen CAR-Systemen nur wenige, in der Forschung dafür viele Ansätze vorhanden. Die wissenschaftlichen Ansätze abstrahieren stark von einer konkreten Applikation, sodass sie kaum applikationsspezifische Anforderungen bei der Optimierung berücksichtigen können. Dies ist aber in der Praxis die Voraussetzung für die Akzeptanz und die Umsetzbarkeit einer Lösung. Die wissenschaftlichen Ergebnisse in LÜTH [101] und SCHWINN [145]) zeigen, dass eine Programmierbarkeit der Optimierungsstrategie erforderlich ist, um praktische Ergebnisse für eine konkrete Aufgabe zu erhalten und den Berechnungsaufwand für die Suche nach der bestmöglichen Lösungsvariante klein zu halten. Die betrachteten Ansätze schätzen die Ausführungszeit für eine Roboterbewegung mithilfe eines einfachen Steuerungsmodells ab. Um möglichst präzise Ausführungszeiten zu erhalten, muss man aber steuerungsspezifische Roboterprogramme ausführen. Des Weiteren kann man in der Planungsphase nur durch die Ausführung von steuerungsspezifischen Roboterprogrammen die spätere Ausführbarkeit in der Roboter-Fertigungszelle gewährleisten.

Insgesamt kann man festhalten, dass es kein System gibt, mit dem der CAR-Anwender steuerungsspezifische Roboterprogramme für verschiedene Hersteller automatisiert erzeugen, ausführen und bewerten kann. Der CAR-Anwender kann in den Programmiersystemen, die eine Programmgenerierung enthalten, die zu verwendende Vorschrift für die Berechnung der Programme oder einzelner Programmteile nicht nach seinen applikationsspezifischen Anforderungen ändern. Eine umfassende Ausführung steuerungsspezifischer Programme in einem Simulationslauf für unterschiedliche Hersteller beherrschen nur wenige CAR-Systeme. Kein System kann Informationen aus dem Simulationslauf (z. B. Ausführungszeit) verwenden, um das Roboterprogramm oder das Planungsergebnis einer Roboter-Fertigungszelle zu verbessern. Erst durch die Möglichkeit, steuerungsspezifische Roboterprogramme automatisiert zu erstellen, auszuführen und zu bewerten, kann man Optimierungsverfahren einsetzen, ohne dass weitere zeitintensive Konvertierungen vor der Inbetriebnahme- oder Umbauphase erforderlich sind. Ein solcher Ansatz ist die Voraussetzung dafür, dass Optimierungsergebnisse in der Praxis zuverlässig umsetzbar sind.

3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an ein System zur Automatisierung der Programmierung und Planung aus den Erkenntnissen des vorangegangenen Kapitels abgeleitet. Diese Anforderungen dienen als Grundlage für das Systemkonzept. Sie beschreiben in ihrer Summe die Neuartigkeit des in dieser Arbeit entwickelten Systems.

3.1 Programmierbarkeit

Das System soll es einem CAR-Anwender ermöglichen, steuerungsspezifische Roboterprogrammvarianten zu generieren, auszuführen und zu bewerten. Es soll gemäß einer vom CAR-Anwender vorgegebenen Algorithmus aus unterschiedlichen Eingangsinformationen (z. B. Geometriedaten, fertigungstechnische Parameter) die Roboterprogrammvarianten generieren. Im Folgenden wird dieser Algorithmus *Erzeugungsalgorithmus* genannt.

Das System soll für alle Varianten eines begrenzten Produktspektrums denselben Erzeugungsalgorithmus verwenden (Bild 3.1). Für das einleitende Beispiel (Bild 1.1, S. 2) bedeutet dies, dass der CAR-Anwender den applikationsspezifischen Erzeugungsalgorithmus festlegt, wie das System aus den Geometriedaten der Metallplatten und einstellbaren Polierparametern die Roboterprogramme in der Sprache MELFA BASIC IV generieren soll. Mit diesem Erzeugungsalgorithmus kann das System alle Polierprogramme für die in Bild 1.1 dargestellten Produktvarianten generieren. Für die Erzeugung eines ausführbaren Roboterprogramms zum Polieren einer neuen Produktvariante ist jetzt kein hoch qualifizierter CAR-Anwender mehr nötig, weil das System die Schritte zur Erzeugung kennt. Es müssen lediglich die Geometriedaten in das System geladen, gegebenenfalls die Polierparameter verändert und das System gestartet werden.

Für eine Programmgenerierung müssen viele Faktoren berücksichtigt werden (z. B. die einzusetzende Robotersteuerung, Struktur und Format der Geometriedaten, einzusetzende Fertigungstechnik, Kommunikationsprotokoll für die Ansteuerung externer Geräte, usw.). Für einen erfolgreichen Einsatz muss das System möglichst alle Faktoren berücksichtigen, die sich in der Praxis auch ändern können. Deshalb ist ein hoher Grad an Flexibilität und Anpassungsfähigkeit nötig, der nur durch die Programmierbarkeit des Systems (Bild 3.1) erreicht werden kann [101, 145].

Anforderung 1: *Ein System, mit dem ein CAR-Anwender steuerungsspezifische Roboterprogrammvarianten generieren, ausführen und bewerten kann, muss mithilfe einer einfachen Sprache (Folge von Anweisungen) programmierbar sein. Die Programmierbarkeit muss es dem CAR-Anwender erlauben, einen applikationsspezifischen Erzeugungsalgorithmus zu realisieren. Dieser legt fest, wie das System die Roboterprogramme aus den Eingangsinformationen erzeugen soll.*

Das System muss dem CAR-Anwender Funktionen zur Verfügung stellen, die er bei der Realisierung des Erzeugungsalgorithmus häufig benötigt. Zu diesen Funktionen gehört u. a. die Möglichkeit, Geometriedaten zur Berechnung von Bahnstützpunkten einzusetzen. Der CAR-Anwender muss z. B. im

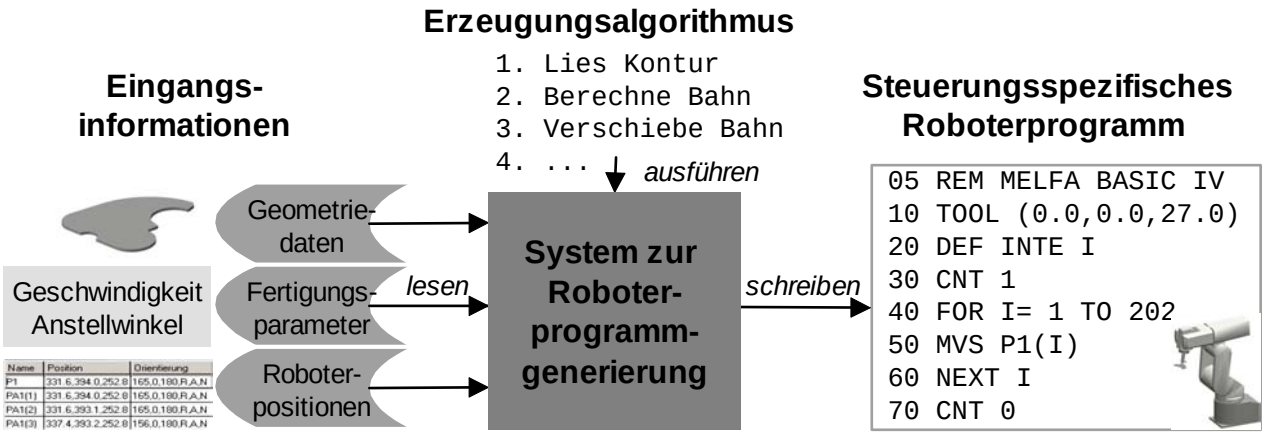


Bild 3.1: Programmierbarkeit des Systems (Anforderung 1)

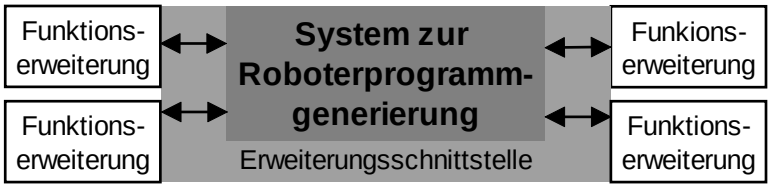


Bild 3.2: Erweiterbarkeit des Systems (Anforderung 3)

Erzeugungsalgorithmus für das einleitende Beispiel (Bild 1.1, S. 2) die Möglichkeit erhalten, Funktionen zu verwenden, die eine Berechnung der Stützpunkte für die Roboterbahn entlang des Metallplattenrands ermöglicht.

Weitere Funktionen benötigt der CAR-Anwender für die Planung und Optimierung von Roboter-Fertigungszellen, z. B. zur Minimierung der Ausführungszeit. Letzteres wird auch im einleitenden Beispiel benötigt, um einen geeigneten Standort für die Poliermaschine zu finden.

Anforderung 2: *Das System muss einen Umfang an Basisfunktionen zur Verfügung stellen, um den CAR-Anwender von der Programmierung von häufig benötigten Funktionen zu befreien, die er für die Generierung von steuerungsspezifischen Roboterprogrammen benötigt.*

Aufgrund der vielen Faktoren, von denen eine konkrete Programmgenerierung abhängt, kann kein Programmiersystem den vollständigen Umfang an Basisfunktionen von Anfang an zur Verfügung stellen [78]. Außerdem gibt es bereits zahlreiche Ansätze zur Lösung von Teilproblemen (z. B. die kollisionsfreie Bahnplanung von Transferbewegungen bei Montagevorgängen). Um vorhandene und neue Verfahren für den CAR-Anwender verfügbar zu machen, muss das System über eine einfache Erweiterungsschnittstelle verfügen (Bild 3.2).

Anforderung 3: D*ie Systemarchitektur muss offen sein, um den Umfang an Basisfunktionen um zusätzliche Funktionen erweitern zu können. Die dazu notwendige Systemschnittstelle muss einfach erlernbar und bedienbar sein.*

Durch die Erfüllung der Anforderungen 1-3 kann der CAR-Anwender abhängig von einer Aufgabe verschiedene Lösungsmechanismen in seinem Erzeugungsalgorithmus kombinieren, um eine für ihn bestmögliche Lösung zu erstellen. Die Lösungsmechanismen stellt das System durch den Umfang an

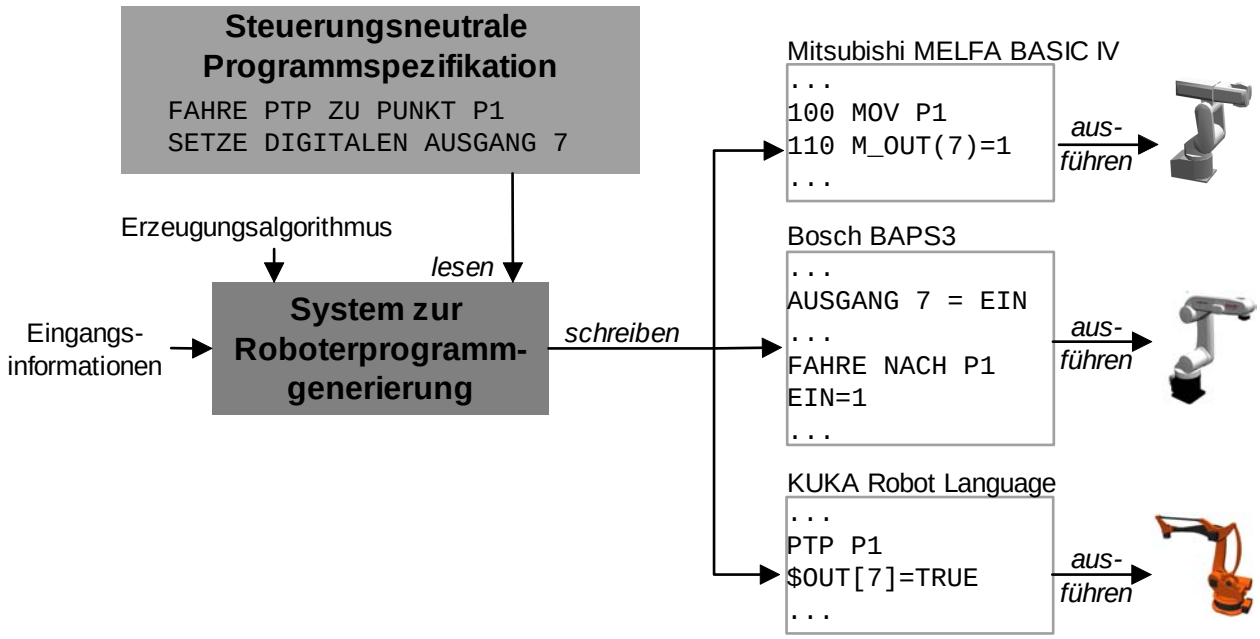


Bild 3.3: Programmspezifikation: Anforderung 4

Basisfunktionen zur Verfügung (Anforderung 2). Diesen Funktionsumfang kann man gegebenenfalls erweitern (Anforderung 3). Der CAR-Anwender kann diese Basisfunktionen aufgrund der Programmierbarkeit des Systems kombinieren (Anforderung 1), um damit einen Erzeugungsalgorithmus zu implementieren, aufgrund dessen das System die Roboterprogrammvarianten erzeugen kann.

3.2 Programmspezifikation

Weil jeder Roboterhersteller eine eigene Sprache zur Programmierung seiner Robotersteuerungen entwickelt hat und weil diese Sprache zwischen den einzelnen Steuerungsversionen variieren kann, muss der CAR-Anwender definieren können, in welcher Steuerungssyntax das System das Roboterprogramm ausgeben soll.

Anforderung 4: *Das System muss für die jeweilige Robotersteuerung angepasste steuerungsspezifische Programme erzeugen.*

Die Angabe des Programminhalts (z. B. enthaltene Roboterbefehle) darf nicht steuerungsspezifisch sein und wird im Folgenden als *Programmspezifikation* bezeichnet (Bild 3.3). Die Programmspezifikation enthält die Angaben, welche Roboterbefehle in einer Programmvariante auftreten können. Damit der CAR-Anwender nicht für jede Robotersteuerung eine eigene Sprache lernen muss, darf die Programmspezifikation nicht auf eine spezifische Steuerung zugeschnitten sein.

Anforderung 5: *Die Programmspezifikation legt fest, welche Roboterbefehle ein zu generierendes Programm enthalten kann. Die Programmspezifikation muss steuerungsneutral sein.*

Der erste Vorteil, der sich durch diese Anforderung ergibt, besteht darin, dass der CAR-Anwender die im Programm zu verwendenden Roboterbefehle nur einmal angeben muss und das System anschlie-

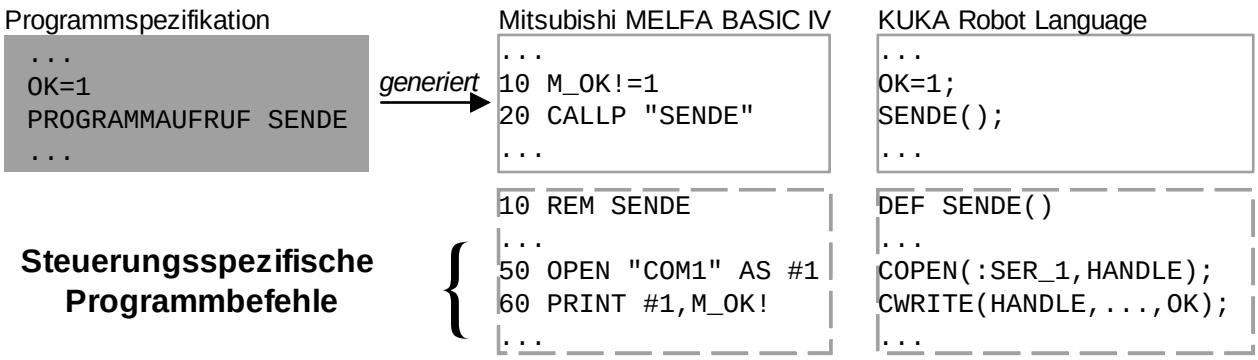


Bild 3.4: Verwendung steuerungsspezifischer Befehle in generierten Programmen (Anforderung 7)

ßend die Programme in unterschiedlichen Sprachen ausgeben kann. Der zweite Vorteil besteht darin, dass der CAR-Anwender die vielen Roboterprogrammiersprachen nicht beherrschen, sondern nur die Kenntnis über die steuerungsneutrale Programmspezifikation besitzen muss.

Anforderung 5 ist vergleichbar mit dem Postprozessoransatz bisheriger herstellerunabhängiger CAR-Systeme und macht sich dadurch die beiden genannten Vorteile nutzbar. Für den CAR-Anwender vereinfacht sich die Roboterauswahl in der Planungsphase, weil er den zu programmierenden Ablauf nur einmal steuerungsneutral formulieren muss. Im Gegensatz zu bisherigen herstellerunabhängigen CAR-Systemen verfolgt das entwickelte System aber den Ansatz, den programmierten Ablauf mit dem jeweiligen steuerungsspezifischen Roboterprogramm zu prüfen und somit Eigenschaften der realen Robotersteuerung in der Planungsphase und nicht erst in der Inbetriebnahme- oder Umrüstphase. Die Roboterprogramme sind bereits bei Beginn dieser Phasen bis auf geringe Stützpunktkorrekturen getestet. Dies hilft, diese Phasen zu verkürzen, weil der CAR-Anwender aufwendige Entwicklungs- und Testarbeiten bereits im Vorfeld erledigen kann.

Jeder Hersteller stellt in seiner Steuerung jeweils eigene spezifische Funktionen zur Verfügung. Das System muss Roboterprogramme generieren können, die sämtliche herstellerspezifischen Steuerungsfunktionen verwenden können. Sobald das System Einschränkungen in den verwendbaren Steuerungsfunktionen besitzt und der CAR-Anwender die ausgeschlossenen Steuerungsfunktionen benötigt, sind die generierten Roboterprogramme in der Praxis wertlos. Dies ist der Grund, dass der CAR-Anwender bei den bisherigen herstellerunabhängigen CAR-Systemen die Programme bei jeder Erzeugung manuell modifizieren muss [128].

Anforderung 6: *Generierte Programme sollen alle herstellerspezifischen Steuerungsfunktionen verwenden können.*

Durch Anforderung 6 ergibt sich das Problem, dass eine steuerungsneutrale Programmspezifikation nur eine Untermenge der gemeinsamen Roboterbefehle heutiger Steuerungen darstellen kann. Die Roboterbefehle, die nicht in der Programmspezifikation definiert und damit nicht in einem generierten Roboterprogramm auftauchen können, müssen deshalb bei gleichzeitigem Verzicht auf seine jeweilige manuelle Modifikation auf eine neue Art von dem generierten Programm verwendbar sein.

Anforderung 7: *Das System muss die Möglichkeit bieten, steuerungsspezifische Programmbefehle aufzurufen und Daten mit einem manuell erstellten Roboterprogramm auszutauschen.*

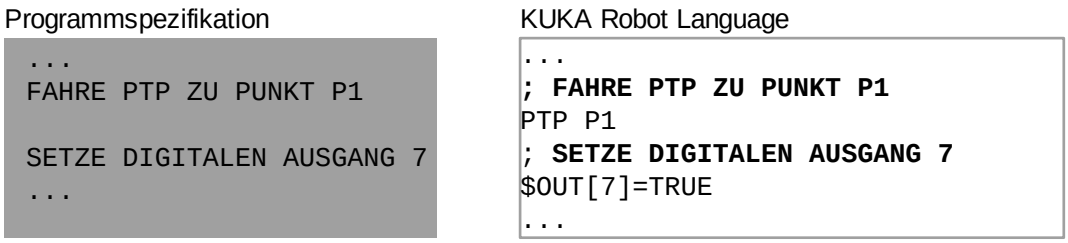


Bild 3.5: Bezüge zur Programmspezifikation in den generierten Programmen (Anforderung 8)

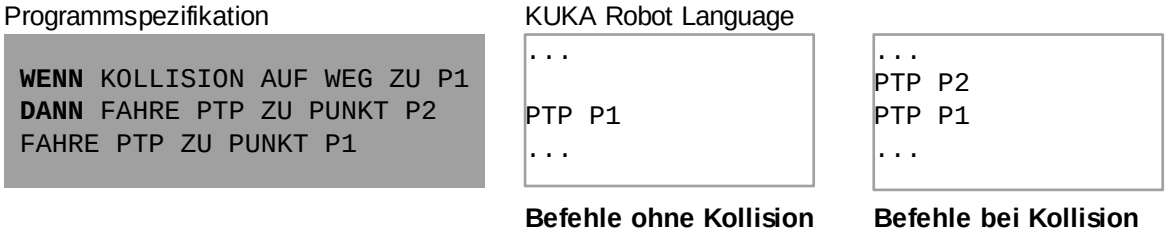


Bild 3.6: Entscheidungen in der Programmspezifikation (Anforderung 9)

Durch diese Anforderung wird es möglich, dass ein generiertes Programm ohne manuelle Modifikation steuerungsspezifische Roboterbefehle aufrufen kann, die nicht in der Programmspezifikation formulierbar sind. Diese Befehle kapselt man in einem Programm, das man nur einmal manuell erstellen muss. Durch Anforderung 7 kann das generierte Programm das manuell erstellte Programm parametrieren und aufrufen (Bild 3.4).

Ein weiterer Grund für Anforderung 7 ist die Möglichkeit zur Verwendung vorhandener Roboterprogrammbibliotheken, wie sie zur Vereinfachung der Programmierung für spezielle Fertigungstechniken oder für die Ansteuerung spezifischer Geräte angeboten werden.

Damit die generierten Programme lesbar und verständlich sind, müssen sie Bezüge auf die Programmspezifikation in Form von Kommentaren enthalten. Somit ist eine Zuordnung möglich, welcher Teil der Programmspezifikation für eine Folge von Roboterbefehlen in dem generierten Programm verantwortlich ist (Bild 3.5).

Anforderung 8: *Die generierten Roboterprogramme müssen Bezüge auf die Programmspezifikation in Form von Kommentaren enthalten.*

Während der Programmgenerierung kann es sich herausstellen, dass man für eine spezielle Programmvariante einen zusätzlichen Roboterbefehl benötigt. Dies ist z. B. erforderlich, wenn ein Roboterprogramm eine Kollision verursacht. Dann muss dieses Roboterprogramm einen zusätzlichen Roboterbefehl enthalten, der eine Ausweichbewegung durchführt. Eine Verallgemeinerung dieses Sachverhalts bedeutet, dass es möglich sein muss, Roboterprogrammvarianten mit jeweils unterschiedlichen Roboterbefehlen zu erzeugen.

Anforderung 9: *Die Programmspezifikation muss Entscheidungen berücksichtigen, die das System erst zur Laufzeit treffen kann. Somit muss das System Roboterprogrammvarianten erzeugen können, die in Abhängigkeit von der Entscheidung jeweils unterschiedliche Roboterbefehle enthalten.*

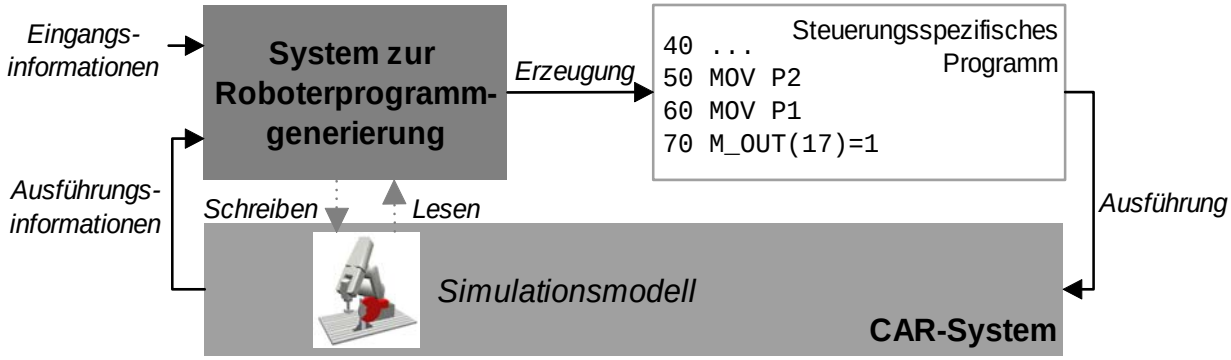


Bild 3.7: Simulation: Anforderungen 10 und 11

Da die Programmspezifikation steuerungsneutral ist, fehlt dem System für die Generierung der Roboterprogramme die Information über die konkrete Steuerungssyntax. Da viele Roboterprogrammiersprachen, inklusive der Unterschiede aufgrund der Steuerungsversionen, existieren, ist es sinnvoll, diese Information extern zu kapseln. Dies ermöglicht Änderungen der Steuerungssyntax, die aufgrund neuer Steuerungsversionen entstehen, ohne dass tief greifende Systemänderungen erforderlich sind. Sofern die Syntax einer (neuen) Roboterprogrammiersprache formal beschreibbar ist, ist der Aufwand zu deren Unterstützung gering.

Anforderung 10: *Die Information über eine Steuerungssyntax soll in einer Datei gekapselt sein.*

Nach den Anforderungen bezüglich der Programmgenerierung in diesem Abschnitt erfolgt im nächsten Abschnitt die Betrachtung der Anforderungen der Programmausführung.

3.3 Simulation

Für eine umfassende Prüfung der generierten Roboterprogramme bietet sich deren Ausführung in einem CAR-System an, um möglichst alle späteren Fehler in der Roboter-Fertigungszelle zu finden. Die Ausführung der Programme muss dabei realitätsnah im Simulationsmodell der Roboter-Fertigungszelle erfolgen (Bild 3.7).

Da es möglich sein soll, dass die generierten Programme manuell erstellte Programme aufrufen (Anforderung 7), muss das System die manuell erstellten Programme ebenfalls simulieren. Durch einen solchen Simulationslauf kann man viele Fehlerquellen finden, z. B. fehlerhafter Sprachgebrauch, den Versuch des Roboters, singuläre Stellungen zu durchfahren, Kollisionen, Fehler in den Kommunikationsprotokollen zwischen der Robotersteuerung und der Peripherie.

Der Simulationslauf dient neben der Entdeckung von Fehlern auch zur Ermittlung von Informationen, die erst nach einer Programmausführung verfügbar sind (z. B. die Ausführungszeit). Diese Informationen dienen als Basis für die anschließende Bewertung der Qualität der Roboterprogramme und damit einer Planungsvariante. Treten während der Programmausführung Fehler auf, so muss man die Programme und damit die zugehörige Planungsvariante als unbrauchbar bewerten.

Anforderung 11: *Ein CAR-System soll die generierten oder manuell erstellten, steuerungsspezifischen Roboterprogramme in einem Simulationslauf ausführen. Dieser Simulationslauf dient zur Prüfung einer Planungsvariante und zur Ermittlung von Laufzeitinformationen, um die Programme und damit die zugehörige Planungsvariante zu bewerten.*

In den Roboterprogrammen werden zum Teil Informationen benötigt, die im Simulationsmodell der Roboter-Fertigungszelle enthalten sind. Dazu gehören u. a. Roboterinformationen (z. B. Anzahl der Achsen des Roboters), Geometrieinformationen (z. B. zur Bearbeitung von Oberflächen) und E/A-Verbindungen zwischen der Robotersteuerung und anderen Fertigungskomponenten (z. B. SPS, Werkzeug). Diese Informationen sollen dem CAR-Anwender bei der Formulierung des Erzeugungsalgorithmus und der Programmspezifikation zur Verfügung stehen. Aus diesem Grund wird ein lesender Zugriff auf das Simulationsmodell gefordert (Bild 3.7).

Da der CAR-Anwender applikationsspezifische Planungsprobleme lösen soll, muss das System vor einer Programmausführung das Simulationsmodell verändern, z. B. die Veränderung des Standortes der Poliermaschine im einleitenden Beispiel (Bild 1.1, S. 2), um Kollisionen korrekt zu erkennen. Die Angabe über die Änderung erfolgt in der vom CAR-Anwender formulierten Erzeugungsalgorithmus. Darin muss er einen schreibenden Zugriff auf das Simulationsmodell erhalten (Bild 3.7).

Anforderung 12: *Der CAR-Anwender muss für den Erzeugungsalgorithmus und für die Programmspezifikation einen lesenden und schreibenden Zugriff auf das Simulationsmodell erhalten. Er muss darin simulierte Fertigungskomponenten löschen, ergänzen, austauschen und deren Eigenschaften abfragen und verändern können.*

Die in diesem Kapitel dargestellten Anforderungen, die die Neuartigkeit des Systems ausmachen, muss das Systemkonzept im nächsten Kapitel vollständig umsetzen. Die Kernanforderungen dieses Kapitels waren die Anforderungen bezüglich des Erzeugungsalgorithmus, der angibt *wie* das System ein Roboterprogramm generieren soll und die Anforderungen bezüglich der Programmspezifikation, die angibt, *was* das Roboterprogramm können muss.

4 Systemkonzept

Unter Berücksichtigung der Anforderungen aus Kapitel 3 wird in diesem Kapitel das Systemkonzept erläutert. Dieses Kapitel führt in die verwendete Terminologie ein und dient als Übersicht für die Kapitel 5-8 dieser Arbeit. Es wird gezeigt, wie das Systemkonzept die Anforderungen umsetzt und damit insgesamt erfüllt.

4.1 Überblick

Das System besteht aus vier Schichten und zwei Rückkopplungen (Bild 4.1). Die Schichten haben folgende Verantwortlichkeiten:

- *Applikationsschicht*: Die Verantwortlichkeit dieser Schicht liegt in der Bereitstellung der Informationen, die für die Erzeugung der Programme nötig sind.
- *Generierungsschicht*: Die Verantwortlichkeit dieser Schicht liegt einerseits in der Koordination des Ablaufs (Modul *Generierungssteuerung*) und andererseits in der Erzeugung der steuerungsspezifischen Programme (Modul *Programmsynthese*).
- *Simulationsschicht*: Die Verantwortlichkeit dieser Schicht liegt in der Ausführung der steuerungsspezifischen Programme in einem Simulationslauf, um Ergebnisse über die Ausführung (z. B. Ausführungsfehler, Ausführungszeit) der Programme zu erhalten.
- *Fertigungsschicht*: Die Verantwortlichkeit dieser Schicht liegt in der realen Ausführung der steuerungsspezifischen Roboterprogramme in der Roboter-Fertigungszelle.

Die beiden Rückkopplungen werden wie folgt unterschieden:

- *Innere Rückkopplung*: Diese Rückkopplung wird durch die Ausführung der Roboterprogramme in der Simulationsschicht und durch die Erzeugung von Simulationsergebnissen für die Generierungsschicht gebildet.
- *Äußere Rückkopplung*: Diese Rückkopplung wird durch die Ausführung der Roboterprogramme in der Fertigungsschicht und die dabei entstandenen Ausführungsergebnisse gebildet. Der Anwender bestimmt durch Beobachtung die Ausführungsergebnisse.

Bei den folgenden Betrachtungen wird das Problem der Generierung von Roboterprogrammen für variantenreiche Produkte in den Vordergrund gestellt. Die Besonderheiten, die beim Einsatz der Programmgenerierung bei der Lösungssuche für Planungsprobleme auftreten, sind gekennzeichnet.

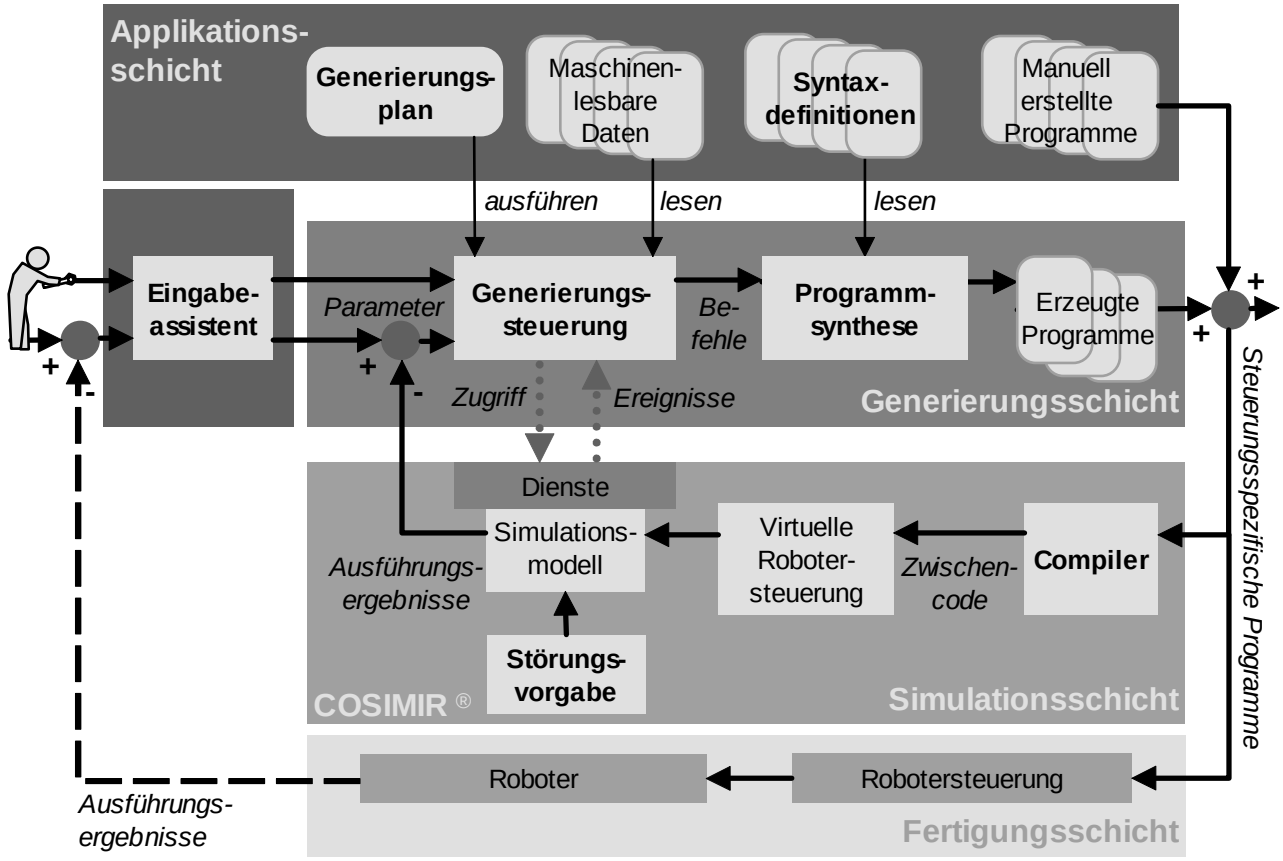


Bild 4.1: Konzept des Gesamtsystems

4.2 Applikationsschicht

Die *Applikationsschicht* enthält die Informationen, die für die Generierung der Roboterprogramme nötig ist. Eine Ausnahme bilden die Informationen (z. B. Geometriedaten, E/A-Belegung der Robotersteuerung), die das Simulationsmodell der Roboter-Fertigungszelle auf der unterlagerten Simulationsschicht zur Verfügung stellt (Anforderung 12, S. 35). Die Applikationsschicht besteht aus zwei Teilen (Bild 4.1):

1. Der erste Teil ist für die Eingabe von applikationsspezifischen Parametern über den *Eingabeassistenten* zuständig. Diese Parameter legen die Erzeugung einer einzelnen Roboterprogrammvariante fest.

2. Der zweite Teil enthält die Informationen, die für die Erzeugung aller möglichen Roboterprogrammvarianten einer Applikation nötig ist, insbesondere die Ablaufbeschreibung und die Steuerungssyntax.

Über den Eingabeassistenten kann der Anwender Parameter eingeben, die die Erzeugung einer einzelnen Roboterprogrammvariante festlegen. Welche Parameter zur Verfügung stehen, hängt von applikationsspezifischen Faktoren (z. B. Fertigungstechnik, Produktspektrum) ab. Für den Poliervorgang der Produktvarianten der Metallplatten (Bild 1.1, S. 2) sind z. B. Parameter für die Bahngeschwindigkeit

und den Anstellwinkel nötig. Mit Letzterem kann der Anwender angeben, wie weit der Roboter die Produktvariante bei Berührung am Kontaktpunkt kippen soll.

Während der Lösungssuche bei Planungsproblemen muss das System in der Regel nicht nur eine Roboterprogrammvariante erzeugen. Hier wird der Eingabeassistent dazu verwendet, die Lösungssuche für das Planungsproblem zu parametrieren. Die Parameter ermöglichen z. B. Einstellungen im Optimierungsverfahren (Schrittweite, maximale Anzahl von Iterationen).

Der zweite Teil der Teil der Applikationsschicht enthält folgende Informationen:

1. Generierungsplan
2. Maschinenlesbare Daten
3. Syntaxdefinitionen
4. Manuell erstellte Roboterprogramme (z. B. Programmbibliotheken)

Der Generierungsplan und eine Syntaxdefinition sind für die Erzeugung aller Roboterprogrammvarianten zwingend erforderlich. Die maschinenlesbaren Daten und die manuell erstellten Programme können optional für eine Applikation zum Einsatz kommen. Für diese Informationen gilt, dass sie zu Beginn einer Programmgenerierung vorliegen müssen, während die Parameterwerte dem System über den Eingabeassistenten erst während der Programmgenerierung zur Verfügung gestellt werden.

Der CAR-Anwender kann aufgrund der Zusammenfassung der Informationen in Dateien das Gesamtsystem für die jeweilige Applikation anpassen und deren spezifische Anforderungen berücksichtigen.

4.2.1 Generierungsplan

Der *Generierungsplan* ist der zentrale Bestandteil des Systemkonzepts. Der CAR-Anwender formuliert den Ablauf der Erzeugung, Ausführung und Bewertung aller Roboterprogrammvarianten einer Applikation in einem Plan. Der Plan enthält sowohl den Erzeugungsalgorithmus (Bild 3.1, S. 30) als auch die Programmspezifikation (Bild 3.3, S. 31). Die Pläne sind mit Skripten aus der Informationsverarbeitung vergleichbar. Ein Plan besteht aus (Bild 4.2):

1. Planvariablen zur Speicherung von Zwischenergebnissen
2. Plananweisungen zur Definition des Ablaufs
3. Eine Folge von Plananweisungen kann zu einem so genannten *Planschritt* (Subroutine) zusammengefasst werden.

Das System führt den Generierungsplan aus und erzeugt dadurch eine neue Roboterprogrammvariante. Der CAR-Anwender kann durch das Konzept der Generierungspläne alle individuellen Anforderungen an die Programmerzeugung für eine Applikation berücksichtigen. Er verwendet dabei für die

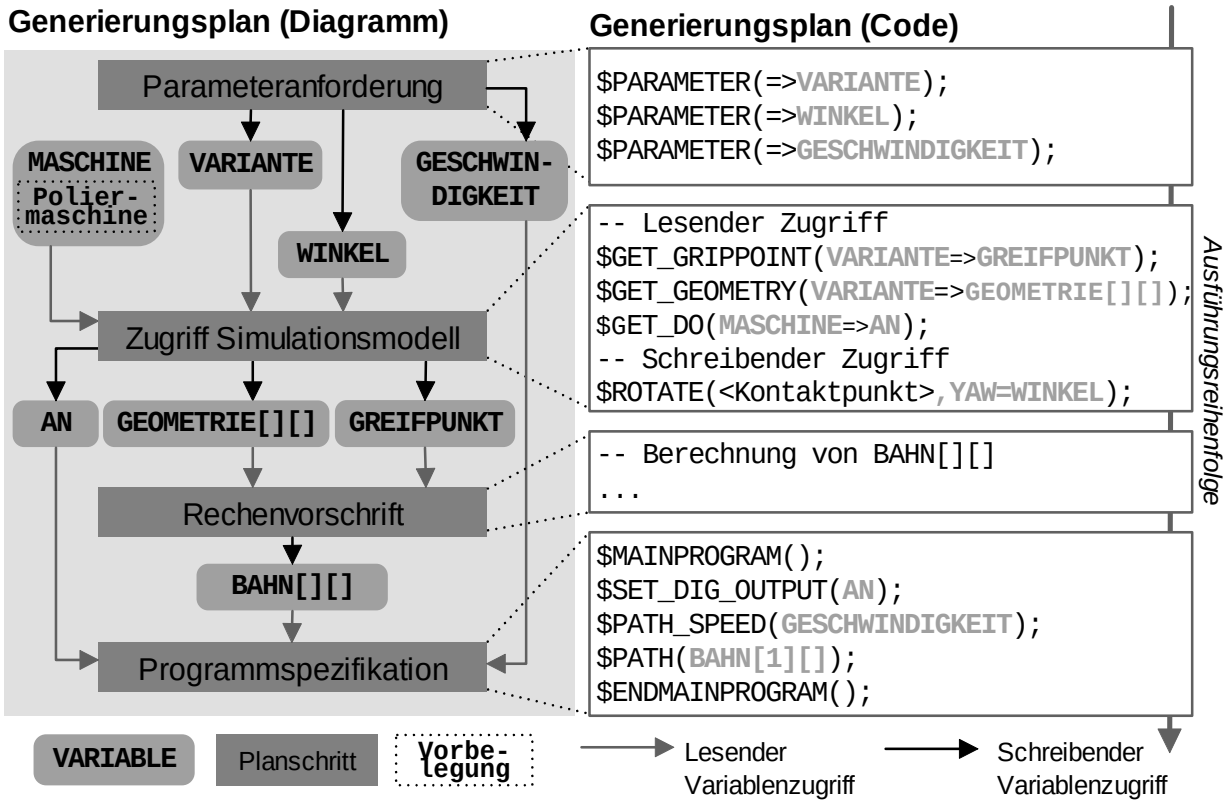


Bild 4.2: Beispiel für einen Generierungsplan

Realisierung eines Plans eine einfache prozedurale Programmiersprache. Das System stellt ihm die Basisfunktionen in Form von vordefinierten Anweisungen (mit Präfix ”$”) zur Verfügung.

Für die Erzeugung der Roboterprogramme zum Polieren der Produktvarianten aus dem einleitenden Beispiel (Bild 1.1, S. 2) ist genau ein Generierungsplan erforderlich. Dieser Plan ist in Bild 4.2 aus Platzgründen vereinfacht dargestellt. Jede Planausführung erzeugt ein neues Roboterprogramm für den Poliervorgang in der Sprache Mitsubishi MELFA BASIC IV. Der Plan teilt den gesamten Ablauf zur Programmgenerierung in Planschritte, die jeweils aus einer Folge von zusammengehörenden Anweisungen bestehen. Ein Plan enthält in der Regel folgende Schritte:

1. Parameteranforderung
2. Zugriff auf das Simulationsmodell
3. Rechenvorschrift
4. Programmspezifikation

Der Schritt *Parameteranforderung* legt alle Parameter fest, die ein Anwender im Eingabeassistenten spezifizieren muss. Der Schritt *Zugriff auf das Simulationsmodell* liest die für die Generierung notwendigen Informationen aus dem Simulationsmodell (z. B. Produktgeometriedaten) oder verändert das Simulationsmodell (z. B. Anfangsstandort der Poliermaschine). Der Schritt *Rechenvorschrift*

berechnet aus den Eingangsinformationen (z. B. Produktgeometriedaten, Parameter) die Bahnstützpunkte für das zu generierende Programm. Der Schritt *Programmspezifikation* enthält die Roboterbefehle (Bild 3.3, S. 31), die in dem zu generierenden Programms auftauchen müssen (z. B. das Ein- und Ausschalten der Poliermaschine über digitale Ein-/Ausgabebefehle). Dabei formuliert der CAR-Anwender diese Befehle steuerungsneutral in Form von Plananweisungen.

Ein Generierungsplan vereint den geforderten Erzeugungsalgorithmus (Erfüllung von Anforderung 1, S. 29) und die geforderte steuerungsneutrale Programmspezifikation (Erfüllung von Anforderung 6, S. 32), die der CAR-Anwender beide unter Verwendung der Skriptsprache formulieren und verändern kann. Für die Planrealisierung stellt das System dem CAR-Anwender häufig benötigte Basisfunktionen in Form von Anweisungen (Präfix "$") zur Verfügung (Erfüllung von Anforderung 2, S. 30).

Bei der Lösung von Planungsproblemen, wie z. B. der Bestimmung eines geeigneten Standortes für die Poliermaschine wird der Algorithmus für die Lösungssuche ebenfalls in einem Generierungsplan angegeben. Anders als bei der Programmgenerierung für Produktvarianten wird während der Ausführung der Generierungspläne nicht nur ein Roboterprogramm erzeugt, sondern eine Vielzahl. Jedes erzeugte Roboterprogramm wird in einem Simulationslauf ausgeführt und der CAR-Anwender kann diese Ausführung im Plan bewerten. Deshalb enthalten solche Pläne noch zwei weitere Schritte:

5. Bewertung
6. Variation

Der Schritt *Bewertung* enthält das Bewertungskriterium für die Programmausführung. In der Praxis ist dies oft die Ausführungszeit. Möglich ist aber auch das Maximum oder die Summe der Achswinkeldifferenzen des Roboters [145, 170] oder der Abstand von Fertigungskomponenten zur Minimierung des Platzbedarfs [48]. Der Schritt *Variation* variiert die Eingangsinformationen (z. B. Parameter, Simulationsmodell), um ein verbessertes Programm zu generieren. Dabei kann der CAR-Anwender Optimierungsverfahren einsetzen, die ihm das System zu Verfügung stellt. Im einleitenden Beispiel variiert dieser Schritt den Standort der Poliermaschine.

4.2.2 Maschinenlesbare Daten

Zu den *maschinenlesbaren Daten* gehören alle applikationsspezifischen Informationen, die man für die Programmgenerierung benötigt und die in einer maschinenlesbaren Form vorliegen. Ein Beispiel für solche Daten sind Listen mit Bahnstützpunkten, die von der Robotersteuerung stammen. Solche Listen kann der CAR-Anwender unter Verwendung vordefinierter Anweisungen dazu einsetzen, die Genauigkeit bei der Roboterpositionierung für berechnete Stützpunkte zu erhöhen.

Beim Roboterfräsen bietet sich ein weiteres Beispiel für maschinenlesbare Daten an. Die zugehörigen Roboterprogramme benötigen in der Regel viele Bahnstützpunkte. Für das NC-Fräsen (Numeric Control) existieren komfortable Softwarelösungen (z. B. das System Tebis der Firma Tebis AG [157]), deren Ausgabe NC-Programme sind. Um den Erhalt der Bahnstützpunkte für das Roboterfräsprogramm

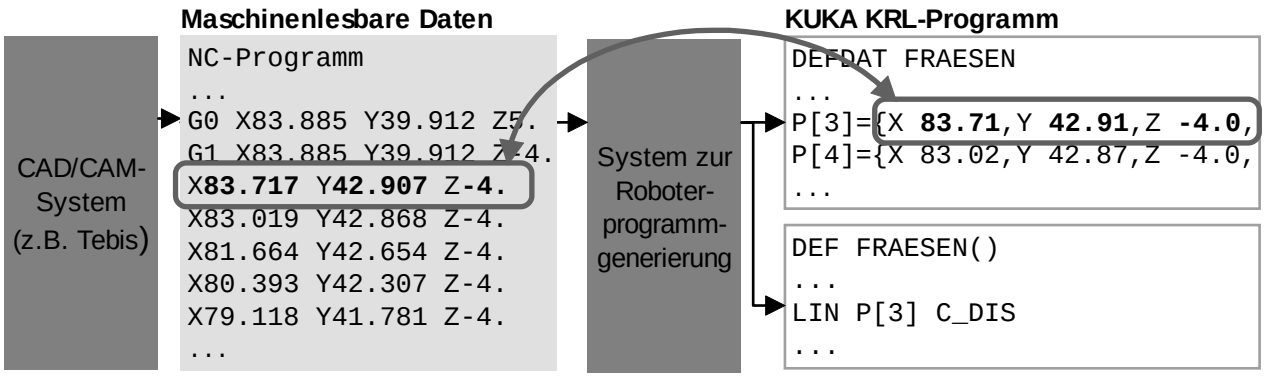


Bild 4.3: Beispiel für maschinenlesbare Informationen beim Roboterfräsen

zu vereinfachen, ist eine einfache Lösung die NC-Stützpunkte aus den zugehörigen NC-Programmen auszulesen und für das Roboterprogramm zu verwenden.

4.2.3 Syntaxdefinition

Die in der Applikationsschicht enthaltene *Syntaxdefinition* enthält die Information über die Syntax und den Aufbau einer Robotersprache. Dazu gehört u. a. die syntaktische Umsetzung aller Roboterbefehle, die der CAR-Anwender im Schritt Programmspezifikation verwenden kann. Jede Robotersprache erfordert genau eine Syntaxdefinition. Der Vorteil, der sich durch die Syntaxdefinitionen ergibt, besteht darin, dass bei syntaktischen Änderungen der Robotersprache nur ein Eingriff in die Syntaxdefinition erfolgen muss (Erfüllung von Anforderung 10, S. 34). Solche syntaktischen Änderungen treten oft auf, wenn ein Roboterhersteller eine neue Steuerungsversion auf den Markt bringt.

In Bild 4.4 sind Auszüge aus zwei Syntaxdefinitionen (KRL, MELFA BASIC IV) abgebildet. Diese weisen Anweisungen der Programmspezifikation (z. B. `$SET_DIG_OUTPUT`) zugehörige Roboterbefehle in der Steuerungssyntax (z. B. für KRL `$OUT` und für MELFA BASIC IV `M_OUT`) zu. Dabei sind Parameter der Programmspezifikation eingerahmt durch das Zeichen ”#” (z. B. `#NUM#`). Dieses Beispiel soll einen ersten Eindruck über das Konzept der Syntaxdefinitionen vermitteln. Die Details werden in einem späteren Abschnitt der Arbeit erläutert.

4.2.4 Manuell erstellte Roboterprogramme

Manuell erstellte Roboterprogramme sind nötig, da der gesamte Funktionsumfang einer Robotersteuerung von einem generierten Programm verwendbar sein soll, um die Anforderungen 6 (S. 32) und Anforderung 7 (S. 32) zu erfüllen. Im Schritt Programmspezifikation kann der CAR-Anwender nur eine Untermenge der Roboterbefehle einer Steuerung angegeben, weil diese steuerungsneutral sein muss, um Anforderung 5 (S. 31) zu erfüllen. Damit ein generiertes Programm trotzdem steuerungsspezifische Befehle verwenden kann, müssen diese in einem manuell erstellten Programm gekapselt werden (Bild 3.4, S. 32).

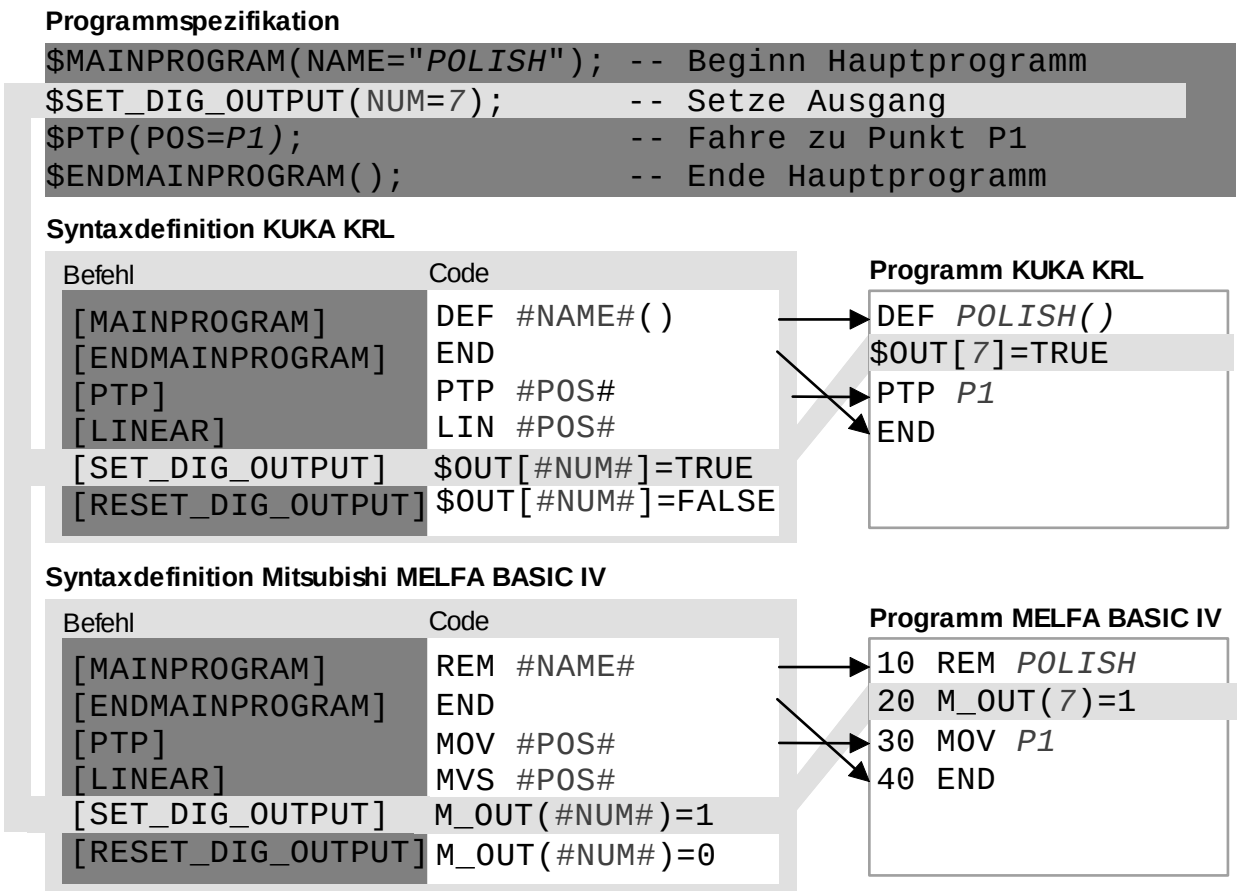


Bild 4.4: Einsatz der Syntaxdefinition

4.3 Generierungsschicht

Die beiden Hauptaufgaben der Generierungsschicht (Bild 4.1, S. 37) sind einerseits die Koordination des Ablaufs und andererseits die Erzeugung der steuerungsspezifischen Roboterprogramme aus den Informationen der überlagerten Applikationsschicht und der unterlagerten Simulationsschicht. Die Generierungsschicht besteht aus den Systemmodulen Generierungssteuerung und Programmsynthese.

4.3.1 Generierungssteuerung

Die *Generierungssteuerung* ist verantwortlich für die Koordination des gesamten Ablaufs. Sie bezieht ihre Informationen über den Ablauf aus den Generierungsplänen der überlagerten Applikationsschicht. Die Aufgaben der Generierungssteuerung sind im Einzelnen:

1. Auf- und Abbau der Repräsentation zur Ausführung der Generierungspläne
2. Ausführung der Generierungspläne
3. Bereitstellung der Basisfunktionen für die Verwendung in den Generierungsplänen
4. Weiterleiten der bearbeiteten Programmspezifikation an die Programmsynthese

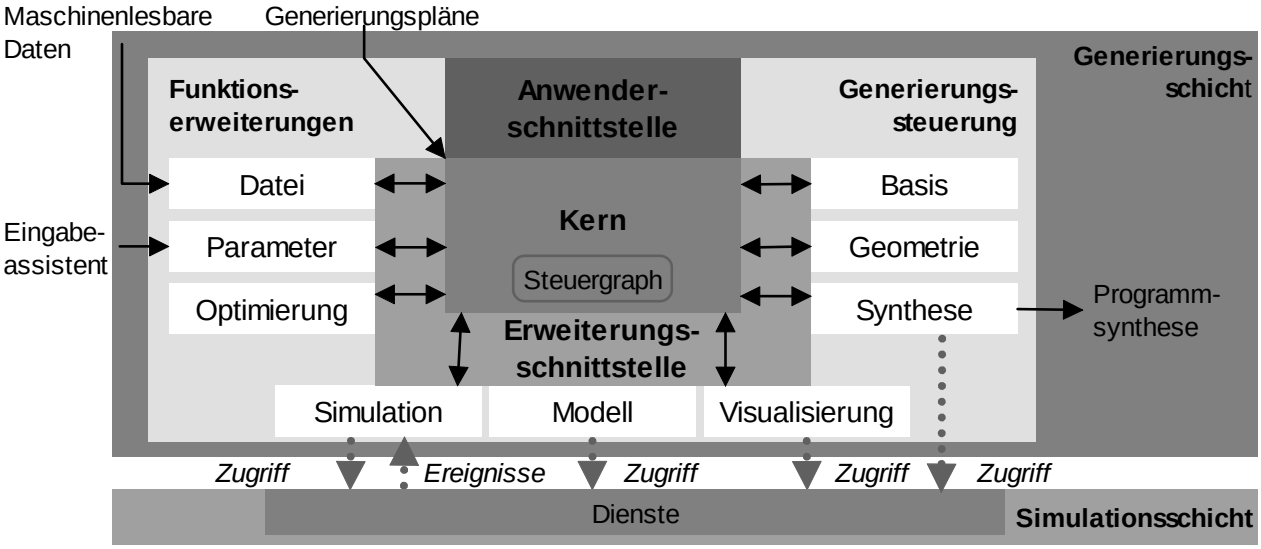


Bild 4.5: Aufbau der Generierungssteuerung

5. Kommunikation mit der unterlagerten Simulationsschicht

Die Generierungssteuerung benötigt für die Planausführung eine systeminterne Repräsentation der Generierungspläne, den so genannten *Steuergraphen*. Das System führt die Generierungspläne aus, indem es den Steuergraphen durchläuft.

Aufbau der Generierungssteuerung

Die Generierungssteuerung besteht aus (Bild 4.5):

- Anwenderschnittstelle
- Kern
- Funktionserweiterungen
- Erweiterungsschnittstelle

Über die *Anwenderschnittstelle* kann der CAR-Anwender die Planausführung kontrollieren (starten, stoppen, unterbrechen). Der *Kern* ist für den Auf- und Abbau und für den Durchlauf des Steuergraphen verantwortlich. Die *Funktionserweiterungen* sind über eine *Erweiterungsschnittstelle* an den Kern gekoppelt und enthalten u. a. die Basisfunktionen, die der CAR-Anwender für die Implementierung der Generierungspläne häufig benötigt. Für jede Funktionserweiterung existiert eine *Schnittstellenbeschreibung*. Diese enthält einerseits die Plansyntax der Basisfunktionen für den CAR-Anwender und andererseits die Aufrufinformationen der Basisfunktionen für den Kern. Funktionserweiterungen können Dienste der Simulationsschicht (z. B. Zugriff auf das Simulationsmodell) verwenden. Je nach

Abhängigkeit von der Simulationsschicht werden simulationsschichtabhängige (z. B. Modell) und simulationsschichtunabhängige Erweiterungen (z. B. Geometrie) unterschieden. Es existieren z. B. Erweiterungen mit mathematischen Funktionen, mit numerischen Optimierungsverfahren (z. B. Hooke-Jeeves [72], Simulated-Annealing [113]) und für Berechnungen von zwei- und dreidimensionalen geometrischen Objekten (z. B. Strecken und Stauchen von Polygonen [104]).

Aufgrund der vielen Faktoren, von denen ein erfolgreicher Einsatz einer Programmgenerierung abhängt, kann kein System alle Basisfunktionen von Beginn an zu Verfügung stellen [78]. Durch die Funktionserweiterungen stellt das System Basisfunktionen zur Verfügung, die der CAR-Anwender häufig benötigt (Erfüllung von Anforderung 2, S. 30). Diese Funktionen kann der CAR-Anwender in Form von Plananweisungen in den Generierungsplänen verwenden (Erfüllung von Anforderung 3, S. 30). Um neue Basisfunktionen in einem Plan verwenden zu können, ist die Realisierung einer Funktionserweiterung erforderlich. Dazu muss man vom System nur die Kenntnis der Erweiterungsschnittstelle besitzen. Über die Erweiterungsschnittstelle ist es u. a. möglich, Planvariablen zu lesen und zu schreiben.

Schnittstelle zwischen Generierungssteuerung und Simulationsschicht

Die Generierungssteuerung verwendet die folgenden Dienste der Simulationsschicht:

1. Sie kann lesend und schreibend auf das Simulationsmodell der Simulationsschicht zugreifen.
2. Sie kann vor einem Simulationslauf Anfragen an die Simulationsschicht stellen, z. B. ob der Roboter einen Bahnstützpunkt erreichen kann, und ob dabei Kollisionen auftreten.
3. Sie kann nach einem Simulationslauf Simulationsergebnisse abfragen, z. B. die Ausführungszeiten der einzelnen Roboter.
4. Die Simulationsschicht schickt *Ereignisse* an die Generierungssteuerung, um sie über Geschehnisse während des Simulationslaufs zu informieren (z. B. Beendigung Programmausführung, Kollisionen, Überschreitung von Geschwindigkeits- oder Beschleunigungsvorgaben).

Die Simulationsschicht stellt der Generierungssteuerung Funktionen zur Verfügung, die es ihr erlauben, Informationen aus dem Simulationsmodell zu lesen (z. B. E/A-Belegung der Robotersteuerung) oder das Simulationsmodell zu verändern. Letzteres ist z. B. bei der Layoutoptimierung der Roboter-Fertigungszelle nötig, weil der CAR-Anwender in seinem Generierungsplan Fertigungskomponenten verschieben muss, um ein günstigeres Layout zu finden. Durch den Zugriff der Generierungssteuerung auf das Simulationsmodell wird Anforderung 11 (S. 34) erfüllt.

Die Ereignisse, die die Simulationsschicht an die Generierungssteuerung schickt, ermöglichen es Letzterer, frühzeitig auf Fehler während eines Simulationslaufs zu reagieren. Der CAR-Anwender kann in den Plänen angeben, wie auf Fehler aus der Simulationsschicht reagiert werden soll (z. B. Abbruch des Simulationslaufs bei Auftritt einer Kollision).

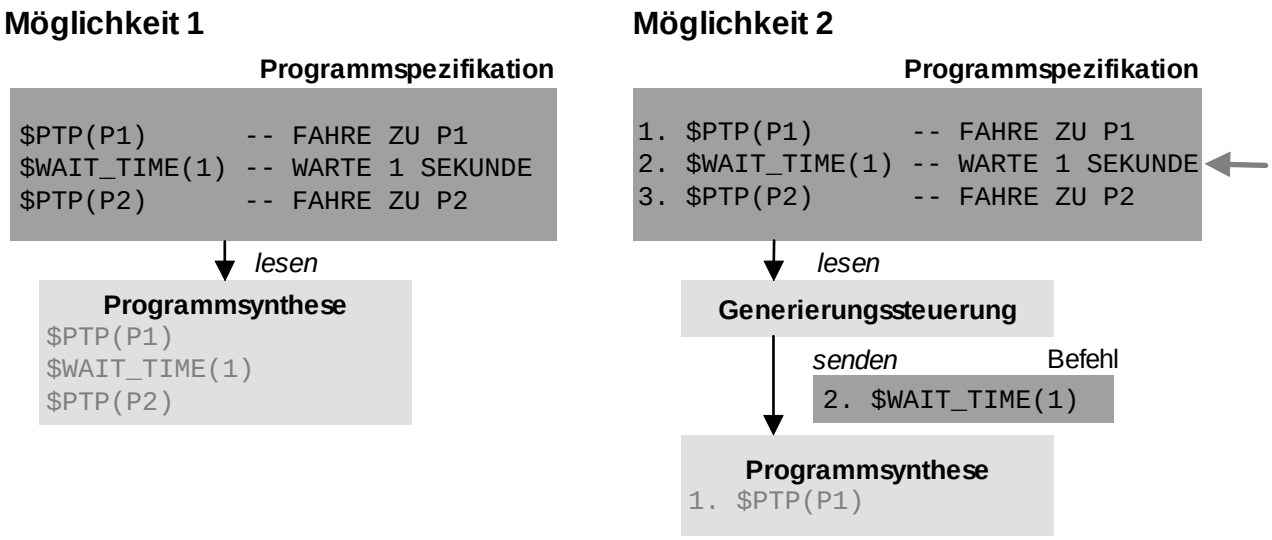


Bild 4.6: Möglichkeiten zur Verarbeitung der Programmspezifikation

4.3.2 Programmsynthese

Die *Programmsynthese* ist verantwortlich für die Erzeugung der steuerungsspezifischen Roboterprogramme (Bild 4.7). Die Informationen, welche Befehle im Roboterprogramm in welcher Reihenfolge auftauchen sollen, stammen aus der Programmspezifikation im Generierungsplan. Die konkrete Syntax der Roboterbefehle entnimmt die Programmsynthese aus der zugehörigen Syntaxdefinition. Für die Verarbeitung der Programmspezifikation existieren zwei Möglichkeiten (Bild 4.6):

1. Die Programmsynthese liest die Programmspezifikation zu einem Zeitpunkt in vollem Umfang.

2. Die Programmsynthese erhält die Befehle der Programmspezifikation einzeln über eine gewisse Zeitspanne.

Der Nachteil der ersten Möglichkeit liegt an der fehlenden Möglichkeit, Entscheidungen zu berücksichtigen (Anforderung 9, S. 33). Die hat zur Folge, dass alle generierten Programmvarianten immer dieselben Befehle besitzen müssen. Dieser Nachteil besteht bei der zweiten Möglichkeit nicht. Aus diesem Grund wird die zweite Möglichkeit verfolgt.

Bei der zweiten Möglichkeit liest die Generierungssteuerung die Programmspezifikation und sendet deren einzelne *Programmbefehle* an die Programmsynthese. Neben den Programmbefehlen muss die Programmsynthese *Steuerbefehle* erhalten (Bild 4.7):

- *Programmbefehle*: Diese Befehle enthalten Angaben über den Inhalt des zu generierenden Programms. Bei jedem Empfang eines Programmbefehls fügt die Programmsynthese diesen der internen Repräsentation des zu generierenden Roboterprogramms hinzu (der Programmbefehl `$PTP` fügt z. B.einen PTP-Bewegungsbefehl hinzu).

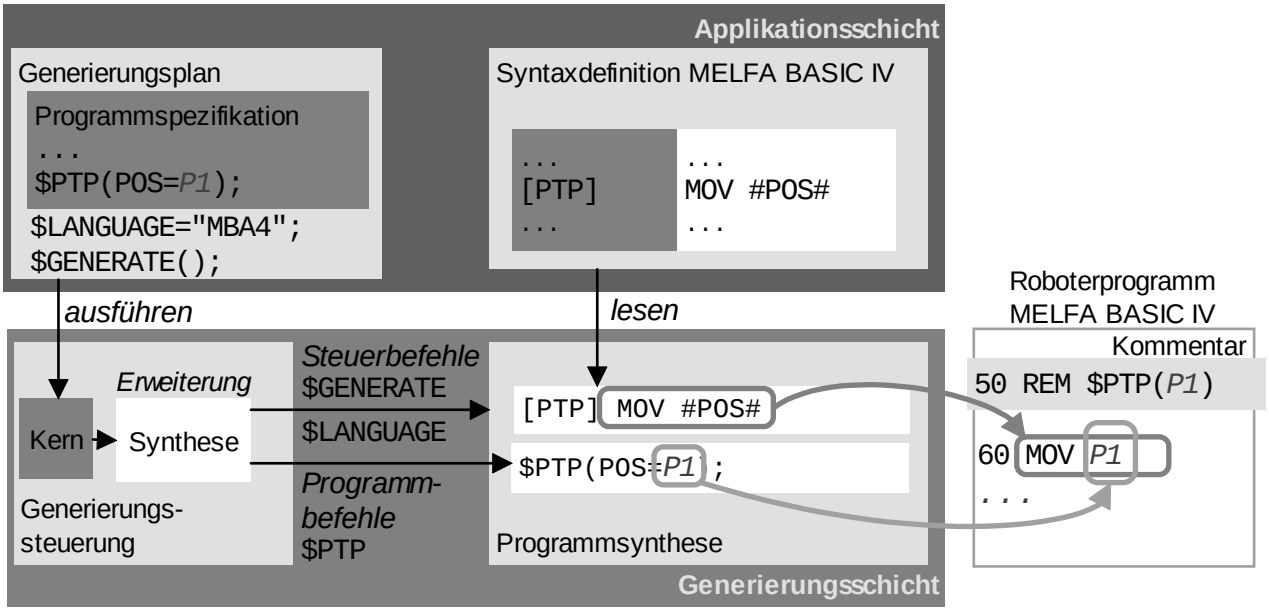


Bild 4.7: Konzept der Programmsynthese

- *Steuerbefehle*: Diese Befehle dienen zur Steuerung der Programmsynthese selbst (z. B. Ausgabe des zu generierenden Programms durch den Steuerbefehl `$GENERATE` oder der Steuerbefehl `$LANGUAGE` zur Festlegung, in welcher Steuerungssyntax die Programmsynthese das Programm ausgeben soll).

Neben den eigentlichen Roboterbefehlen fügt die Programmsynthese Kommentare in das steuerungsspezifische Roboterprogramm ein. In den Kommentaren sind die Generierungsplananweisungen des Schritts Programmspezifikation enthalten (Bild 4.7), die für die Erzeugung der nachfolgenden Befehle im Roboterprogramm verantwortlich sind (Erfüllung von Anforderung 8, S. 33).

Da die Befehle für die Programmspezifikation als Anweisungen in den Generierungsplänen auftauchen, ist es möglich, Programmspezifikationen zu formulieren, die abhängig von der Ausführung der Pläne sind (z. B. die Hinzunahme eines Bewegungsbefehls, um Hindernissen auszuweichen). Somit kann ein Generierungsplan während seiner Ausführung Roboterprogramme mit jeweils unterschiedlichen Roboterbefehlen erzeugen (Erfüllung von Anforderung 9, S. 33)

4.4 Simulationsschicht

Die Simulationsschicht (Bild 4.1, S. 37) wird durch das am Institut für Roboterforschung entwickelte CAR-System COSIMIR® realisiert. Der Gründe hierfür sind dessen Verfügbarkeit, dessen Offenheit [161] und dessen bereitgestellte Funktionen, die im weiteren Verlauf erläutert werden. Die Simulationsschicht übernimmt folgende Aufgaben im Gesamtsystem:

- Die Bereitstellung von Informationen für die Programmgenerierung (z. B. Werkstückgeometrie, E/A-Belegung der Robotersteuerung), die im Simulationsmodell enthalten sind. Dabei verwen-

det die Generierungssteuerung die von der Simulationsschicht angebotenen Dienste für den Zugriff auf das Simulationsmodell.

- Die Ausführung der Programme durch eine *virtuelle Robotersteuerung* (VRC).
- Die Bereitstellung von Ausführungsergebnissen (z. B. Ausführungszeiten) für die in einem Simulationslauf ausgeführten steuerungsspezifischen Roboterprogramme.
- Die Benachrichtigung der Generierungssteuerung durch *Ereignisse* über das Auftreten besonderer Geschehnisse, z. B. eines Ausführungsfehlers während des Simulationslaufs.

Die Simulationsschicht enthält die folgenden für diese Arbeit relevanten Module:

1. Simulationsmodell
2. Virtuelle Robotersteuerung (VRC)
3. Compiler
4. Störungsvorgabe

Im Rahmen dieser Arbeit wurde ein Beitrag an den Compilern und der Störungsvorgabe geleistet. Das Simulationsmodell und die virtuelle Robotersteuerung werden aus Gründen der Verständlichkeit kurz erläutert. Diese beiden Module wurden nicht im Rahmen dieser Arbeit entwickelt.

4.4.1 Simulationsmodell

Eine genaue Beschreibung des Simulationsmodells von COSIMIR® (Bild 4.8) findet sich in ROSSMANN [137] und UTHOFF [161]. Das Simulationsmodell enthält verschiedene Abstraktionsstufen zur Strukturierung der Roboter-Fertigungszelle (*Objekt, Gruppe, Polyeder, Polygon*). Die geometrische Repräsentation der Zellenkomponenten wird in Form von *Polyeder* beschrieben, die wiederum als unsortierte Folgen von *Polygonen* dargestellt werden. Für jedes Polygon legt der die Oberfläche einer Komponente beschreiben. Jedes Polygon wird durch eine sortierte Folge von *Raumpunkten* dargestellt. Diese Sortierung dient zur Definition eines Umlaufsinns Normalenvektor den Umlaufsinn (Gegenuhrzeigersinn) des Polygons fest.

Das Simulationsmodell enthält *Objekte*, die Objekten in der realen Roboter-Fertigungszelle entsprechen (z. B. Roboter, Greifer, Produkt). Jedem Objekt ist eindeutig ein *Objekttyp* zugeordnet. Das Modell teilt die Objekttypen wie folgt ein:

1. Aktive Objekte können selbstständig Bewegungen ausführen (z. B. Roboter, lineare oder rotatorische Achsen, Fließbänder, Greifer).
2. Passive Objekte können keine selbstständigen Bewegungen ausführen (z. B. Werkstücke, Halterungen, Tische).

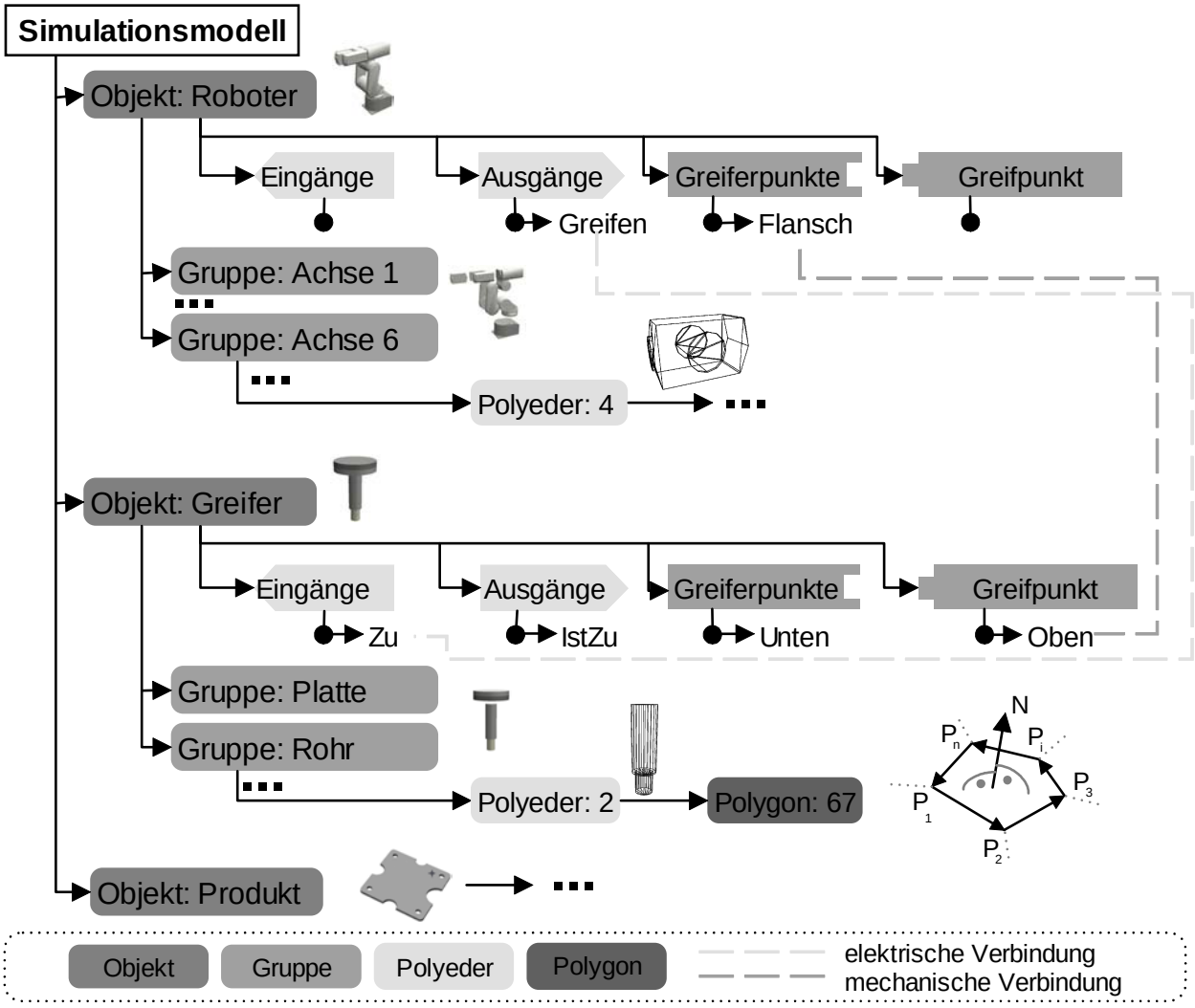


Bild 4.8: Simulationsmodells des CAR-Systems COSIMIR™

Das Simulationsmodell unterscheidet die aktiven Objekte wiederum in programmierbare aktive Objekte (z. B. Roboter) und in aktive Objekte mit vordefiniertem Verhalten (z. B. Greifer). Letztere besitzen *Eingänge*, mit denen man das Verhalten steuern kann und *Ausgänge*, die den Zustand eines Objekts anzeigen (z. B. Greifer offen oder geschlossen). Man kann Ein- und Ausgänge miteinander verbinden. Dadurch werden *elektrische Verbindungen* zwischen realen Objekten modelliert. Zusätzlich besteht die Möglichkeit, einen Eingangswert dauerhaft auf eins oder null zu erzwingen.

Mechanische Verbindungen zwischen Objekten, z. B. ein Greifer nimmt ein Produkt auf, modelliert man durch die Angabe von so genannten *Greif-* und *Greiferpunkten* an den beteiligten Objekten. Liegen ein Greifpunkt und ein Greiferpunkt an derselben Stelle im Raum, so können die beteiligten Objekte eine mechanische Verbindung eingehen, die man auch wieder lösen kann. Der Magnetgreifer mit einem Greiferpunkt ”greift” z. B. Metallplatten am Greifpunkt und lässt diese auch wieder los.

Die Robotermodelle enthalten für die zu generierenden Programme folgende wesentlichen Angaben:

- Anzahl der Achsen und deren Typ (rotatorisch oder translatorisch)
- Achsgrenzwerte, Achsgeschwindigkeit und Achsbeschleunigung

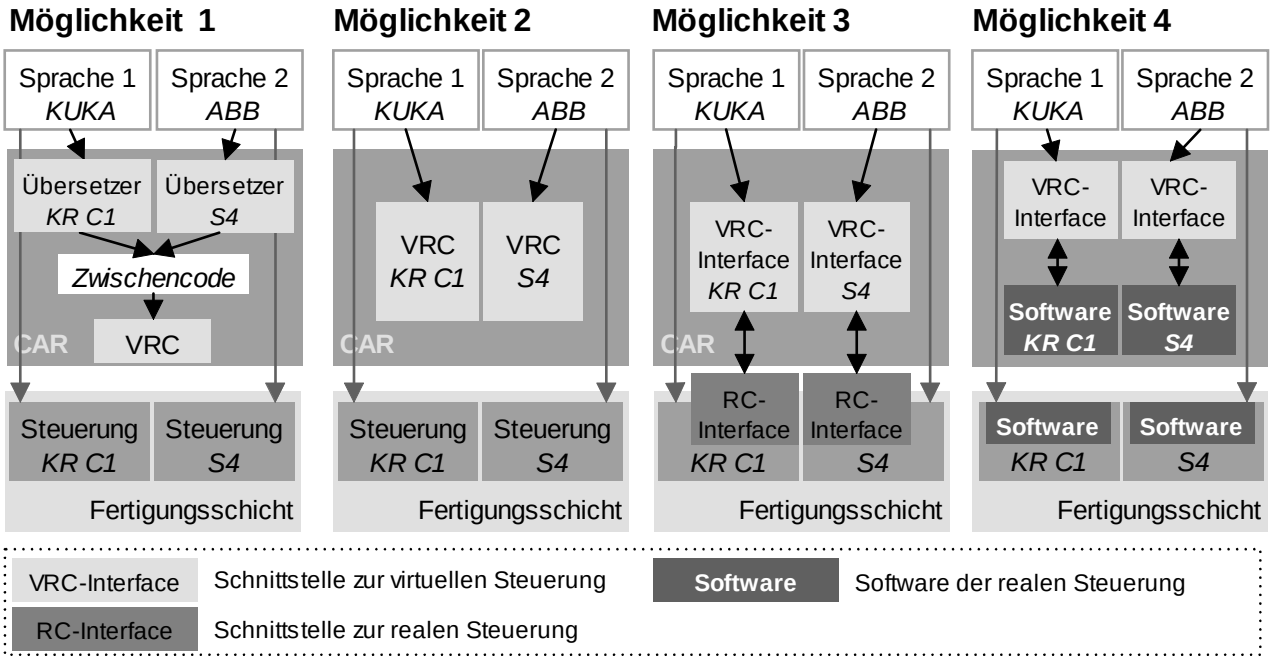


Bild 4.9: Möglichkeiten zur Anbindung einer virtuellen Robotersteuerung an ein CAR-System

- Bahngeschwindigkeit, Bahnbeschleunigung
- E/A-Belegung (Anzahl und Namen von digitalen und analogen E/A, Verbindungen zu Geräten)
- Lage und Orientierung des Tool Center Points (TCP)

Ein generiertes Programm benötigt oftmals diese Informationen, z. B. zur Ansteuerung der Werkzeuge. Für die Datenhaltung im Gesamtsystem existieren zwei Möglichkeiten: Bei der ersten Möglichkeit werden diese Daten in der Applikationsschicht dupliziert, weil die Applikationsschicht die Daten für die Generierung aller Produktvarianten enthält. Bei der zweiten Möglichkeit werden die Daten im Simulationsmodell belassen. In diesem Systemkonzept wird die zweite Möglichkeit verfolgt, damit der Speicherbedarf nicht unnötig hoch ist und damit es keine Mechanismen zur Sicherstellung der Konsistenz bedarf.

4.4.2 Virtuelle Robotersteuerung

Um Anforderung 11 (S. 34) zu erfüllen, ist für die Ausführung der steuerungsspezifischen Programme in der Simulationsschicht eine virtuelle Steuerung nötig. Zu deren Anbindung an ein CAR-System zur Ausführung steuerungsspezifischer Programme gibt es vier Möglichkeiten (Bild 4.9):

1. Die erste Möglichkeit schlägt DILLMANN [37] vor. Ein sprachabhängiger Übersetzer wandelt das steuerungsspezifische Programm in einen einheitlichen Zwischencode (z. B. IRDATA [4], ICR [1] und Controller Modeling Language [112]), den eine VRC ausführt.
2. Bei der zweiten Möglichkeit setzt man für jede reale Steuerung eine spezifische VRC ein.

3. Bei der dritten Möglichkeit wird eine reale Steuerung (ohne Roboter) an das CAR-System angebunden (Hardware-In-The-Loop). Dazu sind Schnittstellen aufseiten der VRC und auf Seite des CAR-Systems nötig. Einige herstellerspezifische Systeme (z. B. KUKA KR Sim [93] und Invision [80]) verfolgen diese Möglichkeit.

4. Bei der vierten Möglichkeit wird die Steuerungssoftware von der Steuerungshardware getrennt und nur die Steuerungssoftware über eine geeignete Schnittstelle an das CAR-System angebunden. Das Forschungsprojekt Realistic Robot Simulation II (RRS-II) [18] verfolgt diesen Ansatz durch die Definition einer offenen Schnittstelle. Das System RobotStudio™ [7] beruht ebenfalls auf dieser Möglichkeit und verwendet eine ABB-interne Schnittstelle.

Bei der dritten und der vierten Möglichkeit sind Schnittstellen notwendig, die der Hersteller zur Verfügung stellen muss. Allerdings bietet nicht jeder Hersteller solche Schnittstellen an und es planen auch nicht alle, solche Schnittstellen anzubieten. Bei der vierten Möglichkeit kommt die Schwierigkeit der kosten- und zeitintensiven Trennung der Steuerungssoftware von der Steuerungshardware hinzu, die nur der Hersteller durchführen kann. Ein weiterer Nachteil der dritten Möglichkeit besteht in der notwendigen Verfügbarkeit der realen Steuerung. Der Vorteil der dritten und der vierten Möglichkeit besteht darin, dass man den vollständigen Sprachumfang der Steuerung verwenden kann. Bei der dritten Möglichkeit hat man zusätzlich den großen Vorteil, dass das simulierte Steuerungsverhalten, verglichen mit dem realen Verhalten (inklusive Fehlverhalten) in der Roboter-Fertigungszelle nahezu gleich ist. Benötigt man aber einen Vergleich von mehreren Steuerungen, um zu entscheiden, welcher Roboter eine Aufgabe am besten erledigen kann, so ist dies mit der dritten Möglichkeit praktisch unmöglich, weil dazu alle zu betrachtenden Steuerungen verfügbar sein müssen.

Der Nachteil der ersten und zweiten Möglichkeit ist der hohe Realisierungsaufwand. Allerdings sind dies die einzigen Möglichkeiten, mit der man die Ausführung steuerungsspezifischer Roboterprogramme für möglichst viele Hersteller in der Praxis durchführen kann. Dies ermöglicht die Erfüllung der Anforderung 5 (S. 31) in Zusammenhang mit Anforderung 11 (S. 34) und den Vergleich von Robotern von unterschiedlichen Herstellern auf der Basis von steuerungsspezifischen Programmen.

Das System COSIMIR® beruht für Programme in der Sprache BAPS3 (Bewegungs-und Ablauf-Programmiersprache) auf der dritten Möglichkeit, da BAUER [17] die Robotersteuerung Bosch rho4 über die offene Schnittstelle OPC (Open Process Control) an COSIMIR® angebunden hat. Sonst beruht COSIMIR® auf der ersten Möglichkeit und besitzt eine Robotersteuerung, die als Zwischencode eine erweiterte Form von IRDATA benötigt (VAN DER VALK [162], KEIBEL [47, 84]). Das in dieser Arbeit entwickelte System beruht deshalb für Programme in der Sprache BAPS auf der dritten Möglichkeit und verwendet im Übrigen die erste Möglichkeit. Die zweite Möglichkeit wird ausgeschlossen, da keine steuerungsspezifische VRC für COSIMIR® existiert. Die vierte Möglichkeit wird nicht ausgeschlossen und soll zukünftig über die RRS-II-Schnittstelle realisiert werden, aber noch existiert keine entsprechende VRC mit dieser Schnittstelle.

Um den Nachteil der ersten Möglichkeit abzuschwächen, d. h. den hohen Realisierungsaufwand zu senken, wird im folgenden Abschnitt ein in dieser Arbeit entwickeltes Konzept dargestellt.

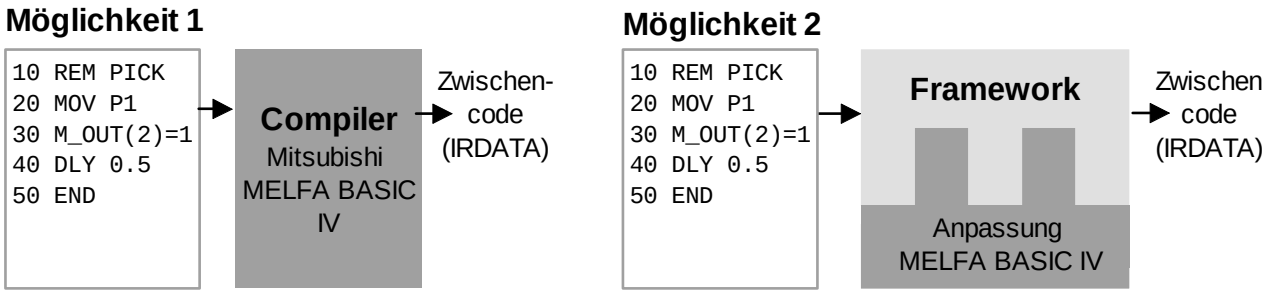


Bild 4.10: Möglichkeiten zur Realisierung von Compilern

4.4.3 Compiler

Für die Realisierung der ersten Möglichkeit (Bild 4.9) wird für jede Sprache einer Steuerung ein Übersetzer benötigt. Will man den Sprachumfang möglichst vollständig unterstützen (z.B. Modulkonzept, Ausdrücke, unterschiedliche Gültigkeitsbereiche für Variablen), reichen Präprozessoren für die Übersetzung nicht aus, sondern es sind umfassende Compiler erforderlich. Die Hauptschwierigkeit bei der Entwicklung solcher Compiler liegt in den gewachsenen Strukturen der heutigen Robotersprachen. Die Roboterhersteller haben aufgrund von Kundenanforderungen ihre Steuerungen um neue Funktionen ergänzt und die Verwendung dieser Funktionen auf verschiedene Arten, z. B. als Variable, als neuen Befehl oder als neuen Parameter in einem bestehenden Befehl, in der Robotersprache abgebildet. Die Folge war, dass man die zu Anfang festgelegten Strukturen der Sprachen missachtet hat. Deshalb muss man in den Compilern viele Ausnahmefälle behandeln. Für die Realisierung von Compilern existieren zwei Möglichkeiten (Bild 4.10):

1. Für jede Robotersprache wird ein kompletter Compiler zur Verfügung gestellt.

2. Die Funktionen, die jeder Compiler benötigt, sind in einem Framework zusammengefasst. Das Framework entspricht einem Standardcompiler, den man nur für die Ausnahmefälle und die spezifischen Spracheigenschaften anpassen muss.

Der Nachteil der ersten Möglichkeit besteht in einem hohen Realisierungsaufwand. Bei der zweiten Möglichkeit (Bild 4.10) muss man für die Realisierung eines neuen Compilers ähnliche Funktionen nicht erneut implementieren. Diese Arbeit verfolgt deshalb die zweite Möglichkeit [51].

4.4.4 Störungsvorgabe

Um die generierten und manuellen Roboterprogramme gegenüber möglichen Störungen zu prüfen, muss es möglich sein, Störungen in das Simulationsmodell einzubringen (Bild 4.11). Aus diesem Grund wurde die *Störungsvorgabe* entwickelt, die im Folgenden kurz erläutert wird. In Abhängigkeit eines Objekttyps kann der CAR-Anwender einem Objekt eine Störung zuordnen (z. B. Stillstand, Ausfall). Für jede Störung ordnet er dieser einen Startzeitpunkt und eine Dauer zu. Während des

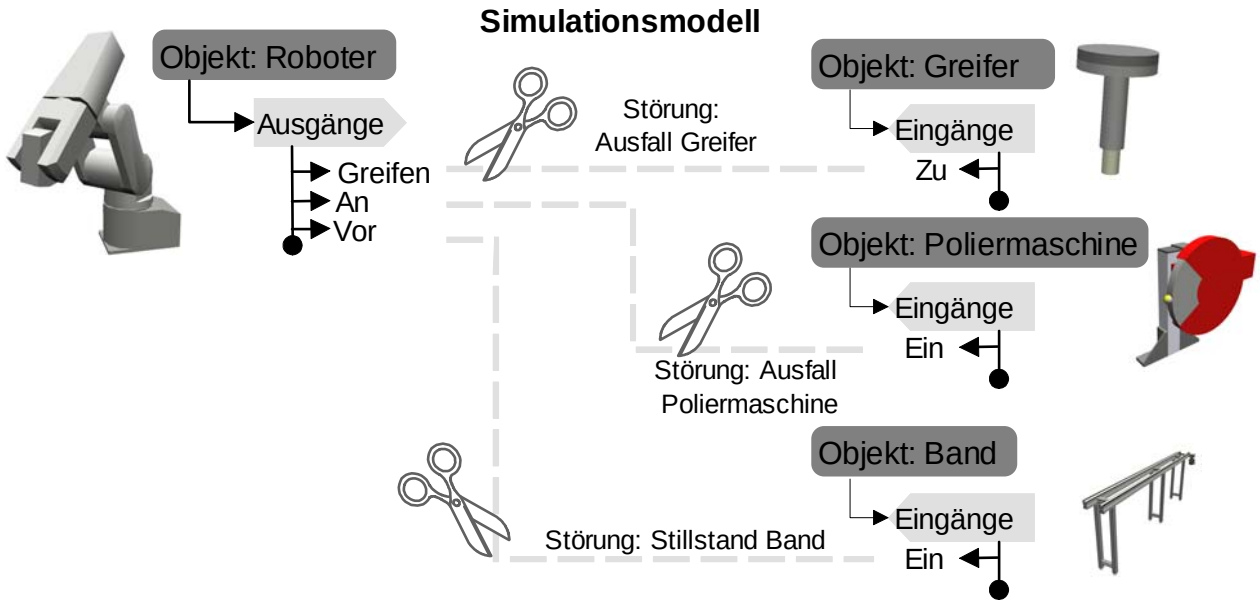


Bild 4.11: Modellierung von Störungen im Simulationsmodell

Simulationslaufs tritt die Störung ein, sobald die Simulationslaufdauer den Startzeitpunkt überschritten hat. Bei Störungseintritt ändert die Störungsvorgabe die elektrischen Verbindungen des gestörten Objekts. Bei Ausfall des Greifers z. B. trennt die Störungsvorgabe die Verbindung zum Roboter und erzwingt den zugehörigen Greifereingang auf null. Nachdem die zu einer Störung zugeordnete Dauer überschritten wurde, stellt die Störungsvorgabe die ursprüngliche Verbindung wieder her.

Auf diese Weise kann das Verhalten der steuerungsspezifischen Roboterprogrammen bei Eintritt von Störungen vorab geprüft werden, bevor die Programe in der realen Roboter-Fertigungszelle eingesetzt werden.

4.5 Fertigungsschicht

Die Fertigungsschicht ist identisch mit der Roboter-Fertigungszelle und enthält deren Hard- und Software. Für die Planung einer Roboter-Fertigungszelle existiert diese Schicht nicht. In der Fertigungsschicht in Bild 4.1 (S. 37) sind aufgrund der vielen unterschiedlichen Fertigungskomponenten in der Praxis, nur die wichtigsten Komponenten abgebildet.

Die Fertigungsschicht hat die Verantwortung der realen Ausführung der Roboterprogramme. Die Rückmeldung über das Ausführungsergebnis für eine Verbesserung der Programmgenerierung erfolgt in der Regel durch die Beobachtung des Anwenders. Für Einzelfälle ist es denkbar, dass man diese Schicht in den automatisierten Ablauf mit entsprechend aufwendiger Sensorik einbindet.

5 Generierungspläne

Nach dem Überblick über das Systemkonzept werden in diesem Kapitel die Generierungspläne der Applikationsschicht genauer betrachtet (Bild 4.1, S. 37), die bereits in Abschnitt 4.2.1 kurz erläutert wurden. Zunächst werden die Konzepte der Plansprache beschrieben. Darauf aufbauend wird das Konzept der Planschablonen erläutert, das die Realisierung von Plänen für ein bestimmtes Einsatzgebiet vereinfacht. Die Darstellung der Verwendung der Schablonen an zwei konkreten Beispielen, die in Anhang C vollständig abgedruckt sind, bildet den Abschluss dieses Kapitels.

5.1 Sprachkonzepte

Nach dem Duden der Informatik ist eine Programmiersprache eine Sprache zur Definition von Rechenvorschriften [30]. Eine Programmiersprache stellt verschiedene Programmierkonzepte zur Verfügung. Man unterscheidet die Sprachen bezüglich der enthaltenen Programmierkonzepte (z. B. prozedurale, funktionale, logische und objektorientierte Sprachen).

5.1.1 Entwurfskriterien

Der CAR-Anwender beschreibt unter Verwendung der Generierungsplansprache den Ablauf, der bei seiner Applikation für die Generierung, Ausführung und Bewertung der steuerungsspezifischen Roboterprogramme nötig ist. Zur Verarbeitung der Pläne, insbesondere für das Parsen, sollen standardisierte Softwarewerkzeuge (z. B. Lex, Yacc [69]) zum Einsatz kommen. Demnach orientiert sich der Entwurf der Generierungsplansprache an den folgenden beiden Kriterien:

1. Der CAR-Anwender muss die Sprache schnell erlernen können.
2. Die Grammatik der Sprache muss die formale Eigenschaft LALR(1) besitzen [10, 69].

Damit der CAR-Anwender die Sprache schnell erlernen kann, enthält sie überwiegend Sprachkonzepte, die der CAR-Anwender von den Roboterprogrammiersprachen kennt. Es handelt sich um eine prozedurale Sprache mit den Konzepten:

1. Module
2. Variablen
3. Ausdrücke
4. Routinen und Funktionen

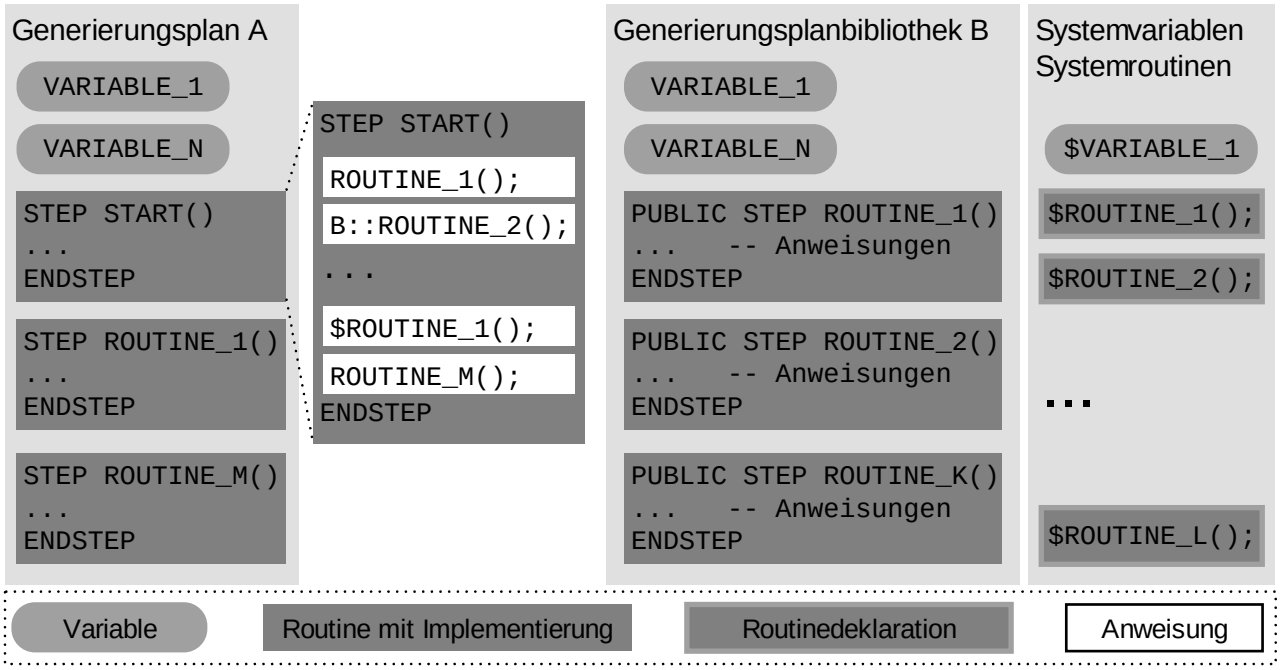


Bild 5.1: Organisation von Generierungsplänen

Die Sprache enthält bewusst keine objektorientierten Konzepte (z. B. Vererbung, dynamisches Binden, Überladen von Operatoren) oder Konzepte der logischen und funktionalen Programmierung (z. B. Unifikation), um die Einarbeitung in die Sprache möglichst kurz zu halten. Die Plangrammatik ist in Anhang A (S. 164ff) abgedruckt.

Den mit dem Generierungsplan beschriebenen Ablauf kann man in Schritte (z. B. Parameteranforderung, Zugriff Simulationsmodell, Rechenvorschrift, Programmspezifikation) zerlegen (Bild 4.2, S. 39). Jeder Schritt wird im Generierungsplan als Routine implementiert. Das System stellt vordefinierte Funktionen (*Systemfunktionen*) und Routinen (*Systemroutinen*), zusammen im Folgenden nur Systemroutinen genannt, zur Verfügung, die der CAR-Anwender aufrufen kann. Der CAR-Anwender kann in Plänen nur Routinen implementieren, aber keine Funktionen.

5.1.2 Module

Der Generierungsplan und die Planbibliothek (Bild 5.1) stellen Module dar. Ein Modul enthält eine Menge an Variablen und Routinen. Die Routinen bestehen aus einer Folge von Anweisungen.

Der Generierungsplan enthält eine Startroutine, bei der die Planausführung beginnt und die weitere Routinen und Systemfunktionen in Form von Anweisungen aufruft. Die Implementierung einer anwenderdefinierten Routine befindet sich entweder in demselben Plan wie der Routinenaufruf oder sie befindet sich in einer Planbibliothek. Die Implementierung einer vordefinierten Systemroutine befindet sich in einer Funktionserweiterung der Generierungssteuerung.

Die Routinen, die der CAR-Anwender in einem Generierungsplan implementiert, kann er nur in demselben Plan aufrufen. Aus Gründen der Wiederverwendung bietet sich für häufig benötigte Routinen deren Implementierung in einer Planbibliothek an.

Das System stellt die geforderten Basisfunktionen (Anforderung 2, S. 30) in Form von vordefinierten Systemroutinen zur Verfügung (z. B. Prüfung auf Erreichbarkeit eines Bahnstützpunkts). Neben Systemroutinen bietet das System auch *Systemvariablen* an. Der CAR-Anwender kann aus einer Systemvariablen entweder Informationen über den Systemzustand auslesen (z. B. Roboterposition im Simulationsmodell) oder sie zur Parametrierung des Systems verwenden (z. B. Angabe der Syntax der zu generierenden Roboterprogramme). Zur Kennzeichnung von Systemroutinen und Systemvariablen besitzen diese das Präfix ”$”. In dieser Arbeit implementierte Systemroutinen und Systemvariablen sind, geordnet nach der Zugehörigkeit zur Funktionserweiterung, in Anhang B (S. 169ff) abgedruckt.

Applikationsunabhängige Routinen stellt entweder das System zur Verfügung oder der CAR-Anwender realisiert sie, falls keine Systemroutinen mit benötigter Funktion vorhanden sind, in eigenen Planbibliotheken. Applikationsspezifische Routinen implementiert der CAR-Anwender in einem Plan.

5.1.3 Variablen

Variablen in den Generierungsplänen dienen zur Zwischenspeicherung von Rechenergebnissen und zum Datenaustausch (Bild 4.2, S. 39).

Datentypen

Die Plansprache ist streng typisiert und enthält vordefinierte Datentypen aus dem Umfeld der Roboterprogrammierung und Robotersimulation (Tabelle 5.1). Durch die strenge Typisierung kann das System viele Fehler in den Plänen bereits vor deren Ausführung finden.

Analog zur Kennzeichnung der Systemvariablen und Systemroutinen besitzen die Namen der vordefinierten Datentypen das Präfix ”$”. Der CAR-Anwender kann eigene Datentypen definieren, die er aus vordefinierten Datentypen zusammensetzt.

Deklaration

Es existieren unterschiedliche Gültigkeitsbereiche für die Variablen in den Plänen. Systemvariablen sind global, d. h. sie sind in jedem Generierungsplan und in jeder Planbibliothek verwendbar. Zusätzlich kann der CAR-Anwender in einem Plan *planlokale* und *routinenlokale* Variablen deklarieren. Planlokale Variablen kann der CAR-Anwender nur in einem Plan verwenden und routinenlokale Variable nur in einer Routine. Die Deklarationen von planlokalen Variablen stehen am Plananfang. Die Deklaration von routinenlokalen Variablen stehen an einer beliebigen Stelle in einer Routine, allerdings vor deren erstmaliger Verwendung. In einer Planbibliothek besteht die Möglichkeit planlokale Variablen zu deklarieren, die aber nur in Routinen derselben Bibliothek verwendbar sind. Auf solche Variablen kann ein Plan nicht direkt, sondern nur über eine Bibliotheksroutine, zugreifen.

Variablen, die von einem vordefinierten Datentyp sind, können bei ihrer Deklaration auch einen Initialisierungswert erhalten. Bei der Initialisierung von planlokalen Variablen sind nur Konstanten zulässig. Im Gegensatz dazu kann der CAR-Anwender bei der Deklaration einer routinenlokalen Variablen

Tabelle 5.1: Vordefinierte Datentypen

Art	Plan Syntax	Beschreibung
Elementar	`$NUMBER`	Reelle Zahlen
	`$TEXT`	Zeichenketten
	`$BOOL`	Boolescher Wert (wahr, falsch)
Geometrie	`$PLANE`	Ebene im Raum
	`$LINE`	Gerade im Raum
	`$SPHERE`	Kugel
	`$BOX`	Quader
	`$CYLINDER`	Zylinder
	`$VECTOR`	Raumpunkt, Vektor
	`$ORIENT`	Orientierung im Raum
	`$FRAME`	Punkt und Orientierung im Raum (Lage)
Simulationsmodell	`$OBJECT`	Name eines Objekts im Simulationsmodell
	`$SECTION`	Name einer Gruppe im Simulationsmodell
Roboter	`$POSITION`	Bahnstützpunkt (Frame, Konfiguration, Turn, Zusatzachsen, Bezugsobjekt)
	`$AXIS`	Roboterachsstellung
System	`$EVENT`	Ereignis

```
Datentyp Name(n)    Initialisierungswert (optional)      Kommentar (optional)
$NUMBER  WINKEL     = 10;                                -- Anstellwinkel(Grad)
$NUMBER  I,J        = 1;                                 -- Mehrfache Initialisierung
$OBJECT  MASCHINE   = <Poliermaschine>;                  -- Name im Simulationsmodell
$OBJECT  PRODUKT    = <Produkt C>;                       -- Name im Simulationsmodell
$VECTOR  P0         = $CREATE_VECTOR(XMIN,YMIN,Z);       -- Definiton mit Ausdruck
```

Bild 5.2: Beispiele für Variablendeklarationen

vollständige Ausdrücke mit bereits deklarierten Variablen angeben. Die Beispieldeklarationen (Bild 5.2) stammen aus einem Generierungsplan für die Erzeugung der Polierprogramme (S. 2). Ein doppeltes Minuszeichen kennzeichnet den Anfang eines Kommentars, der bis zum Zeilenende reicht.

Multidimensionale Felder unbeschränkter Größe

Die Sprache stellt multidimensionale Felder mit unbeschränkter Größe zur Verfügung. Das bedeutet, dass der CAR-Anwender zu Beginn der Planausführung die maximale Zahl der zu speichernden Elemente für jede Dimension offen lassen kann. Diese Felder wachsen dynamisch mit deren Verwendung, ohne dass der CAR-Anwender Anweisungen zum Allokieren und Freigeben von Speicher implementieren muss (Garbage Collection). Bei einem Zugriff wird, falls nicht bereits vorhanden, ein entsprechendes Element vom zugehörigen Basistyp erzeugt und das Feld gegebenenfalls vergrößert. Die Deklaration eines zweidimensionalen Feldes zur Speicherung mehrerer Roboterbahnen (Folge von Stützpunkten) in einem Generierungsplan lautet z. B. wie folgt:

Bild 5.3: Ausdrücke in Generierungsplänen

```
$POSITION[][]  BAHN  -- BAHN[I][J]: Bahn i mit Stützpunkt j
```

Diese Felder erweisen sich als praktisch, weil deren Größe vor der Planausführung oft unbekannt ist. Der CAR-Anwender muss dadurch nicht eine Planausführung wiederholen, die aufgrund einer zu geringen Feldgröße abgebrochen wurde. Für den Poliervorgang im einleitenden Beispiel (S. 2) hängt z. B. die Anzahl der Bahnstützpunkte von den Geometriedaten des Produktrandes ab.

Die Felder eignen sich auch für die Repräsentation von Polyedern (Bild 4.8, S. 48), da deren Größe vor der Planausführung unbekannt ist (z. B. für die Produktvarianten A-E, S. 2). Einfache geometrische Körper (z. B. Kugel, Quader) kann der CAR-Anwender in Variablen speichern, die von einem vordefinierten Geometriedatentyp sind (Tabelle 5.1). Ein Polyeder (Folge von Polygonen) aus dem Simulationsmodell lässt sich in einem zweidimensionalen Feld von Punkten speichern:

```
$VECTOR[][]  GEOMETRIE  -- GEOMETRIE[I][J]: Polygon i mit Punkt j
```

Beim Zugriff auf die Elemente eines multidimensionalen Feldes kann man die hinteren Indizes weglassen. Z. B. repräsentiert `GEOMETRIE[][]` einen vollständigen Polyeder, `GEOMETRIE[i][]` repräsentiert ein Oberflächenpolygon `i` des Polyeders und `GEOMETRIE[i][j]` repräsentiert den Punkt `j` des Oberflächenpolygons `i`.

5.1.4 Ausdrücke

Wie aus anderen Programmiersprachen bekannt, ist auch die Verwendung von Ausdrücken in der Plansprache möglich. Einen Überblick über die möglichen Ausdrücke mit Angabe der Priorität für deren Auswertung zeigt Bild 5.3. Einen besonders wichtigen Operator in dem eingesetzten Umfeld des Systems ist der Operator ”*” zur Berechnung der homogenen Transformation. Die Unterscheidung zur Multiplikation mit dem Operator ”*” erfolgt in der semantischen Analyse über die Datentypen der zugehörigen Operanden. Die Multiplikation benötigt Operanden vom Typ `$NUMBER`. Die homogene Transformation benötigt Operanden vom Typ `$FRAME`.

5.1.5 Routinen

Eine Routine in einem Plan oder in einer Planbibliothek besteht aus einer Folge von Anweisungen. Die Plansprache stellt dazu vordefinierte Anweisungen zur Verfügung. Sie bietet Zuweisungen und Kontrollanweisungen an, wie sie aus prozeduralen Programmiersprachen bekannt sind (Anhang A, S. 164ff). Dazu zählen u. a. die bedingte Anweisung (`IF`), Schleifen (`WHILE`, `UNTIL`, `FOR`) und spezielle Sprunganweisungen (`BREAK`, `CONTINUE`, `RETURN`). Das zentrale Sprachkonzept für die Planerstellung sind jedoch die vom CAR-Anwender angegebenen Routinen.

Definition

Die Definition einer anwenderdefinierten Routine in einem Plan ist wie folgt aufgebaut:

```
STEP NAME(...,TYP_i^IN NAME_i^IN=INIT_i^IN,...=>..., TYP_j^OUT NAME_j^OUT=INIT_j^OUT,...)
... -- Anweisungen
ENDSTEP
```

Eine Routine kann Eingangsparameter (call-by-value) und Ausgangsparameter (call-by-reference) besitzen, die durch einen Pfeil (=>) getrennt sind. Die Trennung wurde gewählt, um klar herauszustellen, welche Parameter eine Routine benötigt und welche sie liefert. Streng genommen können die Ausgangsparameter auch zur Eingabe missbraucht werden. Die Parameter sind typisiert ($\texttt{TYP}_i^{IN}$ bzw. $\texttt{TYP}_j^{OUT}$) und besitzen jeweils einen Namen ($\texttt{NAME}_i^{IN}$ bzw. $\texttt{NAME}_j^{OUT}$). Vor jedem $\texttt{TYP}_i^{IN}$ kann das Schlüsselwort `CONST` stehen. Damit legt der CAR-Anwender fest, dass der zugehörige Eingabeparameter $\texttt{NAME}_i^{IN}$ in der Routine nicht änderbar ist. Jeder Parameter mit einem vordefinierten Datentyp kann einen Initialisierungswert erhalten ($\texttt{INIT}_i^{IN}$ bzw. $\texttt{INIT}_j^{OUT}$). Das Schlüsselwort `PUBLIC` wird vor das Schlüsselwort `STEP` gestellt, wenn es sich um eine Bibliotheksroutine handelt (Bild 5.1). Für deren Aufruf ist keine zusätzliche Deklaration im aufrufenden Plan erforderlich.

Aufrufkonvention

Der Aufruf einer Routine hat folgendes Aussehen:

```
NAME(...,NAME_i^IN=PARAM_i^IN,...=>..., NAME_j^OUT=PARAM_j^OUT,...);
```

Wird eine Bibliotheksroutine aufgerufen, so wird vor den Routinennamen der Bibliotheksname, gefolgt von einem "`::`", gestellt (`B::ROUTINE_2()` in der Startroutine von Bild 5.1). Die Eingangsparameter und die Ausgangsparameter sind im Aufruf, analog zur Deklaration, ebenfalls getrennt, um zu verdeutlichen, welche Parameter die Routine benötigt ($\texttt{PARAM}_i^{IN}$) und welche sie liefert ($\texttt{PARAM}_j^{OUT}$). Beim Aufruf können einzelne Parameter entfallen. Ein nicht aufgeführter Parameter ($\texttt{PARAM}_i^{IN}$ bzw. $\texttt{PARAM}_j^{OUT}$) erhält den Initialisierungswert ($\texttt{INIT}_i^{IN}$ bzw. $\texttt{INIT}_j^{OUT}$). Der Vorteil, der sich durch die Möglichkeit des Weglassens von Parametern ergibt, besteht darin, dass der CAR-Anwender einer existierenden Routine einen neuen Parameter hinzufügen kann, ohne dass er nachträglich die zugehörigen, existierenden Aufrufe ändern muss. Ein weiterer Vorteil besteht darin, dass der CAR-Anwender

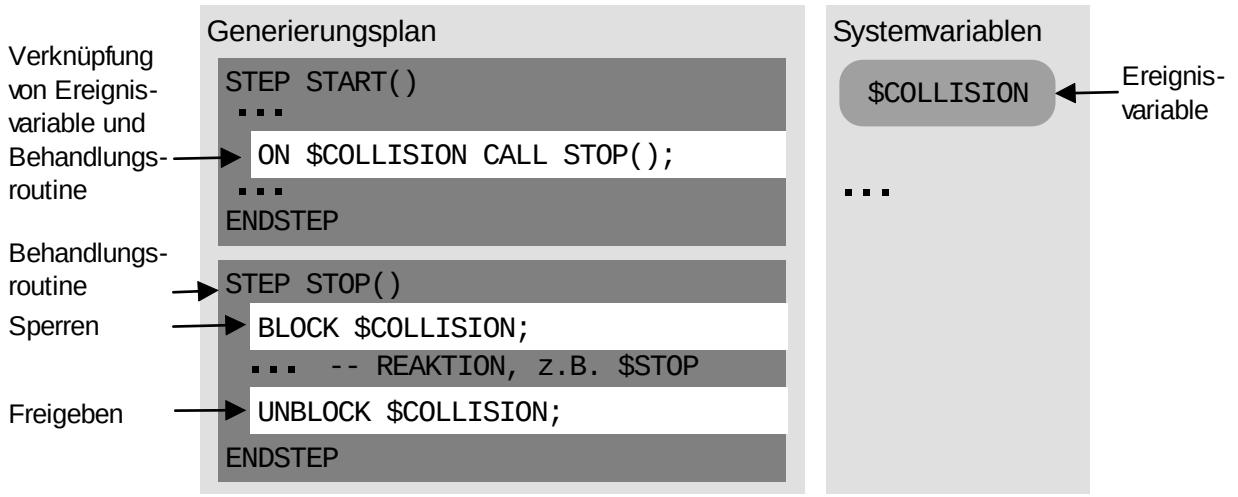


Bild 5.4: Reaktion auf ein Ereignis aus der Simulationsschicht

einen nicht benötigten Ausgangsparameter weglassen kann und dadurch der Aufruf nur so kurz wie nötig wird.

Die Parameternamen ($\texttt{NAME}_i^{IN}$ bzw. $\texttt{NAME}_j^{OUT}$) muss der CAR-Anwender nicht angeben, sofern die Zuordnung der Parameter der Deklaration und des Aufrufs eindeutig ist. Wenn er die Parameternamen beim Aufruf angibt, so ist die Reihenfolge der Parameter änderbar. Der Vorteil, der sich durch die Angabe der Namen ergibt, ist die Verständlichkeit, wenn sprechende Parameternamen vergeben werden, weil die Bedeutung eines Parameters bereits im Aufruf erkennbar ist.

Reaktion auf Ereignisse

Tritt während des Simulationslaufs ein Ausführungsfehler der Roboterprogramme auf, z. B. eine Kollision, so sendet die Simulationsschicht ein entsprechendes Ereignis an die Generierungsschicht (Abschnitt 4.3.1, S. 44). Damit der CAR-Anwender applikationsspezifisch auf ein Ereignis reagieren kann, muss die Plansprache ein entsprechendes Konzept enthalten. Eine flexible Lösung ist die Verknüpfung des Ereignisses mit einem oder mehreren Routinenaufrufen (Bild 5.4).

Für jedes Ereignis stellt das System eine zugehörige Variable (z. B. `$COLLISION`) vom Typ `$EVENT` zur Verfügung. Eine solche Variable lässt sich dann mit einem oder mehreren Routinenaufrufen verknüpfen, die die Ereignisbehandlung durchführen (z. B. `ON $COLLISION CALL STOP()`). Die zugehörige Routine (z. B. `STOP`) muss als erste Anweisung das Ereignis sperren (`BLOCK`), und als letzte Anweisung das Ereignis wieder freigeben (`UNBLOCK`), damit die Ereignisbehandlung selbst nicht unterbrochen wird. Der Simulationslauf selbst wird bei Eintritt des Ereignisses nicht unterbrochen. Eine Unterbrechung wird aber durch den Aufruf der Systemroutine `$STOP` erreicht. Die Ereignisse werden systemintern in eine Warteschlange eingereiht. Eine Priorisierung der Ereignisse ist nicht erforderlich.

Dieses Konzept ermöglicht dem CAR-Anwender eine applikationsspezifische Reaktion auf die während des Simulationslaufs auftretenden Fehler.

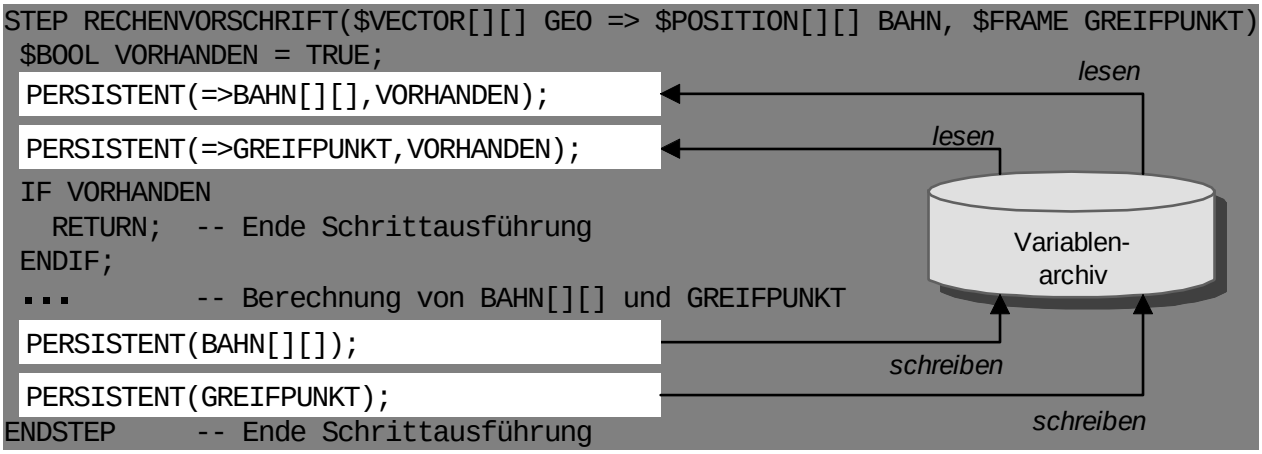


Bild 5.5: Mechanismus zur Beschleunigung einer Schrittausführung

Persistenz

Während der Planentwicklung werden oftmals am Anfang stehende Routinen öfter ausgeführt als am Ende stehende Routinen, insbesondere wenn der CAR-Anwender am Ende stehende Routinen ändern muss und er den Plan zu Testzwecken mehrmals ausführt. Für rechenintensive Schritte (z. B. `RECHENVORSCHRIFT` für geometrische Berechnungen) führt das folgende Konzept (Bild 5.5) der Variablenpersistenz zu einer wesentlichen Verkürzung der Gesamtentwicklungszeit, weil es die mehrmalige Ausführung am Anfang stehender Routinen verkürzt.

Für jeden Ausgangsparameter einer Routine kann der CAR-Anwender am Anfang eine Anweisung `PERSISTENT` mit zwei Ausgangsparametern und am Ende eine Anweisung `PERSISTENT` mit einem Eingangsparameter angeben. Die erste Anweisung stellt eine Anfrage an ein Variablenarchiv, ob eine Variable mit einem solchen Namen darin existiert und liefert im ersten Ausgangsparameter - falls vorhanden - den Wert. Die zweite `PERSISTENT`-Anweisung schreibt die Variable zusammen mit ihrem Wert in das Variablenarchiv. Damit `PERSISTENT`-Anweisungen für mehrere Variablen direkt hintereinander stehen können, wird der boolesche Wert des zweiten Ausgangsparameters mit der folgenden Aussage durch die boolesche Operation UND verknüpft, ob die Variable im Variablenarchiv vorhanden ist. Diese Vorgehensweise erspart zusätzliche Plananweisungen zur Berechnung der notwendigen Hilfsvariablen (`VORHANDEN`, Bild 5.5).

In Bild 5.5 ist die Routine ohne weitere Berechnungen beendet, wenn die Variablenwerte im Archiv vorhanden sind. Im anderen Fall werden die rechenintensiven Anweisungen ausgeführt. An deren Ende wird die Variable mit ihrem Wert in das Archiv geschrieben. Wenn man die rechenintensiven Anweisungen erneut ausführen möchte, so muss man die Variable aus dem Archiv löschen. Das Archiv ist eine lesbare Datei, in der man Variablen manuell oder über Plananweisungen löschen kann.

Durch dieses Konzept der Persistenz wird die mehrmalige Ausführung rechenintensiver Routinen verkürzt, sodass man die zu testenden nachfolgenden Schritte früher erreicht.

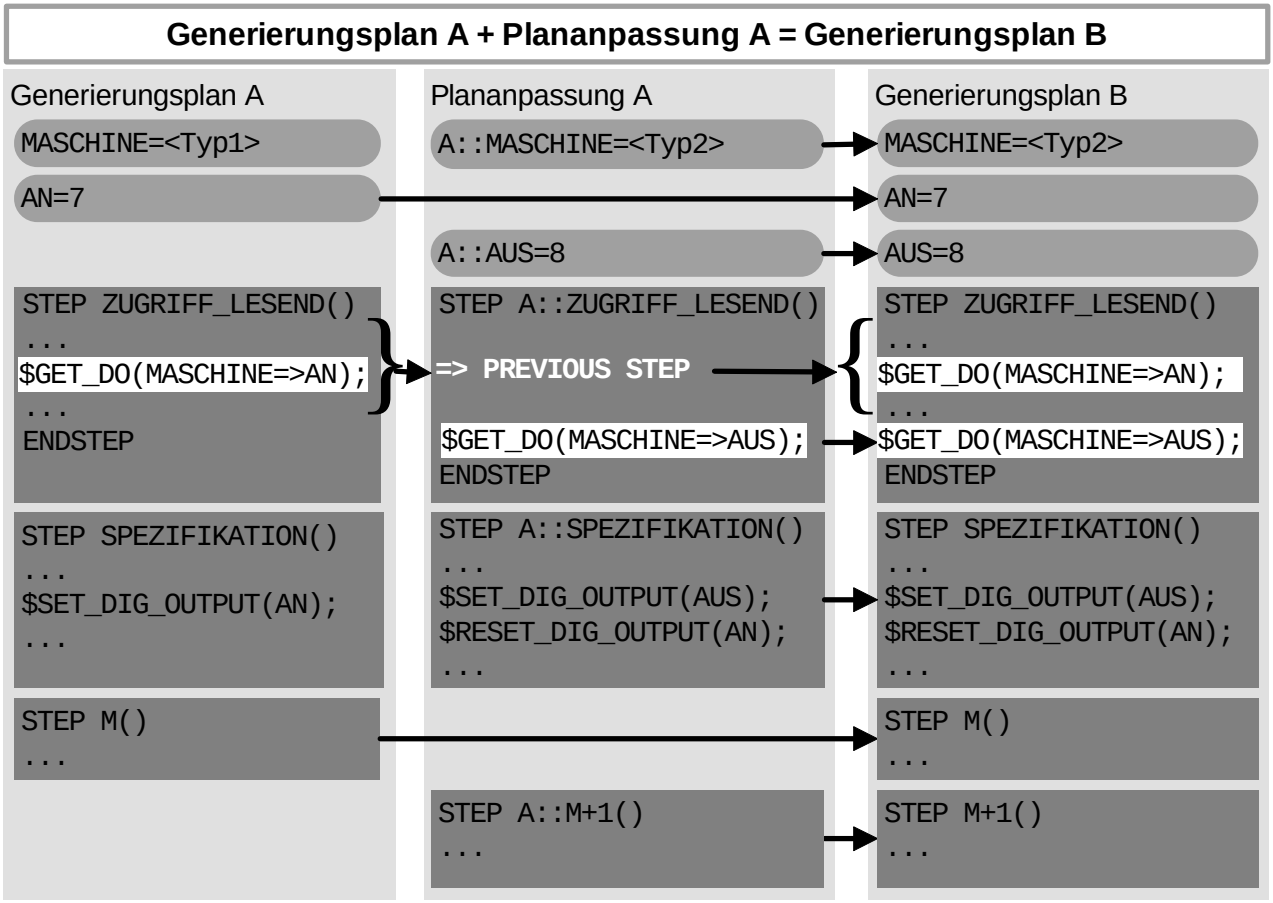


Bild 5.6: Erstellung eines neuen Plans durch "'Addition"' eines Plans und einer Anpassung

5.1.6 Planaddition

Praktische Erfahrungen haben gezeigt, dass der CAR-Anwender oftmals einen neuen Generierungsplan erstellen muss, der nur kleine Änderungen an einem existierenden Plan erfordert (z. B. Ergänzung neuer Befehle für das zu generierende Roboterprogramm). Die Vorgehensweise zur Erstellung dieses neuen Planes war ursprünglich zunächst das Anlegen einer Kopie des existierenden Plans und die anschließende Änderung der Kopie. Problematisch sind nachträgliche Änderungen am existierenden Plan, denn diese muss man im neuen Plan manuell nachführen.

Für das einleitende Beispiel (Bild 1.1, S. 2) möchte der CAR-Anwender z. B. statt der ursprünglichen Poliermaschine (Typ 1) eine neue Poliermaschine eines anderen Herstellers einsetzen (Typ 2). Die Ansteuerung der neuen Poliermaschine besitzt zwei Eingänge, im Gegensatz zur ursprünglichen Poliermaschine, die nur einen Eingang benötigt. Für das Ein- und Ausschalten der neuen Poliermaschine müssen die Eingangswerte invertiert zu einander sein. Die zu generierenden Programme für die Poliermaschine vom Typ 2 müssen demnach zusätzliche Befehle zum Ein- und Ausschalten erhalten. Für eine Erstellung von Planvarianten dient das folgende neue Konzept der *Planaddition* (Bild 5.6).

Durch eine Planaddition kann man einen neuen Plan aus einem existierenden Plan erstellen. Dazu muss man in einem speziellen Plan (*Plananpassung*) nur die zu ändernden Routinen und Variablen angeben, der existierende Plan bleibt unverändert.

Für die ”Addition” gelten die folgenden Regeln. In Klammern sind Beispiele aus Bild 5.6 angegeben.

1. Ist eine Variable in der Plananpassung und im existierenden Plan vorhanden (Name und Typ sind gleich), so trägt das System die Variable mit dem Wert aus der Plananpassung als planlokale Variable in den neuen Plan ein (z. B. Variable `MASCHINE` mit Wert <Typ2>).

2. Enthält der existierende Plan eine Variable, die nicht in der Plananpassung vorhanden ist, so trägt das System diese Variable als planlokale Variable in den neuen Plan ein (z. B. Variable `AN`). Umgekehrt gilt, enthält die Plananpassung eine Variable, die nicht im existierenden Plan vorhanden ist, so trägt das System diese Variable als planlokale Variable in den neuen Plan ein (z. B. Variable `AUS`).

3. Enthält der existierende Plan eine Routine, der nicht in der Plananpassung vorhanden ist, so übernimmt das System diese Routine in den neuen Plan (z. B. Schritt `STEP M`). Umgekehrt gilt, enthält die Plananpassung eine Routine, der nicht in dem existierenden Plan vorhanden ist, so übernimmt das System diese Routine in den neuen Plan (z. B. Schritt `STEP M+1`).

4. Ist eine Routine in der Plananpassung und im existierenden Plan vorhanden, so trägt das System die Routine aus der Plananpassung in den neuen Plan ein (z. B. `SPEZIFIKATION` und `ZUGRIFF_LESEND`). Taucht in dieser Routine der Plananpassung die Anweisung `=>PREVIOUS STEP` auf, so ersetzt das System die Anweisung `=>PREVIOUS STEP` in der neuen Routine durch alle Anweisungen der existierenden Routine mit demselben Namen (z. B. `ZUGRIFF_LESEND`).

Die Planaddition ist ein leistungsfähiges Konzept, um durch Angabe von Änderungen an bestehenden Plänen neue Pläne zu erstellen, ohne die bestehenden Pläne selbst zu ändern. Die Änderungen am bestehenden Plan können Folgendes betreffen:

1. Hinzufügen einer neuen Variablen (z. B. `AUS`)

2. Ändern des Initialisierungswerts einer bestehenden Variablen (z. B. `MASCHINE`)

3. Hinzufügen einer neuen Routine (z. B. Routine `M`)

4. Ersetzen einer bestehenden Routine (z. B. Routine `SPEZIFIKATION`)

5. Anweisungen entweder am Routinenanfang oder -ende hinzufügen (z. B. `ZUGRIFF_LESEND`)

Muss man nach dem Erstellen der Plananpassung den existierenden Plan an einer Stelle modifizieren, die nicht von der Plananpassung betroffen ist, so übernimmt das System automatisch diese Modifikationen in den neuen Plan. Ein über die Plananpassung erstellter Plan kann durch eine weitere Plananpassung diesen erneut ändern.

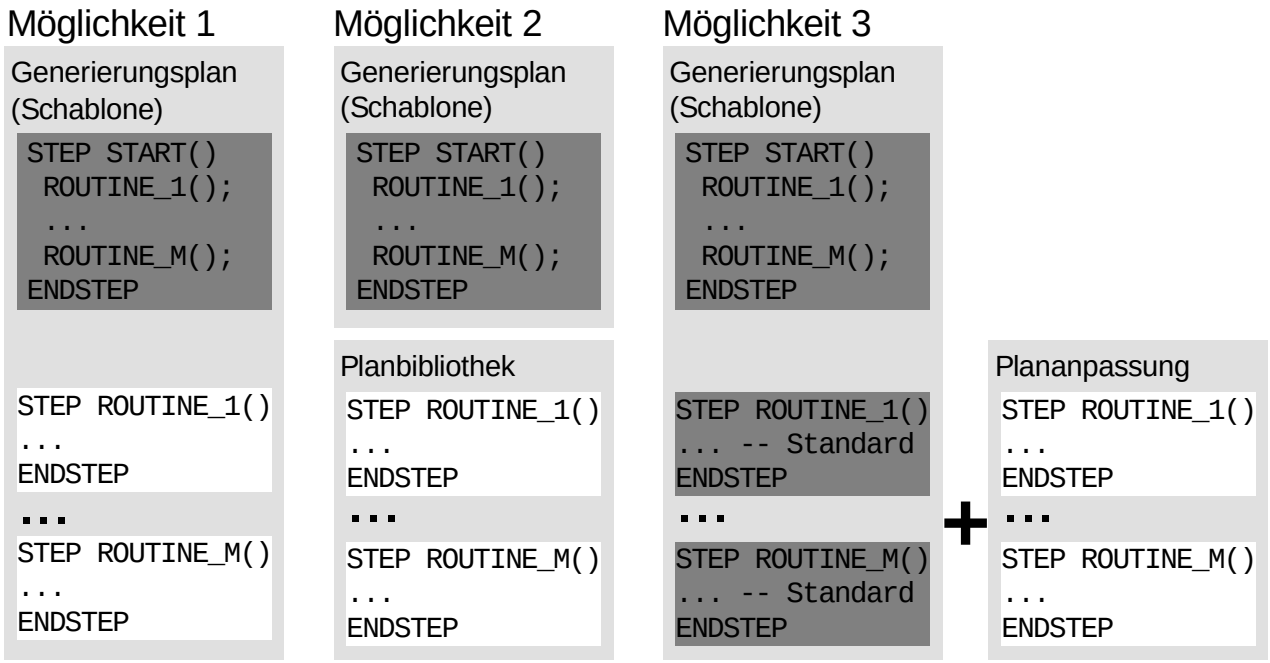


Bild 5.7: Möglichkeiten zur Anbindung von applikationsspezifischen Schritten an eine Schablone

5.2 Planschablonen

Das System soll es ermöglichen, den Programmieraufwand bei variantenreichen Produkten zu reduzieren und die Lösungssuche bei Planungsproblemen für Roboter-Fertigungszellen zu automatisieren. Das bedeutet, dass das System für unterschiedliche Arten von Problemen (*Problemklassen*) einsetzbar sein muss. Für die Lösung eines konkreten Problems muss der CAR-Anwender dazu einen Generierungsplan erstellen. Damit der CAR-Anwender einen Plan schnell realisieren und sich dabei auf seine applikationsspezifischen Anforderungen konzentrieren kann, wird im Folgenden das Konzept der *Planschablonen* beschrieben. Es folgt die Darstellung je einer Schablonenausprägung für die Problemklassen ”Programmgenerierung für variantenreiche Produkte” und ”Layoutoptimierung” sowie deren Erläuterung anhand des einleitenden Beispiels (Bild 1.1, S. 2).

Definition: *Eine Schablone ist ein Generierungsplan, der einen Algorithmus für die Lösung der Probleme aus einer Problemklasse mit applikationsunabhängigen Schritten auf hohem Abstraktionsniveau beschreibt. Eine Schablone ist problembezogen, aber applikationsunabhängig. Die applikationsunabhängigen Schritte sind im Plan als Routinen implementiert.*

Für die Erstellung eines applikationsspezifischen Generierungsplans, z. B. für die Standortoptimierung der Poliermaschine aus dem einleitenden Beispiel, muss der CAR-Anwender lediglich die applikationsunabhängigen Schritte der Schablone mit applikationsspezifischen Anweisungen füllen.

5.2.1 Realisierung von Schablonen

Zur Realisierung der Schablonen gibt es die in Bild 5.7 dargestellten Möglichkeiten. Darin sind helle Routinen abhängig, dunkle Routinen unabhängig von einer Applikation. Die Möglichkeiten sind:

1. Der CAR-Anwender trägt die applikationsspezifischen Anweisungen direkt in die Schablone ein. Der Nachteil dieser Möglichkeit besteht in der Notwendigkeit des Kopierens der Schablone für jedes neue Problem. Ein weiterer Nachteil ist die mangelnde Flexibilität bei Änderungen an der Schablone, weil man dann bestehende Pläne nachträglich ändern muss.

2. Der CAR-Anwender trägt die applikationsspezifischen Anweisungen in eine Planbibliothek ein, deren Routinen die Schablone aufruft. Das Anlegen einer Kopie der Schablone entfällt bei dieser Möglichkeit. Allerdings besteht der Nachteil, dass wenn man in eine existierende Schablone den Aufruf zu einer neuen Routine hinzufügt, Änderungen an den existierenden Planbibliotheken erforderlich sind, nämlich die Implementierung der neuen Routine.

3. Der CAR-Anwender trägt die applikationsspezifischen Anweisungen in eine Plananpassung ein, die mit der Schablone durch eine Planaddition verknüpft wird. Der Vorteil dieser Möglichkeit besteht in der Flexibilität beim Hinzufügen neuer Routinen in die Schablone, weil der CAR-Anwender existierende Plananpassungen nicht ändern muss. Man erreicht diese Flexibilität, indem die neue Routine in der Schablone eine Standardimplementierung erhält, sodass die existierenden Plananpassungen weiterhin lauffähig sind.

Ein weiterer Vorteil der dritten Möglichkeit besteht in der Möglichkeit für eine *Schrittzerlegung* in einer existierenden Schablone, d. h. ein CAR-Anwender kann nachträglich eine Routine i einer Schablone in mehrere Teilroutinen zerlegen. Jede Teilroutine übernimmt dabei eine Teilfunktion der Funktion der ursprünglichen Routine. Die Routine `ZUGRIFF_SIMULATIONSMODELL` (Bild 4.2, S. 39) ist z. B. zerlegbar in eine Routine zum Auslesen von Informationen aus dem Simulationsmodell und eine Routine zum Verändern des Simulationsmodells. In Bild 5.8 ist eine solche Schrittzerlegung dargestellt. Darin wurden die verwendeten Namen aus Platzgründen abgekürzt.

Durch das Konzept der Planaddition ist eine ursprüngliche Plananpassung (erstellt auf Basis der Schablone ohne Zerlegung) auch mit der neuen Schablone mit Zerlegung lauffähig, weil die ursprüngliche Plananpassung die zerlegte Routine i vollständig ersetzt (Bild 5.8). Eine neue Plananpassung, die der CAR-Anwender mit der neuen Schablone erstellt, kann entweder die Teilroutinen i.1 und i.2 oder die Gesamtroutine i ersetzen. Die Schrittzerlegung ermöglicht es, einzelne Schritte des auf hohem Abstraktionsniveau beschriebenen Algorithmus der Schablone zu zerlegen und damit zu präzisieren.

Im Folgenden wird zur Realisierung von Schablonen die dritte Möglichkeit (Bild 5.7) verwendet aus denen in diesem Abschnitt genannten Vorteilen.

5.2.2 Schablone zur Programmgenerierung für variantenreiche Produkte

Im Folgenden wird eine Schablone zur Erstellung von Plänen zur Programmgenerierung für variantenreiche Produkte vorgestellt. Diese Schablone entstand aufgrund von Erfahrungen aus Planrealisierungen für unterschiedliche Applikationen. Diese Schablone kann man auch für die Erstellung eines Plans verwenden, um Programme für das einleitende Beispiel zu erzeugen (Bild 1.1, S. 2).

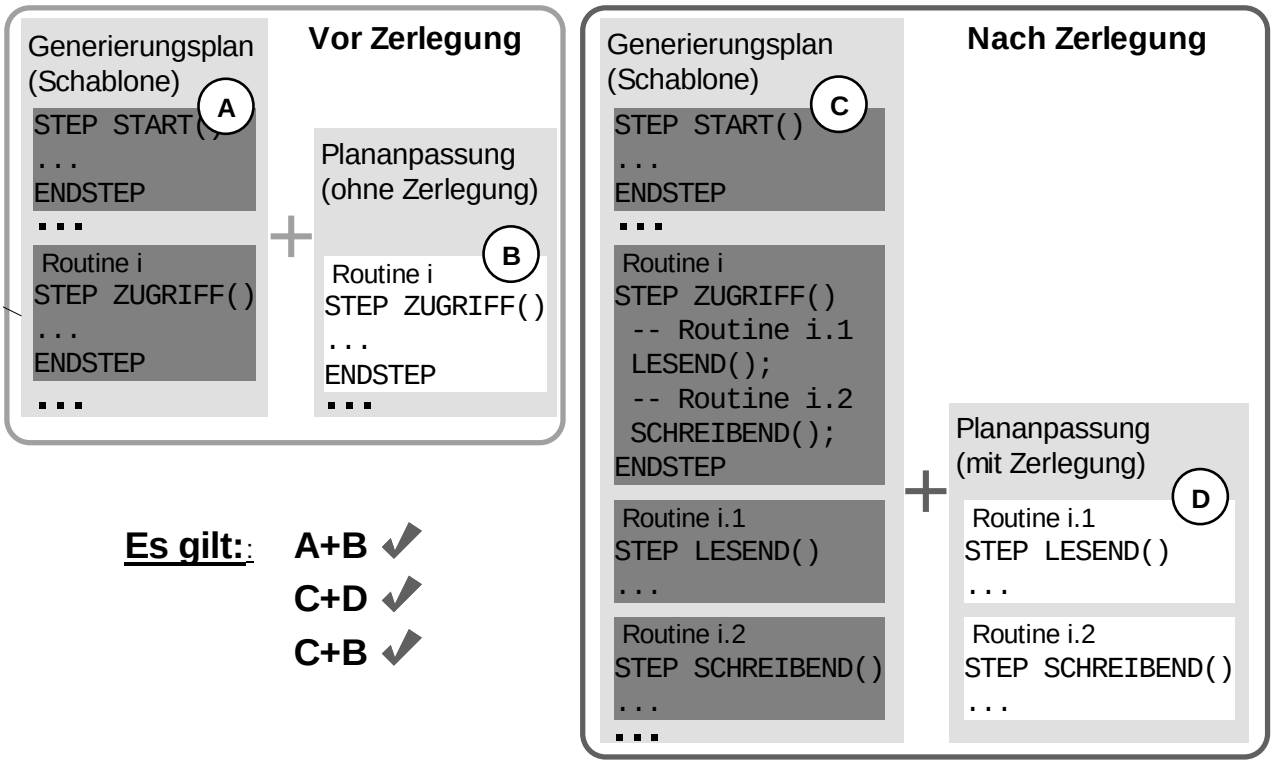


Bild 5.8: Schrittzerlegung einer allgemeinen Schablone

Die vollständige Implementierung der Schablone ist in Anhang C.1.1 abgedruckt. Der Startschritt der Schablone (Bild 5.9) enthält den applikationsunabhängigen Ablauf für die Programmgenerierung. Für die darin hell markierten Schritte muss der CAR-Anwender applikationsspezifische Anweisungen in einer Plananpassung ergänzen. Für die dunkel markierten Schritte stellt die Schablone eine Standardimplementierung zur Verfügung, die aber auch anpassbar ist.

In Bild 5.10 sind alle Teilschritte in der Reihenfolge ihres Aufrufs dargestellt. Diese Darstellung dient dem CAR-Anwender als Anleitung für die Realisierung seines Plans. Benötigt er für einen Schritt eine andere Zerlegung als es die Schablone vorschlägt, so kann er den Schritt mit seinen Teilschritten, wie im vorherigen Abschnitt (Bild 5.8) beschrieben, ersetzen.

Der Startschritt ruft weitere Schritte auf, die anhand des einleitenden Beispiels (Bild 1.1, S. 2) erläutert werden. Die Schritte mit Standardimplementierung sind gekennzeichnet. Für die Übrigen muss der CAR-Anwender das applikationsspezifische Verhalten in eine Plananpassung eintragen.

1. `INITIALISIERUNG`: Dieser Schritt erfordert die Festlegung der steuerungsspezifischen Syntax für die zu generierenden Roboterprogramme (Setzen der Systemvariablen `$LANGUAGE`).

2. `PARAMETERANFORDERUNG`: Dieser Schritt fordert die Parameter an, die ein Anwender über den Eingabeassistenten eingeben soll. `$PARAMETER(=>P)` fordert einen Parameter P an.

3. `ZUGRIFF_SIMULATIONSMODELL`: Dieser Schritt liest Informationen aus dem Simulationsmodell und verändert gegebenenfalls das Simulationsmodell.

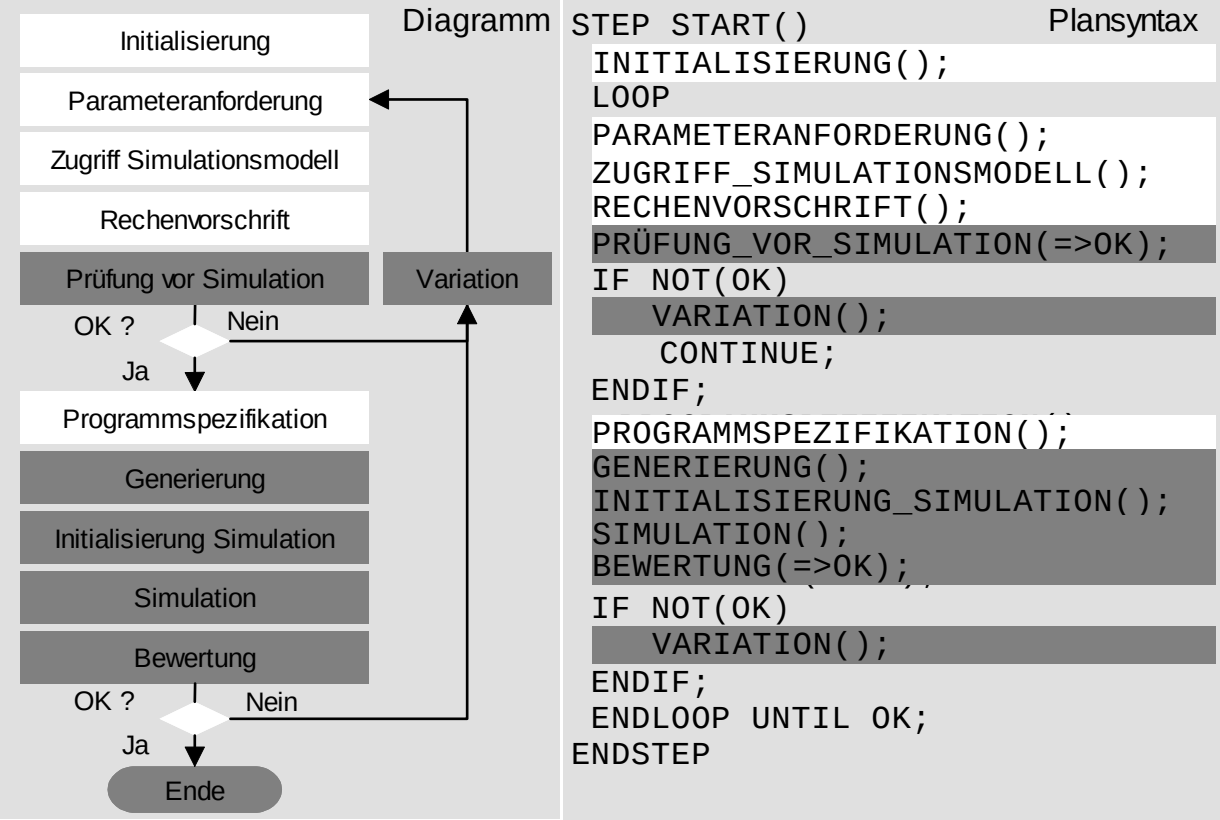


Bild 5.9: Startschritt der Schablone zur Programmgenerierung für variantenreiche Produkte

4. `RECHENVORSCHRIFT`: Dieser Schritt berechnet die Stützpunkte der Roboterbahnen (Positionen) und speichert diese in der planlokalen Variablen `BAHN[][]` (S. 56).

5. `PRÜFUNG_VOR_SIMULATION` (Standard): Dieser Schritt prüft alle Positionen der Variablen `BAHN[][]` darauf, ob der Roboter sie kollisionsfrei erreichen kann. Des Weiteren erkennt der Schritt vorhandene Kollisionen im Simulationsmodell.

6. `PROGRAMMSPEZIFIKATION`: Dieser Schritt erfordert die Angabe der Roboterbefehle, die in dem zu generierenden Roboterprogramm auftauchen sollen.

7. `GENERIERUNG` (Standard): Dieser Schritt erzeugt das steuerungsspezifische Roboterprogramm.

8. `INITIALISIERUNG_SIMULATION` (Standard): Dieser Schritt bringt den Roboter in eine Startstellung, von der aus der Roboter den Simulationslauf beginnt. Die Standardimplementierung setzt den Roboter auf die erste Position der ersten Bahn (`BAHN[1][1]`).

9. `SIMULATION` (Standard): In der Standardimplementierung führt dieser Schritt das generierte Roboterprogramm in einem Simulationslauf aus. Der Schritt ist beendet, wenn entweder der Simulationslauf erfolgreich war oder er durch einen Fehler (z. B. Kollision) abgebrochen wurde. Das Ergebnis steht in der booleschen Systemvariablen `$SIMULATION_OK`.

10. `BEWERTUNG` (Standard): Dieser Schritt bewertet den vorangegangenen Simulationslauf. Die Standardimplementierung liefert das Ergebnis der Systemvariablen `$SIMULATION_OK`.

```
1  INITIALISIERUNG();
2  PARAMETERANFORDERUNG();
3  ZUGRIFF_SIMULATIONSMODELL();
   3.1   ZUGRIFF_SIMULATIONSMODELL_LESEND();
      3.1.1  LESEN_WERKSTUECK_GEOMETRIE();
      3.1.2  LESEN_ROBOTER_INFORMATIONEN();
      3.1.3  LESEN_WEITERE_INFORMATIONEN();
   3.2   ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND();
4  RECHENVORSCHRIFT();
   4.1   BERECHNUNG_STÜTZPUNKTE();
      4.1.1  BERECHNUNG_SCHNITTGEOMETRIE();
      4.1.2  SCHNITTBERECHNUNG();
      4.1.3  SORTIERUNG STÜTZPUNKTE();
      4.1.4  NACHBEARBEITUNG();
   4.2   ZUORDNUNG_ORIENTIERUNG();
   4.3   REDUKTION_BAHNPOSITIONEN();
   4.4   ZUORDNUNG_BEZUG();
   4.5   BERECHNUNG_AN_UND_ABFAHRWEGE();
   4.6   UMRECHNUNG_EXTERNES_WERKZEUG();
   4.7   ZUORDNUNG_KONFIGURATION_UND_TURN();
5  PRÜFUNG_VOR_SIMULATION();
6  PROGRAMMSPEZIFIKATION();
7  GENERIERUNG();
8  INITIALISIERUNG_SIMULATION();
9  SIMULATION();
10 BEWERTUNG();
11 VARIATION();
```

Bild 5.10: Zerlegung der Schablone zur Programmgenerierung für variantenreiche Produkte

11. `VARIATION` (Standard): Dieser Schritt dient dazu, das Simulationsmodell zu ändern. Die Standardimplementierung enthält keine Anweisungen.

Zur Erstellung eines Generierungsplans für das einleitende Beispiel (Bild 1.1, S. 2) muss der CAR-Anwender eine Plananpassung implementieren. Eine vollständige Implementierung ist in Anhang C.1.2 (S. 177ff) abgedruckt, die im Folgenden kurz erläutert wird. Darin setzt der Schritt `INITIALISIERUNG` (S. 177) die Systemvariable `$LANGUAGE` auf den Wert für die Sprache Mitsubishi MELFA BASIC IV. Der Schritt `PARAMETERANFORDERUNG` (S. 177) fordert beim Eingabeassistenten folgende Werte an: Namen der Produktvariante (`VARIANTE`), Anstellwinkel (`WINKEL`) und Bahngeschwindigkeit (`GESCHWINDIGKEIT`). Der Anstellwinkel gibt an, wieweit der Roboter das Produkt an der Polierscheibe kippen soll (Bild 5.11b). Die Deklarationen der zugehörigen Parametervariablen trägt man als planlokale Variable in die Plananpassung ein. Die Schritte `PARAMETERANFORDERUNG`, `RECHENVORSCHRIFT` und `PROGRAMMSPEZIFIKATION` werden in den folgenden Abschnitten genauer erläutert. Für die Schritte `PRÜFUNG_VOR_SIMULATION`, `INITIALISIERUNG_SIMULATION`, `SIMULATION`, `BEWERTUNG` und `VARIATION` genügt die Standardimplementierung. Möchte der CAR-Anwender während der Planausführung mehrere Roboterprogramme erzeugen, weil er z. B. unterschiedliche Roboter ausprobieren möchte oder weil er Programme für mehrere Produktvarianten mit einer Planausführung erzeugen will, so muss der Schritt `VARIATION` entsprechende Anweisungen, z. B. für den Austausch des Roboters oder des Produkts (`$EXCHANGE`), enthalten.

Zugriff auf das Simulationsmodell

Der Schritt ZUGRIFF_SIMULATIONSMODELL ist in der Schablone in einen lesenden und einen schreibenden Teilschritt zerlegt. Der Teilschritt ZUGRIFF_SIMULATIONSMODELL_LESEND ist wiederum unterteilt (Bild 5.10).

3.1 ZUGRIFF_SIMULATIONSMODELL_LESEND: Dieser Schritt liest alle Informationen aus dem Simulationsmodell, die man für die weiteren Berechnungen benötigt.

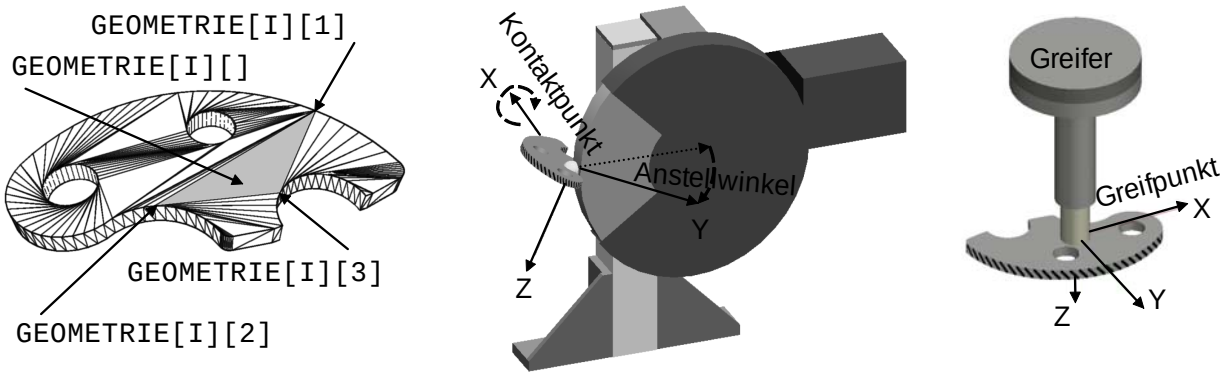
3.1.1 LESEN_WERKSTÜCK_GEOMETRIE: Dieser Schritt liest die Geometriedaten des zu bearbeitenden Werkstücks und speichert einfache geometrische Körper in Variablen, die einen vordefinierten Geometrietyp besitzen, und Polyeder in zweidimensionalen Feldern, z. B. GEOMETRIE[][] (S. 57).

3.1.2 LESEN_ROBOTER_INFORMATIONEN: Dieser Schritt liest Informationen über den Roboter aus dem Simulationsmodell (z. B. Tool Center Point, digitale/analoge Ein- und Ausgänge, die maximale Bahngeschwindigkeit).

3.1.3 LESEN_WEITERE_INFORMATIONEN: Dieser Schritt liest bisher nicht ausgelesene Informationen aus (z. B. Objektpositionen, vorhandene Greif- und Greiferpunkte).

3.2 ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND: Dieser Schritt trägt fehlende Informationen in das Simulationsmodell ein (z. B. nicht vorhandene Greif- oder Greiferpunkte).

Für die einleitende Polierapplikation (Bild 1.1, S. 2) muss man in der zugehörigen Plananpassung (Anhang C.1.2, S. 177ff) im Schritt LESEN_WERKSTÜCK_GEOMETRIE (S. 177) die Geometriedaten der Produktvariante auslesen, um daraus die Roboterbahn entlang des Randes zu berechnen. Die Werkstückgeometriedaten erhält man durch den Aufruf des Systemschritts $GET_GEOMETRY mit dem Produktnamen (VARIANTE) als Eingabeparameter und der Variablen GEOMETRIE[][] als Ausgangsparameter (Bild 5.11a). Im Schritt LESEN_ROBOTER_INFORMATIONEN (S. 177) liest man die Nummer des digitalen Ausgangs aus, mit dem die Poliermaschine verbunden ist. Dies wird über die Anweisung $GET_DO und dem Namen der Poliermaschine (MASCHINE) als Eingabeparameter und mit der Variablen AN von Typ $NUMBER als Ausgangsparameter erreicht. Der Schritt LESEN_WEITERE_INFORMATIONEN (S. 177) liest einen eventuell am Produkt vorhandenen Greifpunkt mit der Anweisung $GET_GRIPPOINT aus (Bild 5.11c). Der Greifpunkt gibt an, wo der Greifer das Produkt greifen soll. Da CAD-Daten noch keinen Greifpunkt enthalten, muss der CAR-Anwender diesen ergänzen. Diesen Greifpunkt kann er manuell vor der Planausführung im Simulationsmodell ergänzen und im Schritt LESEN_WEITERE_INFORMATIONEN auslesen. Es ist aber auch denkbar einen Standardgreifpunkt (z. B. Schwerpunkt) im Schritt ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND zu erzeugen. Um die Kippung des Produkts um den Anstellwinkel zu berücksichtigen, wird im Schritt ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND (S. 177) der Kontaktpunkt um den Anstellwinkel gedreht (Bild 5.11b). Dies bewirkt der Systemaufruf $ROTATE mit dem Eingangsparametern Anstellwinkel (WINKEL) und dem Namen des zu drehenden Objekts (<Kontaktpunkt>). Des Weiteren wird das Produkt mit dem Greifer verbunden ($GRIP).



(a) Lesen der Geometriedaten **(b)** Drehen des Kontaktpunktes **(c)** Greifpunkt

Bild 5.11: Zugriff auf das Simulationsmodell

Rechenvorschrift

Der Schritt `RECHENVORSCHRIFT` mit folgenden Teilschritten berechnet die Stützpunkte der Bahnen:

4.1 `BERECHNUNG_STÜTZPUNKTE`: Dieser Schritt berechnet die X-, Y- und Z-Koordinatenwerte der Bahnstützpunkte und trägt diese in das Feld `BAHN[][]` ein (S. 56).

4.1.1 `BERECHNUNG_SCHNITTGEOMETRIE`: Dieser Schritt berechnet einfache geometrische Elemente (z. B. Linien, Ebenen), die zum Schneiden mit der Werkstückgeometrie dienen.

4.1.2 `SCHNITTBERECHNUNG`: Dieser Schritt schneidet die geometrischen Elemente mit der Werkstückgeometrie. Das Ergebnis sind Folgen von (unsortierten) Schnittpunkten.

4.1.3 `SORTIERUNG_STÜTZPUNKTE` (Standard); Der Schritt sortiert die Schnittpunkte in die spätere Abfahrreihenfolge für den Roboter. Die Standardimplementierung sortiert nicht.

4.1.4 `NACHBEARBEITUNG` (Standard): Dieser Schritt kann weitere Berechnungen an den Stützpunkten durchführen. Die Standardimplementierung enthält keine Anweisungen.

4.2 `ZUORDNUNG_ORIENTIERUNG`: Dieser Schritt weist jedem Stützpunkt eine Orientierung zu.

4.3 `REDUKTION_BAHNPOSITION` (Standard): Dieser Schritt reduziert die Zahl der Stützpunkte. Die Standardimplementierung enthält keine Reduktion.

4.4 `ZUORDNUNG_BEZUG`: Dieser Schritt ordnet jedem Stützpunkt ein Bezugsobjekt zu, auf dessen Koordinatensystem sich der Stützpunkt bezieht. In der Regel trägt man das Werkstück ein.

4.5 `BERECHNUNG_AN_UND_ABFAHRWEGE` (Standard): Dieser Schritt berechnet Anfahr- und Abfahrbewegungen für die Bahnen. Die Standardimplementierung berechnet keine Bahnen.

4.6 `UMRECHNUNG_EXTERNES_WERKZEUG` (Standard): Falls der Roboter das Werkstück führt, rechnet dieser Schritt die Bahnstützpunkte auf das externe Werkzeug um. Die Standardimplementierung nimmt an, dass der Roboter das Werkzeug führt und enthält somit keine Anweisungen.

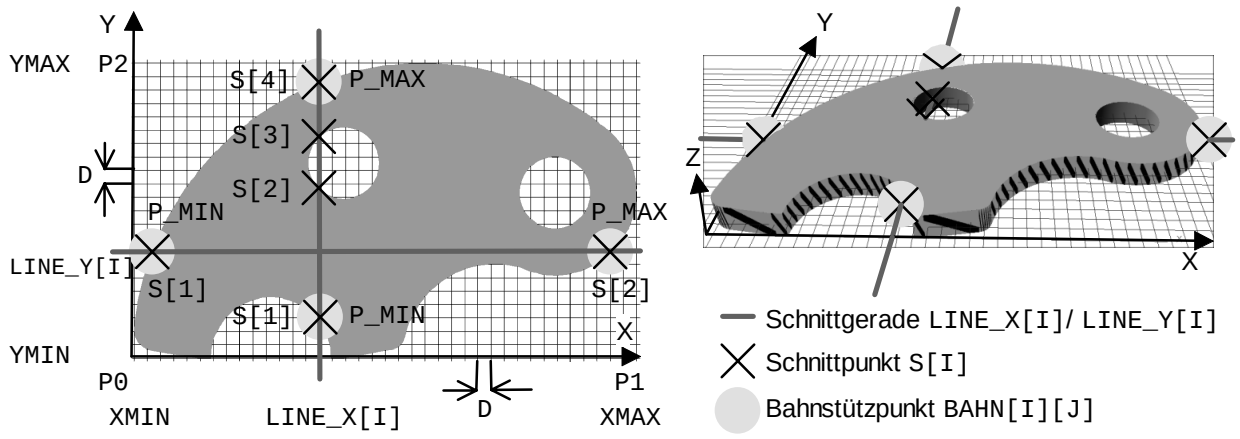


Bild 5.12: Berechnung der Bahnstützpunkte

4.7 `ZUORDNUNG_KONFIGURATION_UND_TURN`: Dieser Schritt ordnet jedem Stützpunkt eine Konfiguration und Turns zu, um Mehrdeutigkeiten für die Roboterstellung zu beseitigen.

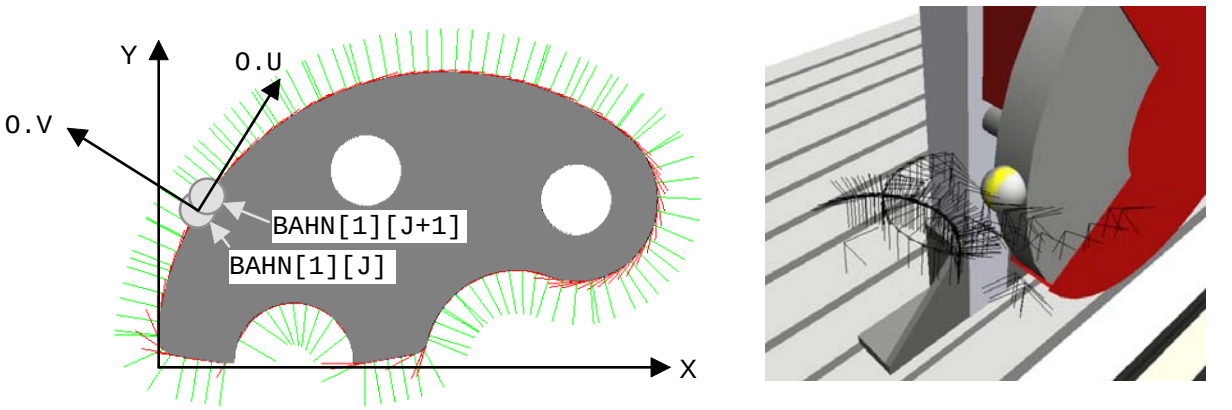
Die Idee, die hinter der Zerlegung des Schritts `BERECHNUNG_STÜTZPUNKTE` steckt, ist ein Schneiden der Werkstückgeometrien mit Linien und Flächen. Sollte der CAR-Anwender diese Vorgehensweise nicht verwenden wollen, weil er z. B. direkt Punkte aus den Geometriedaten verwenden will, so kann er in einer Plananpassung den vollständigen Schritt `BERECHNUNG_STÜTZPUNKTE` gemäß seinen Vorstellungen ersetzen (Bild 5.7).

Eine Möglichkeit zur Berechnung der Roboterbahnen für den Poliervorgang (Bild 1.1, S. 2) wird im Folgenden dargestellt. Die Implementierung ist in Anhang C.1.2 (S. 177ff) abgedruckt. Die notwendigen Berechnungen werden den oben aufgeführten Schritten der Schablone zugeordnet. Der Schritt `BERECHNUNG_STÜTZPUNKTE` schneidet die Produktvarianten mit Geraden, die parallel zur X-Achse (`LINE_X[]`) oder zur Y-Achse (`LINE_Y[]`) des Produktkoordinatensystems verlaufen (Bild 5.12).

Der Teilschritt `BERECHNUNG_SCHNITTGEOMETRIE` (S. 177) berechnet dazu äquidistante Schnittgeraden im Abstand `D`, die parallel zur XY-Ebene des Produktkoordinatensystems verlaufen, und speichert diese in den Variablen `LINE_X[]` und `LINE_Y[]`. Der Schritt bestimmt zunächst die maximalen und die minimalen X-, Y- und Z-Koordinatenwerte (`XMIN`, `YMIN`, `ZMIN`, `XMAX`, `YMAX`, `ZMAX`) der Produktgeometriedaten. Anschließend folgt die Bestimmung des Z-Werts `Z` aller Geraden und die Berechnung der Anzahl der Geraden `NX` und `NY`. Des Weiteren sind Hilfspunkte `P0`, `P1`, `P2` auf den Achsen für die Konstruktion der Geraden erforderlich, die zur Berechnung der Geraden dienen.

Der Schritt `SCHNITTBERECHNUNG` (S. 177) schneidet jede Gerade mit der Produktgeometrie (`GEOMETRIE[][]`). Die entstehenden Schnittpunkte schreibt er in das Feld `S[]`. Für die Bahnstützpunkte sind nur die beiden Schnittpunkte interessant, die den größten und kleinsten Wert (`P_MIN`, `P_MAX`) bezüglich der X-Koordinate für die Geraden `LINE_Y[]` und bezüglich der Y-Koordinate für die Geraden `LINE_X[]` besitzen. Diese beiden Punkte werden als Stützpunkte im Array `BAHN[][]` aufgenommen (Bild 5.12). Das Ergebnis ist eine Bahn (`BAHN[1][]`) mit unsortierten Stützpunkten.

Der Schritt `SORTIERUNG_STÜTZPUNKTE` (S. 178) sortiert die Reihenfolge der Bahn `BAHN[1][]`, indem er beginnend beim Stützpunkt `BAHN[1][1]` immer den nächstgelegenen Punkt als nächsten



(a) Orientierungsberechnung **(b)** Umrechnung externes Werkzeug

Bild 5.13: Zuordnung der Orientierung und Bahnverlauf für externes Werkzeug

Bahnstützpunkt verwendet ($SORT_DISTANCE). Der Schritt ZUORDNUNG_ORIENTIERUNG (S. 178) berechnet für jeden Bahnpunkt BAHN[1][J] eine Orientierung O (Bild 5.13a), indem der er den Richtungsvektor O.U zum folgenden Stützpunkt BAHN[1][J+1]zeigen lässt. Der Richtungsvektor O.W zeigt in Richtung der Z-Achse des Produktkoordinatensystems und der Richtungsvektor O.V bildet mit den anderen beiden Richtungsvektoren ein Rechtssystem.

Der Schritt REDUKTION_BAHNPOSITION enthält bei dieser Applikation keine Anweisungen, weil bereits über den Abstand D die Anzahl der Bahnstützpunkte beeinflussbar ist. Der Schritt ZUORDNUNG_BEZUG ordnet jedem Bahnstützpunkt die Produktvariante als Bezugsobjekt zu, weil alle Berechnungen bezüglich des Produktkoordinatensystems durchgeführt wurden. Der Schritt BERECHNUNG_AN_UND_ABFAHRWEGE (S. 178) fügt vor den ersten bzw. hinter den letzten Stützpunkt einer Bahn jeweils einen Stützpunkt ein, der einen Abstand ANFAHRDISTANZ entlang der V-Achse des Stützpunkts vom ersten bzw. vom letzten Stützpunkt einer Bahn hat.

Im Schritt UMRECHNUNG_EXTERNES_WERKZEUG (S. 178) findet die Umrechnung der Bahn statt, sodass der Roboter das Produkt entlang des Kontaktpunktes führt (Bild 5.13b). Dazu überschreibt er das Bezugsobjekt von ”Produktvariante” auf ”Kontaktpunkt” und rechnet die Stützpunkte um. Der Schritt ZUORDNUNG_KONFIGURATION_UND_TURN (S. 178) weist jedem Stützpunkt eine Standardkonfiguration (Right, Above, No Flip) und Standardturns (alle Null) zu.

Programmspezifikation

Für die Polierapplikation (Bild 1.1, S. 2) enthält der Schritt PROGRAMMSPEZIFIKATION (S. 179) Anweisungen zur Erzeugung eines Roboterprogramms mit einer Hauptroutine. Die darin enthaltenen Anweisungen schalten zunächst die Poliermaschine ein ($SET_DIG_OUTPUT), setzen anschließend die angegebene Bahngeschwindigkeit ($PATH_SPEED), führen die im Feld BAHN[][] gespeicherte Bahn aus ($PATH) und schalten schließlich die Poliermaschine ab ($RESET_DIG_OUTPUT).

5.2.3 Schablone zur Layoutoptimierung

Diese Schablone zur Layoutoptimierung testet mehrere Zellenlayoutkandidaten. Falls ein Kandidat Erfolg verspricht, wird dieser gemäß einer vorgegebenen Zielfunktion F optimiert. Als Standardzielfunktion enthält die Schablone die Ausführungszeit des erzeugten Roboterprogramms. Ein Zellenlayout ist definiert durch die anfängliche Zellenanordnung vor Beginn der Optimierung und einer Zellenänderung $\overrightarrow{x} = (x_1, ..., x_n)^T \in \mathbb{R}^n$, d. h. Verschiebung und Drehung einzelner Objekte. Dabei entspricht jede Vektorkomponente x_i der Änderung eines Freiheitsgrads eines Objekts in der Zelle, d. h. x_i entspricht einer Verschiebung entlang einer Achse des jeweiligen Objektkoordinatensystems oder einer Drehung um eine dieser Achsen (Roll, Pitch, Yaw). Die zulässigen Zellenänderungen sind beschränkt durch $x_i \in \left[x_i^{\min}, x_i^{\max}\right]$.

Die Schablone legt die Zellenlayoutkandidaten und damit die zu testenden Zellenänderungen durch eine Diskretisierung aller zulässigen Zellenänderungen $\left[x_i^{\min}, x_i^{\max}\right]$ in $n_i > 1$ Werte mit äquidistanten Abständen Δx_i fest.

$$\Delta x_i = \frac{(x_i^{\max} - x_i^{\min})}{(n_i - 1)} \tag{5.1}$$

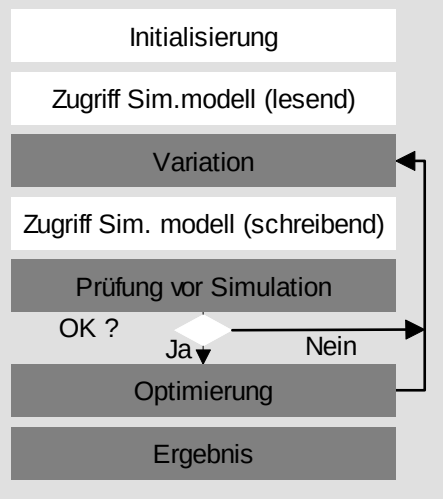
Insgesamt existieren demnach m zu testende Zellenlayoutkandidaten:

$$m = \prod_{i=1}^{n} (n_i) \tag{5.2}$$

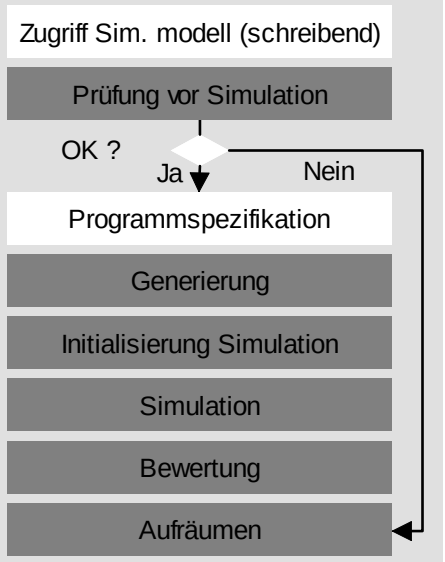
Um die Zellenlayoutkandidaten nacheinander zu testen werden diese wie folgt geordnet. Dadurch ergibt sich eine beschränkte Folge X der zu testenden Zellenlayoutkandidaten mit $n_0 = 1$:

$$X = \left\{ \vec{x}_k | \vec{x_k} = (x_{k1}, ..., x_{kn})^T \right\}_{k=1}^{m} \text{ mit } x_{ki} = x_i^{\min} + n_{ki}\Delta x_i \text{ und } n_{ki} = \left\lfloor \frac{(k-1)}{\prod_{j=0}^{i-1} n_j} \right\rfloor \bmod n_i \tag{5.3}$$

Ein Zellenlayoutkandidat $\vec{x}_k$ wird gemäß der in der Schablone angegebenen Zielfunktion F optimiert, wenn der Roboter für dieses Layout alle anzufahrenden Stellungen kollisionsfrei erreichen kann. Die Optimierung erfolgt durch Bestimmung eines lokalen Minimums $F(\overrightarrow{x_k}^*)$ an der Stelle $\overrightarrow{x_k}^*$ für jeden einzelnen Zellenlayoutkandidaten $\vec{x}_k$ aus der Folge X. Für die Bestimmung der Funktionswerte von F (z. B. Ausführungszeit) ist die Ausführung eines steuerungsspezifischen Roboterprogramms in einem Simulationslauf erforderlich.




```
STEP START()
 INITIALISIERUNG();
 ZUGRIFF_SIMULATIONSMODELL_LESEND();
 WHILE $VARIATION(=>ÄNDERUNG[K][]) LOOP
 ZUGRIFF_SIM.MODELL_SCHREIBEND(ÄNDERUNG[K][]);
 PRÜFUNG_VOR_SIMULATION(=>OK);
  IF OK
    $OPTIMIZATION("ZIELFUNKTION",ÄNDERUNG[K][]
     => LOKALES_MINIMUM[K], ÄNDERUNG[K][]);
  ENDIF;
  K = K + 1;
 ENDLOOP;
 ERGEBNIS();
ENDSTEP
```




```
STEP ZIELFUNKTION($NUMBER[] PARAM => $NUMBER FOPT,
                  $BOOL LIMIT=TRUE)
 ZUGRIFF_SIM.MODELL_SCHREIBEND(PARAM[]);
 PRÜFUNG_VOR_SIMULATION(=>OK);
 IF OK
    PROGRAMMSPEZIFIKATION();
    GENERIERUNG();
    INITIALISIERUNG_SIMULATION();
    SIMULATION();
    BEWERTUNG(=> LIMIT, FOPT);
 ENDIF;
 AUFRÄUMEN();
ENDSTEP
```

Bild 5.14: Startschritt und Zielfunktionsschritt der Schablone für die Layoutoptimierung

Schablonenaufbau

Die Implementierung der Schablone ist in Anhang C.2.1 (S. 179ff) abgedruckt. Die wichtigen Schritte der Schablone sind in Bild 5.14) dargestellt: die Schritte `START` und `ZIELFUNKTION` zur Berechnung des Zielfunktionswerts. Hell markierte Schritte müssen vom CAR-Anwender in einer Plananpassung ausgefüllt werden. Die dunkel markierten Schritten enthalten ein Standardverhalten, das entweder änderbar, gemäß Abschnitt 5.2.1 (z. B. `BEWERTUNG`), oder konfigurierbar ist (z. B. `$OPTIMIZATION`). Die Folge X wird im Feld `ÄNDERUNG[][]` gespeichert, `ÄNDERUNG[K][]` entspricht somit einem Zellenlayoutkandidaten $\vec{x}_k$und die Vektorkomponente x_{ki} dem Feldelement `ÄNDERUNG[K][I]`. Die Schrittaufrufe aus dem Startschritt mit allen Teilschritten sind in Bild 5.15 dargestellt.

Für die Erstellung eines Generierungsplans schlägt die Schablone folgende Struktur vor:

1. `INITIALISIERUNG`: Der CAR-Anwender legt in diesem Schritt fest, welche Objekte bei der Layoutoptimierung verschoben oder gedreht werden. Für jeden Objektfreiheitsgrad initialisiert

```
1 INITIALISIERUNG();
2 ZUGRIFF_SIMULATIONSMODELL_LESEND();
  2.1 LESEN_ROBOTER_POSITIONEN();
  2.2 LESEN_WEITERE_INFORMATIONEN();
3 $VARIATION(...)
4 ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND();
  4.1 VARIATION_ZU_OBJEKTEN();
  4.2 OBJEKTE_ANORDNEN();
5 PRUEFUNG_VOR_SIMULATION();
6 $OPTIMIZATION(...)
7 ERGEBNIS();
```

```
8 ZIELFUNKTION();
  8.1 ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND();
  8.2 PRUEFUNG_VOR_SIMULATION();
  8.3 PROGRAMMSPEZIFIKATION();
  8.4 GENERIERUNG();
  8.5 INITIALISIERUNG_SIMULATION();
  8.6 SIMULATION();
  8.7 BEWERTUNG();
  8.8 AUFRÄUMEN();
```

Bild 5.15: Schrittzerlegung der Schablone für die Layoutoptimierung

der Schritt die Werte $x_i^{\min}$ (`MIN[]`), $x_i^{\max}$ (`MAX[]`) und n_i (`AUFLÖSUNG[]`). Zusätzlich werden hier das zu verwendende Optimierungsverfahren (`$STRATEGY`) und die Sprache der zu erzeugenden Programme (`$LANGUAGE`) eingestellt. Der CAR-Anwender kann hier weitere Initialisierungen vornehmen. Dazu zählen eine Beschränkung der Zahl der Zielfunktionsberechnungen (`$MAX_ITERATION`) oder die Angaben von erlaubten Kollisionen (`$COLLISION_EXCEPTION`). Zusätzlich fügt das System weitere Anweisungen über `=> PREVIOUS STEP` hinzu aus der Schablone hinzu (S. 180). Dazu zählt das Speichern der anfänglichen Positionen der Objekte (`INITIALER_OBJEKT_STANDORT[]`) und des Zustands des Simulationsmodells vor der Optimierung (`$STORE_TO_TIDY`). Auch wird durch den Aufruf von `$INIT_VARIATION` mit den Parametern `MIN[]`, `MAX[]` und `AUFLÖSUNG[]` der in der Schablone später auftretende Schritt `$VARIATION` initialisiert.

2. `ZUGRIFF_SIMULATIONSMODELL_LESEND`: Dieser Schritt liest die benötigten Informationen aus dem Simulationsmodell.

   2.1 `LESEN_ROBOTER_POSITIONEN`: Dieser Schritt liest die benötigten Roboterpositionen aus vorhandenen Positionslisten und schreibt sie in die Variable `BAHN[][]` (S. 56). Die Listen müssen Bezugsobjekte für die Positionen besitzen, damit klar ist, ob eine Position bei einer Objektverschiebung ebenfalls verschoben werden muss.

   2.2 `LESEN_WEITERE_INFORMATIONEN`: In diesem Schritt werden weitere Informationen aus dem Simulationsmodell ausgelesen, z. B. digitale/analoge Ein-/Ausgangsbelegungen.

3. `$VARIATION` (Standard): Dieser Schritt erzeugt für jeden Aufruf einen neuen Layoutkandidaten $\vec{x}_k$ aus der Folge X nach Formel 5.3 und speichert dessen Zellenänderung in das Feld `ÄNDERUNG[][]`. Der Rückgabewert des Schritts ist so lange wahr, bis alle Zellenlayoutkandidaten $\vec{x}_k$ der Folge X einmal erzeugt wurden. Nach m Durchläufen bricht die Schleife ab.

4. `ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND`: Dieser Schritt ordnet die Objekte im Simulationsmodell gemäß der Zellenänderung $\vec{x}_k$ an.

   4.1 `VARIATION_ZU_OBJEKTEN`: Dieser Schritt ordnet jeder Vektorkomponente x_{ki} einer Zellenänderung $\vec{x}_k$ einen Freiheitsgrad eines Objekts zu.

   4.2 `OBJEKTE_ANORDNEN` (Standard): Dieser Schritt verschiebt und dreht die Objekte im Simulationsmodell.

5. `PRÜFUNG_VOR_SIMULATION` (Standard): Dieser Schritt prüft alle Positionen der Variablen `BAHN[][]` darauf, ob der Roboter sie kollisionsfrei erreichen kann. Des Weiteren erkennt der Schritt vorhandene Kollisionen im Simulationsmodell.

6. `$OPTIMIZATION` (Standard): Dieser Schritt führt eine Parameteroptimierung unter der Verwendung des in `$STRATEGY` eingestellten Verfahrens (z. B. Hooke-Jeeves [72], Simulated-Annealing [113]) aus. Startvektor ist die Belegung von `ÄNDERUNG[K][]`. Die zu optimierende Zielfunktion, d. h. der Schritt `ZIELFUNKTION` zur Berechnung eines Zielfunktionswerts, wird als Parameter (`"ZIELFUNKTION"`) übergeben und von diesem Schritt während des Optimierungsvorgangs aufgerufen. Das gefundene lokale Minimum $F(\vec{x}_k^*)$ wird im Feldelement `LOKALES_MINIMUM[K]` gespeichert.

7. `ERGEBNIS` (Standard): Für das kleinste aller lokalen Minima (`LOKALES_MINIMUM[]`) wird das Simulationsmodell entsprechend angeordnet und ein Roboterprogramm erzeugt.

8. `ZIELFUNKTION`: Dieser Schritt berechnet den Zielfunktionswert eines Layouts.

   8.1 `ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND`: siehe 4.

   8.2 `PRÜFUNG_VOR_SIMULATION` (Standard): siehe 5.

   8.3 `PROGRAMMSPEZIFIKATION`: In diesem Schritt stehen alle Roboterbefehle, die in dem zu generierenden Roboterprogramm auftauchen sollen.

   8.4 `GENERIERUNG` (Standard): Dieser Schritt erzeugt das steuerungsspezifische Programm.

   8.5 `INITIALISIERUNG_SIMULATION` (Standard): Dieser Schritt bringt den Roboter in eine Startstellung, von der aus der Roboter den Simulationslauf beginnt. Die Standardimplementierung setzt den Roboter auf die erste Position der ersten Bahn (`BAHN[1][1]`).

   8.6 `SIMULATION` (Standard): Dieser Schritt führt das generierte Roboterprogramm in einem Simulationslauf aus. Der Schritt ist beendet, wenn entweder der Simulationslauf erfolgreich war oder er durch einen Fehler (z. B. Kollision) abgebrochen wurde. Das Ergebnis steht in der booleschen Systemvariablen `$SIMULATION_OK`.

8.7 `BEWERTUNG` (Standard): Dieser Schritt bewertet den durchgeführten Simulationslauf und liefert in `LIMIT`, ob der Bewertungswert gültig ist. Wenn ja, steht der Wert in `FOPT`. Die Standardimplementierung liefert als Bewertungswert die Ausführungszeit des Roboterprogramms.

8.8 `AUFRÄUMEN` (Standard): Dieser Schritt bringt das Simulationsmodell in den ursprünglichen Zustand.

Für die meisten Applikationen muss der CAR-Anwender nur die hell markierten Schritte aus Bild 5.15 in einer Plananpassung implementieren. Die Schritte mit Standardimplementierung muss er in der Regel nicht betrachten. In seltenen Fällen, falls die Ausführungszeit nicht das Optimierungskriterium ist, muss er den Schritt `BEWERTUNG` ändern.

Beispiel

In der Polierapplikation (Bild 1.1, S. 2) soll ein Standort für die Poliermaschine auf der Grundplatte gefunden werden, sodass der Poliervorgang für eine Produktvariante möglichst kurz wird. Zur Verdeutlichung werden nur zwei Freiheitsgrade zur Verschiebung der Poliermaschine auf der Grundplatte betrachtet (Bild 5.16). Die für dieses Beispiel notwendige Implementierung einer Plananpassung zur Erstellung eines Generierungsplan ist in Anhang C.2.2 (S. 181ff) abgedruckt.

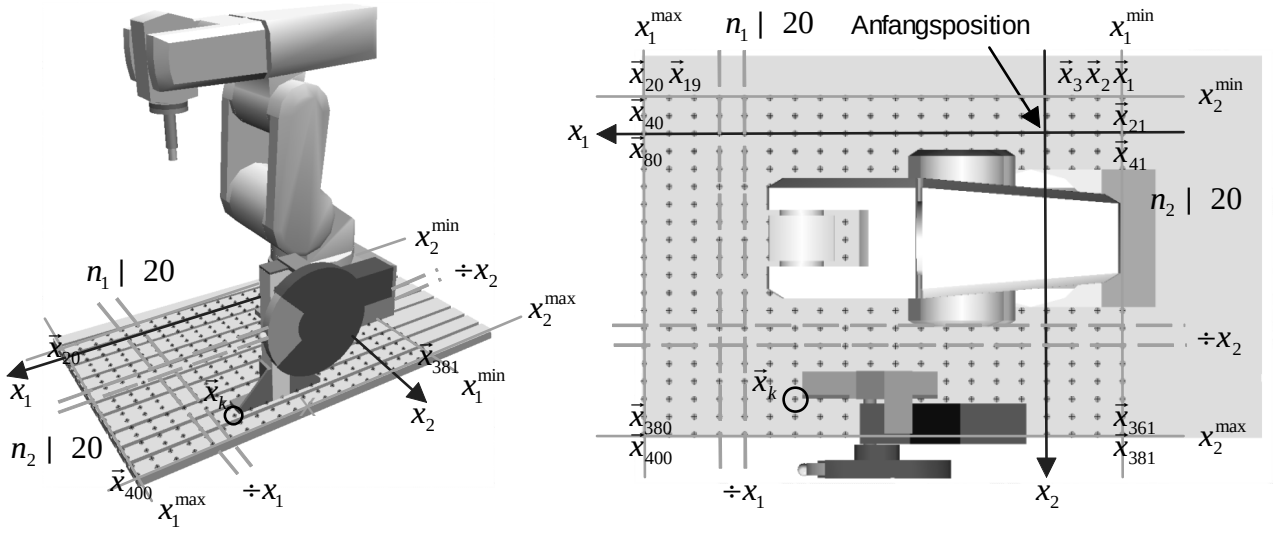


Bild 5.16: Anwendung der Schablone für die Layoutoptimierung

Dazu ist in Schritt `INITIALISIERUNG` (S. 181) zunächst die Sprache der zur erzeugenden Roboterprogramme (MELFA BASIC IV) und das zu verwendende Optimierungsverfahren (Verfahren von Hooke-Jeeves) angegeben. Des Weiteren ist das zu verschiebende Objekt, die Poliermaschine, mit den benötigten Werten $x_i^{\min}$ (`MIN[I]`), $x_i^{\max}$ (`MAX[I]`) und n_i (`AUFLÖSUNG[I]`) für alle Freiheitsgrade aufgeführt.

Der Schritt `LESEN_ROBOTER_POSITIONEN` (S. 181) liest alle benötigten Stützpunkte: eine Aufnahmeposition (`"Aufnahme"`) und eine Position oberhalb des Werkstückträgers (`"ÜberAufnahme"`).

Beide Positionen werden auch dazu verwendet das Produkt wieder auf den Träger zu legen. Für den Poliervorgang wurde eine Positionsliste (`"Polierbahn"`) mit einem Generierungsplan erzeugt, der mit der Schablone aus dem vorangegangenen Abschnitt 5.2.2) erstellt wurde.

Der Schritt `LESEN_WEITERE_INFORMATIONEN` (S. 181) liest die E/A-Belegung zur Ansteuerung der Poliermaschine und des Greifers aus dem Simulationsmodell. Der Schritt `VARIATION_ZU_OBJEKTEN` (S. 181) weist einer Zellenänderung $\vec{x}_k$, die in Form eines Feldes reeller Zahlen vorliegt, die beiden translatorischen Freiheitsgrade für die Poliermaschine zu.

Der Schritt `PROGRAMMSPEZIFIKATION` (S. 181) legt den Inhalt des zu erzeugenden Programms fest: Der Roboter nimmt das Produkt vom Träger, poliert es und legt es auf den Träger zurück. Für alle übrigen Schritte genügt die Standardimplementierung.

6 Generierungssteuerung

Dieses Kapitel beschreibt die Generierungssteuerung (Bild 4.1, S. 37), die für die Ausführung der im vorangegangenen Kapitel entworfenen Pläne und für die Bereitstellung der Basisfunktionen zuständig ist (Anforderung 2, S. 30).

Die Planverarbeitung in der Generierungssteuerung unterteilt sich in vier Phasen:

1. Aufbau der internen Repräsentation der Generierungspläne
2. Initialisierung der Planausführung
3. Ausführung
4. Abbau der internen Repräsentation der Generierungspläne

Zunächst muss die Generierungssteuerung in der ersten *Aufbauphase* die interne Repräsentation der Generierungspläne aufbauen. Sie durchläuft anschließend diese Repräsentation zwei Mal (zweite und dritte Phase). Dabei verarbeitet sie jeweils einen anderen der Teil der Repräsentation. In der *Initialisierungsphase* legt die Generierungssteuerung die globalen Systemvariablen und die planlokalen Variablen an und weist ihnen einen Standardwert zu (S. 55). In der *Ausführungsphase* führt die Generierungssteuerung die Plananweisungen aus. In der *Abbauphase* baut die Generierungssteuerung die interne Repräsentation wieder ab.

6.1 Steuergraph

Für die Ausführung der Pläne benötigt die Generierungssteuerung eine interne Repräsentation, die im Folgenden beschrieben wird.

6.1.1 Motivation

Üblicherweise übersetzt man prozeduralen Programmcode (z. B. Pascal, C) zunächst in einen Zwischencode, aus dem am der Ende der Synthese Assemblercode hervorgeht, bevor eine Zielmaschine diesen interpretiert [10]. Der konkrete Befehlssatz des Zwischencodes hängt dabei vom ausführenden Prozessor ab. Der Abstraktionsgrad eines solchen Befehlssatzes ist niedrig und enthält in der Regel Zwei- oder Dreiadressbefehle [10]. Der Vorteil eines einfachen Befehlssatzes liegt in einem einfachen Aufbau der ausführenden Zielmaschine [16]. Dem entgegen stehen der komplexe Aufbau eines entsprechenden Übersetzers und die Schwierigkeit, den Bezug zwischen den Programmzeilen in der Quellsyntax und dem erzeugten Code in der Zielsprache herzustellen.

Für die Ausführung der Pläne wird in dieser Arbeit eine interne Repräsentation auf höherem Abstraktionsniveau angestrebt, sodass sich die Sprachkonstrukte der Pläne in der Repräsentation wiederfinden. Dadurch ist zwar die ausführende Zielmaschine aufwendiger, aber die Übersetzung der Pläne in die interne Repräsentation vereinfacht sich.

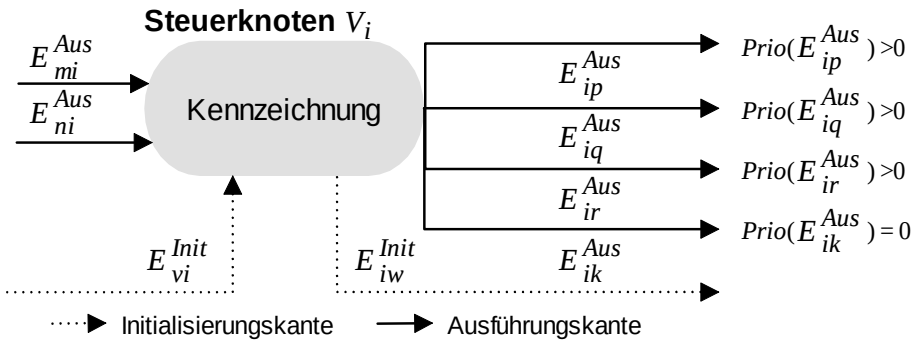


Bild 6.1: Steuerknoten

6.1.2 Definition

Für die Ausführung werden die Pläne in den so genannten *Steuergraphen* übersetzt, den die Generierungssteuerung zwei Mal durchläuft. Der Steuergraph ist ein gerichteter Graph (V, E) mit *Steuerknoten* $V = \{V_i | 1 \leq i \leq N\}$ und Kanten E. Der Steuerknoten V_i stellt die kleinste Einheit dar, den die Generierungssteuerung verarbeitet. Er enthält alle dazu benötigten Informationen.

Entsprechend den beiden Phasen, der Initialisierungsphase und der Ausführungsphase, existieren zwei unterschiedliche Typen von gerichteten Kanten zwischen benachbarten Steuerknoten: *Initialisierungskanten* E^{Init} und *Ausführungskanten* E^{Aus} (Bild 6.1).

Die Initialisierungskanten verbinden Knoten, die die Generierungssteuerung während der Initialisierungsphase verarbeitet. Im Gegensatz dazu verbinden die Ausführungskanten die Knoten, die die Generierungssteuerung während der Ausführungsphase verarbeitet. Die Initialisierungskanten bilden den so genannten *Initialisierungspfad*, weil ein Steuerknoten maximal eine ausgehende und eine eingehende Initialisierungskante besitzt. Der Initialisierungspfad ermöglicht der Generierungssteuerung einen linearen Durchlauf für das Anlegen der globalen und planlokalen Variablen und für das Auffinden der Startroutine.

Im Gegensatz zur Initialisierungsphase ist für die Ausführungsphase kein linearer Durchlauf möglich, z. B. aufgrund von Ausdrücken, Schleifen und Unterprogrammaufrufen. In der Ausführungsphase bestimmt eine Tiefensuche die Reihenfolge, in der die Generierungssteuerung die Knoten verarbeitet. Eine gerichtete Ausführungskante E^{Aus}_{ij} zeigt auf den Knoten V_j, den die Generierungssteuerung während der Ausführungsphase vor der Verarbeitung des Knotens V_i verarbeiten muss. Es können mehrere Ausführungskanten E^{Aus}_{ij} von einem Knoten V_i abgehen. Deshalb erhält jede Ausführungskante E^{Aus}_{ij} eine Priorität bezüglich des Knotens V_i. Diese Priorität legt fest, in welcher Reihenfolge die Generierungssteuerung die adjazenten Knoten verarbeiten muss. Sie liegt in Form einer Abbildung $Prio: \ E^{Aus} \longmapsto \mathbb{N}_0$ vor, wobei kleine Werte eine hohe Priorität bedeuten (Bild 6.1). Es werden alle adjazenten Knoten V_j eines Knoten V_i mit $Prio(E^{Aus}_{ij}) > 0$ vor Knoten V_i verarbeitet.. Anschließend erfolgt die Verarbeitung von V_i und abschließend einer der V_k mit $Prio(E^{Aus}_{ik}) = 0$. Welcher V_k mit $Prio(E^{Aus}_{ik}) = 0$ verarbeitet wird, hängt von der Verarbeitung von V_i ab.

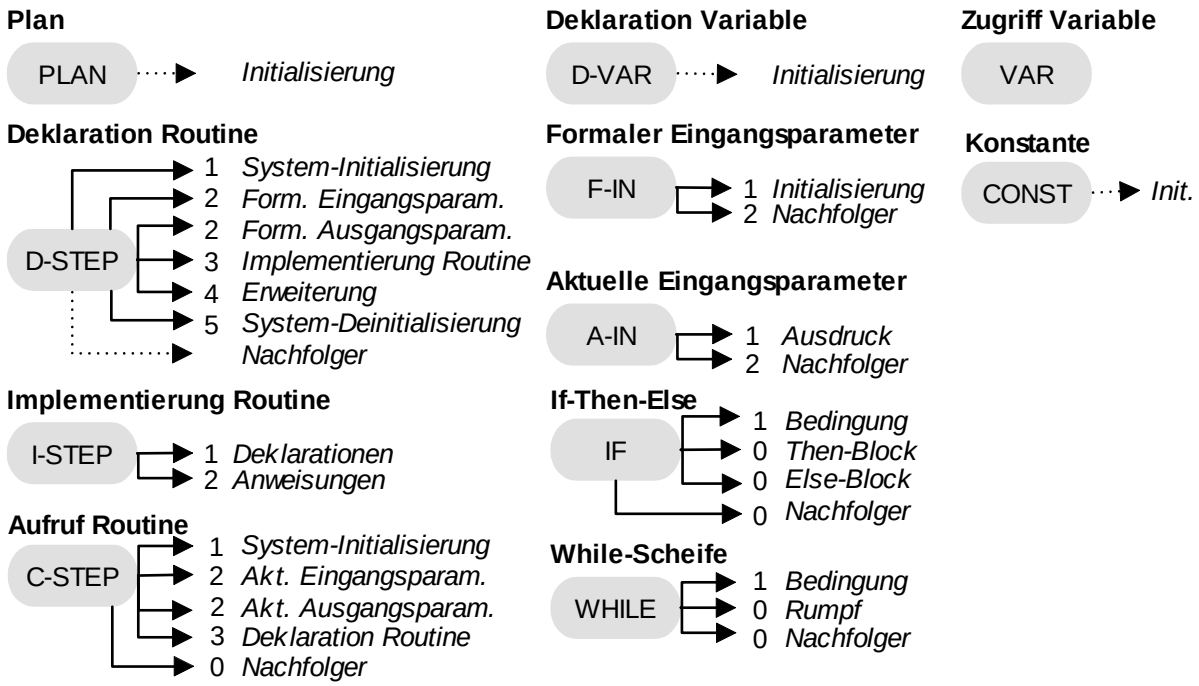


Bild 6.2: Auswahl an Steuerknoten

Es gilt also folgende Verarbeitungsreihenfolge für Knoten V_i:

1. Verarbeitung aller Steuerknoten V_j mit $Prio(E_{ij}^{Aus}) > 0$
2. Verarbeitung von Steuerknoten V_i
3. Verarbeitung eines Steuerknotens V_k mit $Prio(E_{ik}^{Aus}) = 0$

Eine Auswahl an Steuerknoten zeigt Bild 6.2. Besitzen zwei von einem Knoten ausgehende Ausführungskanten dieselbe Priorität, so spielt die Reihenfolge der Verarbeitung entweder keine Rolle (z. B. die Auswertung der Eingangs- und Ausgangsparameter des Knotens *Aufruf Routine)* oder sie hängt von der Verarbeitung eines adjazenten Knotens mit höherer Priorität ab. Für den Knoten *If-Then-Else* hängt z. B die Verarbeitung von Knoten *Then* oder von Knoten *Else* von der Verarbeitung des Knotens *Bedingung* ab. Bei Schleifen kann die Generierungssteuerung adjazente Knoten (Schleifenrumpf) mehrmals oder gar nicht verarbeiten, weil die Verarbeitung des Schleifenrumpfs von der Verarbeitung der Schleifenbedingung abhängt.

In der Aufbauphase baut die Generierungssteuerung für die Systemroutinen und Systemvariablen aus den Schnittstellenbeschreibungen der Funktionserweiterungen einen Subgraphen innerhalb des Steuergraphen auf. Der Subgraph enthält ausschließlich Steuerknoten der Typen *Plan, Deklaration Routine, Deklaration Variable* und *Konstante.* Der Subgraph ist ebenfalls ein Pfad, der an den Initialisierungspfad angehängt wird. Dieser Ansatz wird aus folgendem Grund gewählt. Die Generierungssteuerung muss die globalen Systemvariablen und -routinen zur Verarbeitung kennen. Da die Generierungssteuerung für die Verarbeitung des Initialisierungspfads ein internes Modul benötigt, kann dieses Modul auch die Bekanntmachung der Systemroutinen und Systemvariablen, d. h. die Verarbeitung des Subgraphen, übernehmen.

6.1.3 Beispiel

Zur Verdeutlichung des Konzepts des Steuergraphen wird hier ein Beispiel beschrieben. Dazu dient ein Ausschnitt eines Plans (Bild 6.3). Der Plan PRODUKT enthält eine planlokale Variable VARIANTE, die Routine START, bei dem die Ausführung beginnt, und einen Routinenaufruf ZUGRIFF_SIMULATIONSMODELL mit dem Eingangsparameter WERKSTÜCK. Beide Routinen enthalten weitere nicht dargestellte Anweisungen.

```
GENERATION PLAN PRODUKT

-- Variablen
$OBJECT VARIANTE = <Produkt C>;

-- Routinen
STEP START()
  ZUGRIFF_SIMULATIONSMODELL(WERKSTÜCK=VARIANTE);
  ...
ENDSTEP

STEP ZUGRIFF_SIMULATIONSMODELL($OBJECT WERKSTÜCK)
  ...
ENDSTEP
```

Bild 6.3: Beispiel eines Generierungsplans

Der zu dem Plan gehörende Steuergraph ist in Bild 6.4 dargestellt. Für alle Steuerknoten sind die Initialisierungs- und Ausführungskanten mit deren Prioritätswerten abgebildet. Die Kantenenden haben eine Markierung in Abhängigkeit davon, ob die Kante auf einen dargestellten Knoten oder auf einen nicht dargestellten Knoten zeigt, oder ob ein adjazenter Knoten nicht existiert.

Der Initialisierungspfad in diesem Beispiel enthält die folgenden Steuerknoten, die die Generierungssteuerung in der folgenden Reihenfolge verarbeitet:

1. *Plan* (PLAN, PRODUKT)
2. *Deklaration Variable* (D-VAR, VARIANTE)
3. *Konstante* (CONST, <Produkt C>)
4. *Deklaration Routine* (D-STEP, START)
5. *Deklaration Routine* (D-STEP, ZUGRIFF_SIMULATIONSMODELL).

Die Generierungssteuerung verarbeitet in der Ausführungsphase die Knoten V_i gemäß der Verarbeitungsreihenfolge (Abschnitt 6.1.2, S. 80) unter Verwendung der angegebenen Prioritäten. Im Folgenden werden nur die in Bild 6.4 dargestellten Knoten betrachtet. Beginnend bei Knoten *Deklaration Routine* (START) durchläuft die Generierungssteuerung die adjazenten Knoten mit den jeweils höchsten Prioritätswerten: *Implementierung Routine* (START), *Aufruf Routine* (ZUGRIFF_SIMULATIONSMODELL), *Aktueller Eingangsparameter* (WERKSTÜCK) und *Zugriff Variable*

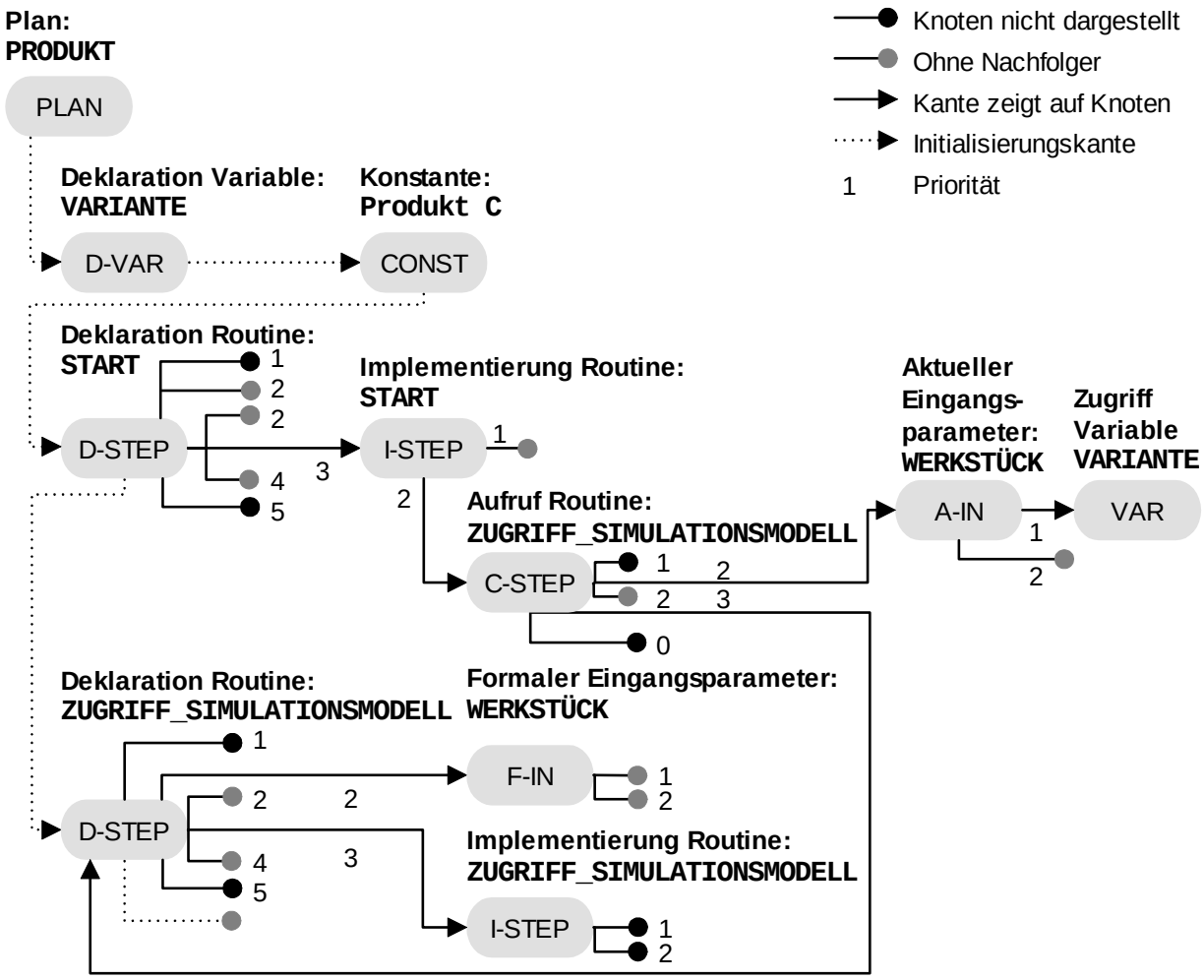


Bild 6.4: Beispiel eines Steuergraphen

(VARIANTE). Die Generierungssteuerung kann Letzteren jetzt verarbeiten, da dieser keine Nachfolger besitzt. Anschließend verarbeitet sie den Knoten *Aktueller Eingangsparameter* (WERKSTÜCK), da sie dessen Nachfolger soeben verarbeitet hat. Es ergibt sich die Verarbeitungsreihenfolge:

1. *Zugriff Variable* (VAR, VARIANTE)
2. *Aktueller Eingangsparameter* (A-IN, WERKSTÜCK)
3. *Formaler Eingangsparameter* (F-IN, WERKSTÜCK)
4. *Implementierung Routine* (I-STEP, ZUGRIFF_SIMULATIONSMODELL)
5. *Deklaration Routine* (D-STEP, ZUGRIFF_SIMULATIONSMODELL)
6. *Aufruf Routine* (C-STEP, ZUGRIFF_SIMULATIONSMODELL)
7. *Implementierung Routine* (I_STEP, START)
8. *Deklaration Routine* (D_STEP, START)

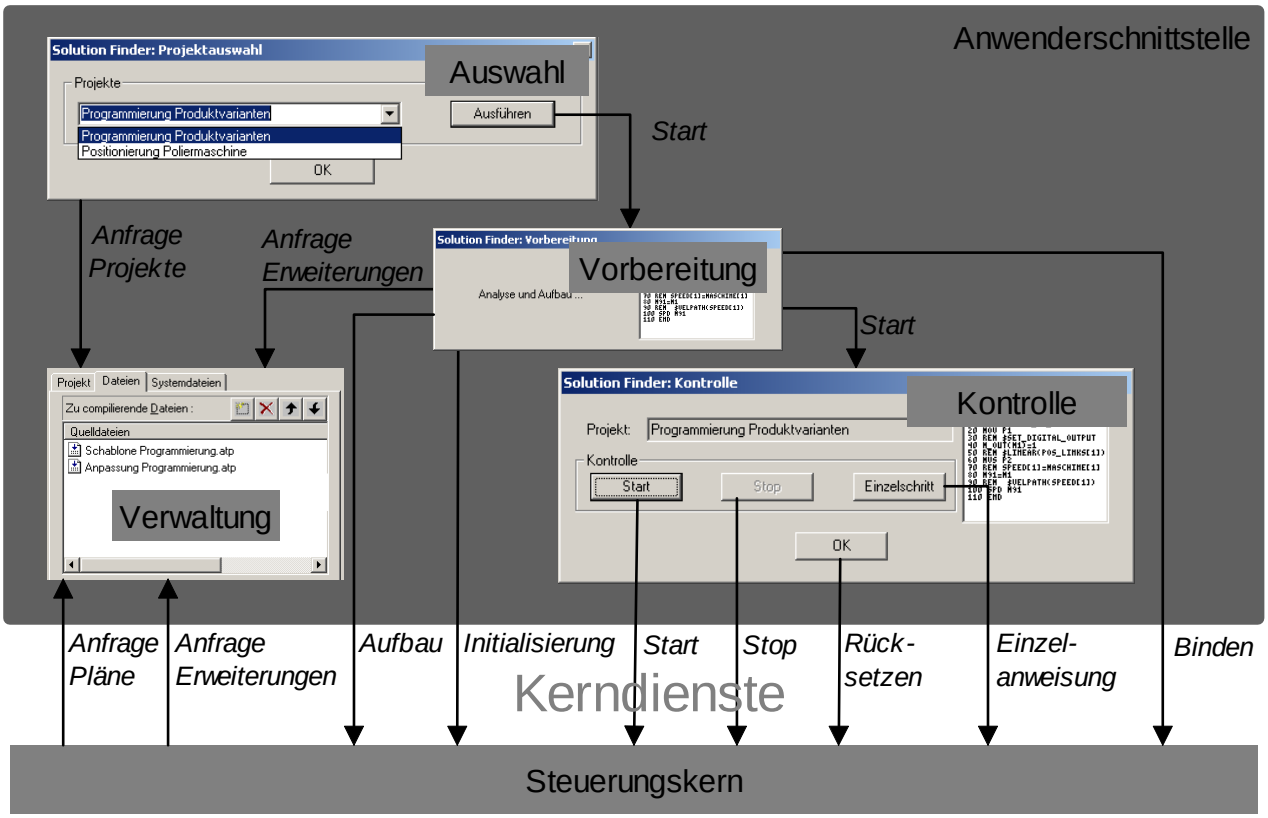


Bild 6.5: Aufbau der Anwenderschnittstelle

Bei den Knoten *Deklaration Routine* und *Implementierung Routine* ist nach Verarbeitung ihrer Nachfolger keine weitere Verarbeitung nötig. Diese Knoten wurden aus Strukturierungsgründen eingeführt.

6.2 Architektur

Die Architektur der Generierungssteuerung besteht aus den in Abschnitt 4.3.1 (S. 43) dargestellten Komponenten, die im Folgenden genauer betrachtet werden: Anwenderschnittstelle, Kern, Erweiterungsschnittstelle und Funktionserweiterung. Letztere kann die Dienste der Simulationsschicht nutzen. Die Konzeption der Funktionserweiterung und der Erweiterungsschnittstelle ist von entscheidender Bedeutung zur Erfüllung der Anforderungen 2 und 3 (S. 30).

6.2.1 Anwenderschnittstelle

Über die Anwenderschnittstelle kann der CAR-Anwender die Planausführung kontrollieren. Dazu zählt das Starten, das Stoppen, das Unterbrechen und das Ausführen einzelner Plananweisungen. Die Anwenderschnittstelle besteht aus den vier Modulen *Verwaltung*, *Auswahl*, *Vorbereitung* und *Kontrolle* (Bild 6.5), die dem Anwender jeweils einen Systemdialog zur Verfügung stellen.

Die Zusammenfassung der zu einer Applikation gehörenden Generierungspläne (Schablonen, Anpassungen, Bibliotheken) und Funktionserweiterungen erfolgt in so genannten *Generierungsprojekten.*

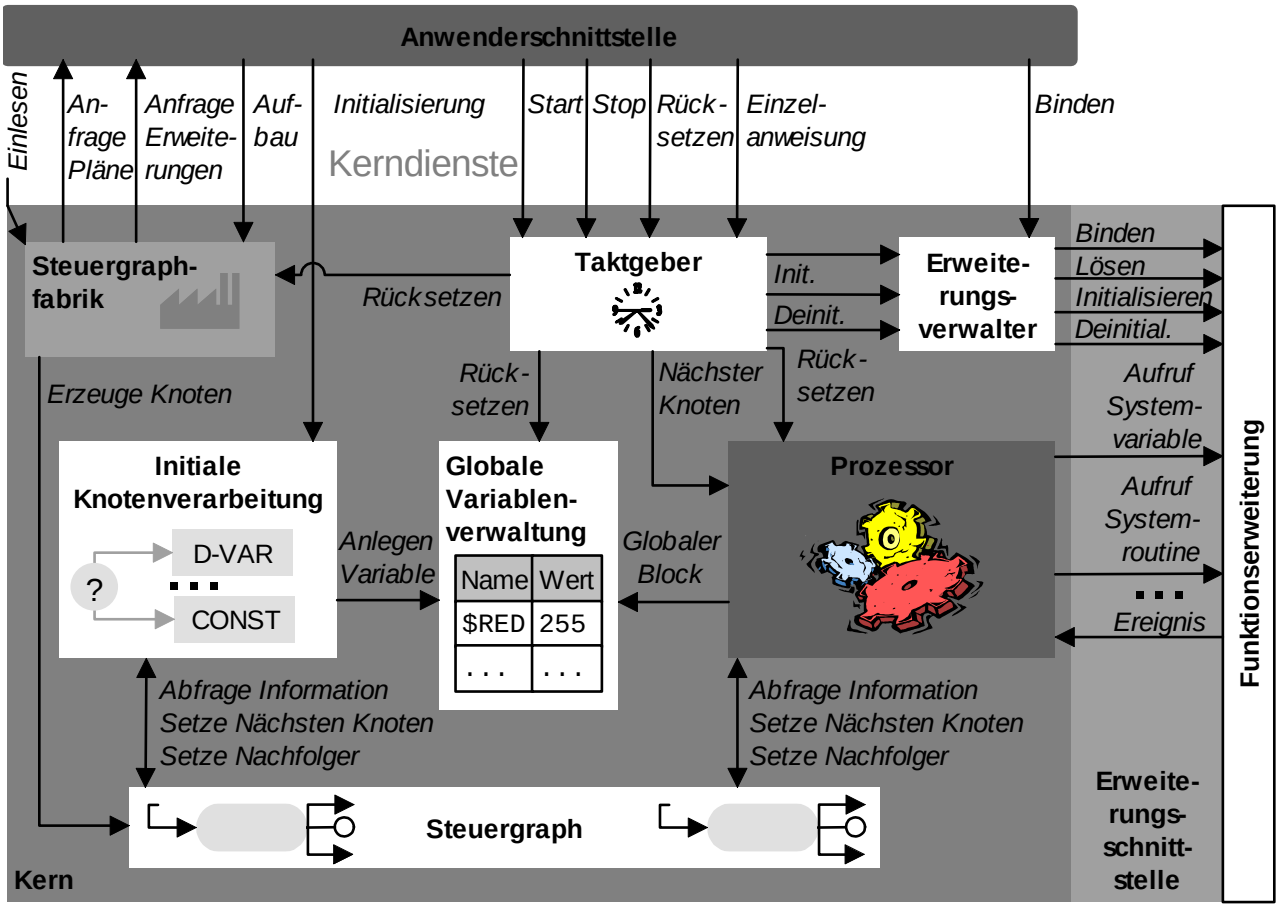


Bild 6.6: Aufbau des Kerns der Generierungssteuerung

Es existiert z. B. ein Projekt zur Programmgenerierung für variantenreiche Produkte der einleitenden Polierapplikation und ein Projekt für die Bestimmung der Poliermaschinenposition.

Mit dem Modul *Verwaltung* kann der Anwender die Generierungsprojekte erstellen, ändern und löschen. Das Modul ermöglicht dem CAR-Anwender festzulegen, dass der Kern nur die Funktionserweiterungen bindet, die er für eine Applikation benötigt.

Mit dem Modul *Auswahl* bestimmt der Anwender, welches Generierungsprojekt ausgeführt werden soll. Zur Ermittlung der verfügbaren Projekte stellt dieses Modul eine Anfrage an das Modul *Verwaltung*. Nachdem der Anwender ein Projekt ausgewählt hat, startet das Modul *Vorbereitung* durch Verwendung des Kerndienstes *Aufbau* den Aufbau des Steuergraphen. Nach Beendigung der Aufbauphase initiiert das Modul *Vorbereitung* die Anbindung der Funktionserweiterungen durch Verwendung des Kerndienstes *Binden*. Danach startet das Modul *Vorbereitung* durch Verwendung des Kerndienstes *Initialisierung* das Anlegen der globalen und planlokalen Variablen. Nach Beendigung der Initialisierungsphase kann der Anwender über das Modul *Kontrolle* die Ausführung der Pläne starten, abbrechen, unterbrechen und fortsetzen.

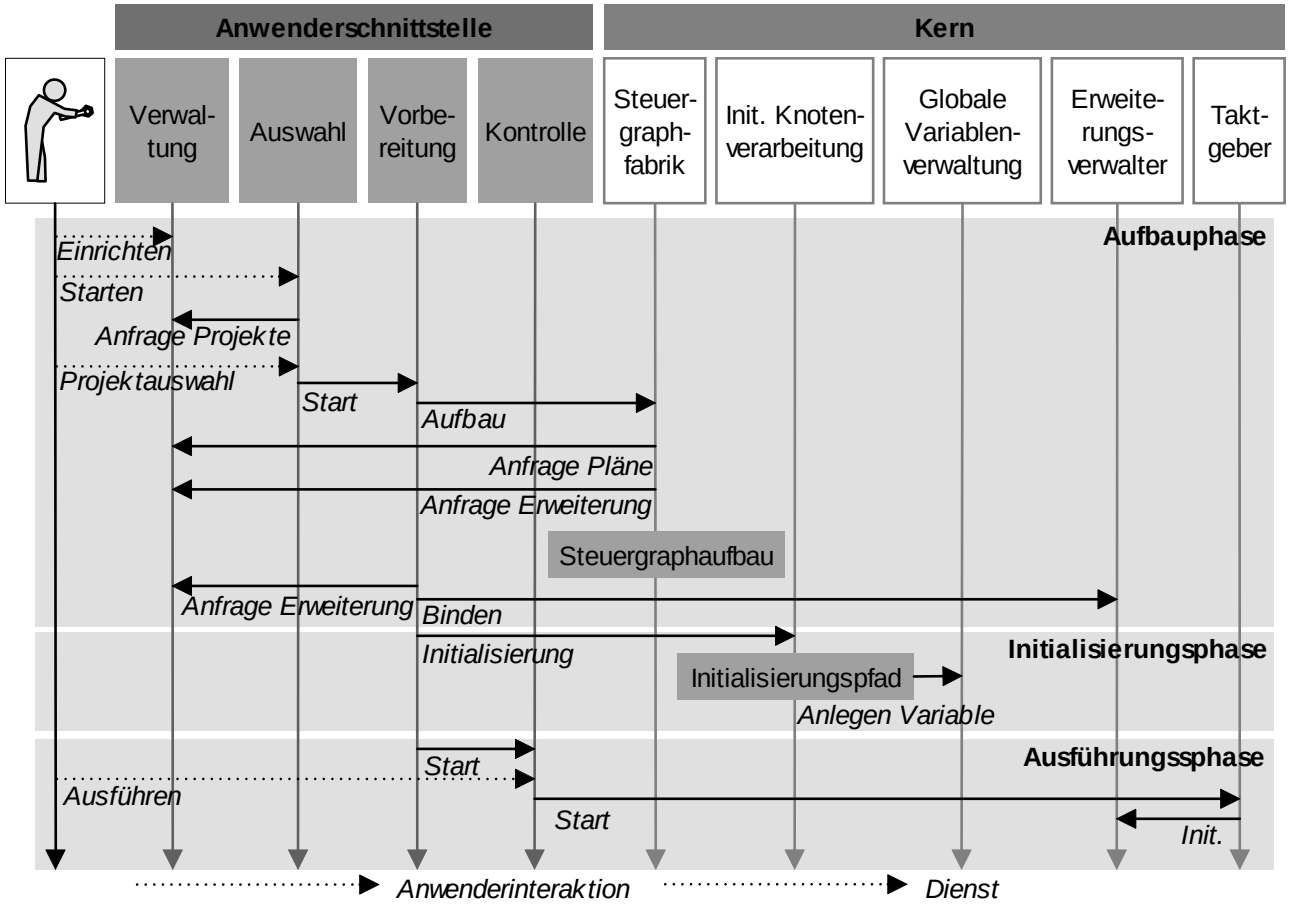


Bild 6.7: Arbeitsweise des Kerns

6.2.2 Steuerungskern

Der Kern selbst enthält die Module *Taktgeber*, *Erweiterungsverwalter, Steuergraphfabrik, initiale Knotenverarbeitung*, *globale Variablenverwaltung* und *Prozessor* (Bild 6.6). Die Kernmodule *initiale Knotenverarbeitung* und *Prozessor* arbeiten auf dem Steuergraphen.

In Bild 6.7 ist die Arbeitsweise des Kerns in der Aufbau- und Initialisierungsphase dargestellt. Des Weiteren veranschaulicht Bild 6.7 die Aufbau- und Initialisierungsphase und den Beginn der Ausführungsphase der Generierungsteuerung.

Die *Steuergraphfabrik*, die im nächsten Abschnitt genauer erläutert wird, erzeugt bei Inanspruchnahme des Kerndienstes *Aufbau* aus den Generierungsplänen (Schablonen, Anpassungen, Bibliotheken) und den Schnittstellenbeschreibungen der Funktionserweiterungen den Steuergraphen. Sie ist ebenfalls für den Abbau des Steuergraphen nach dessen Verarbeitung zuständig (Dienst *Rücksetzen*). Der *Erweiterungsverwalter* bindet in der Aufbauphase die Funktionserweiterungen an den Kern (Dienst *Binden*) und ruft den Dienst *Initialisierung* auf, bei dem die Funktionserweiterung spezifische Initialisierungen durchführen kann.

Die *initiale Knotenverarbeitung* ist für den ersten Durchlauf des Steuergraphen (Initialisierungsphase) zuständig (Dienst *Initialisierung*) und durchläuft den Initialisierungspfad des Steuergraphen (S. 79ff). Sie kann ausschließlich Knotentypen verarbeiten, die im Initialisierungspfad auftreten können. In Ab-

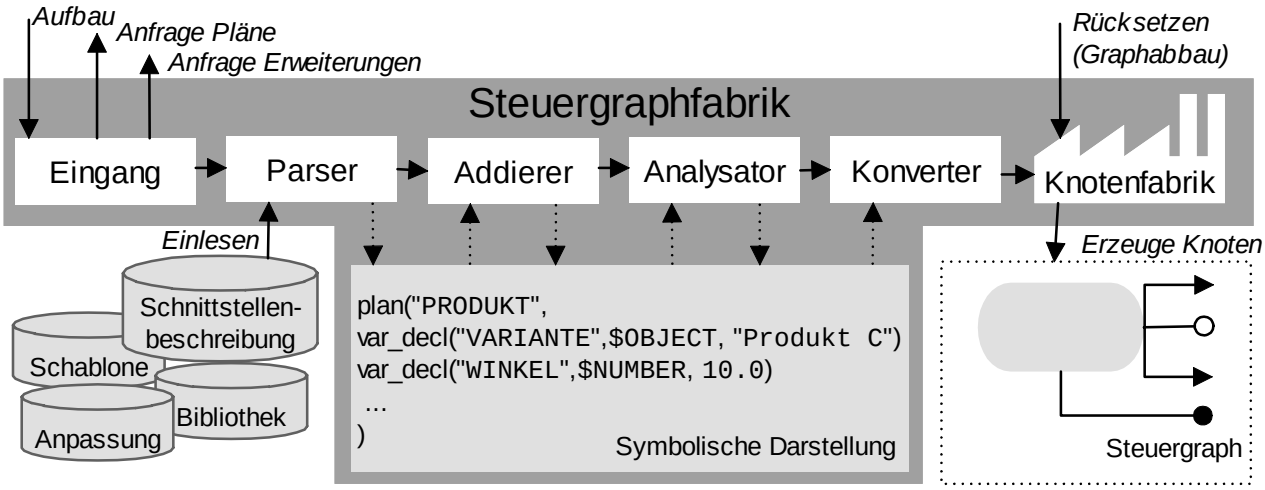


Bild 6.8: Aufbau der Steuergraphfabrik

hängigkeit des Knotentyps verarbeitet sie diese Steuerknoten auf unterschiedliche Weise und ruft dazu Dienste anderer Kernmodule auf. Sie legt in der Initialisierungsphase alle globalen und planlokalen Variablen in dem Modul *globale Variablenverwaltung* durch jeweilige Beanspruchung des Diensts *Anlegen Variable* an. Die *globale Variablenverwaltung* besitzt eine Symboltabelle, die die Zuordnung zwischen Name und Wert einer Variablen herstellt. Der Anwender kann die Initialisierungsphase nicht unterbrechen.

Der *Prozessor* ist für den zweiten Durchlauf des Steuergraphen (Ausführungsphase) zuständig. Er verarbeitet die Steuerknoten entlang der Ausführungskanten. Im Prozessor sind alle ausführungsrelevanten Daten gespeichert, z. B. die Werte von routinenlokalen Variablen, mit Ausnahme der Werte von globalen und planlokalen Variablen. Für diese besitzt er Zugriff auf die *globale Knotenverwaltung* (Dienst *Globaler Block*). Der Prozessor verarbeitet pro Aufruf von *Nächster Knoten* exakt einen Steuerknoten. Dieser Dienst wird vom *Taktgeber* so lange aufgerufen, bis der zweite Durchlauf durch den Steuergraphen und damit die Ausführung der Pläne beendet ist. Über den Rückgabewert des Diensts *Nächster Knoten* erkennt der Taktgeber das Ausführungsende. Der Taktgeber kann über den Kerndienst *Stop* angehalten und somit die Ausführungsphase unterbrochen werden. Über den Kerndienst *Start* wird die Ausführung fortgesetzt. Bei Beanspruchung des Diensts *Einzelanweisung* wird pro Aufruf eine Plananweisung ausgeführt. Der Anwender kann die Ausführung der Pläne vorzeitig beenden. Dazu ruft die Anwenderschnittstelle den Kerndienst *Rücksetzen* auf, den der Taktgeber an die betreffenden Kernmodule weiterleitet.

Steuergraphfabrik

Die *Steuergraphfabrik* ist für den Auf- und Abbau des Steuergraphen zuständig und besteht aus den Modulen *Eingang*, *Parser*, *Addierer*, *Analysator*, *Konverter* und *Knotenfabrik* (Bild 6.8).

Um die Arbeitsweise der *Steuergraphfabrik* zu verdeutlichen, wird im Folgenden der Ablauf der Steuergrapherzeugung erläutert. Nachdem das Modul *Eingang* vom Modul *Vorbereitung* der überlagerten Anwenderschnittstelle den Auftrag zum Aufbau des Graphen bekommen hat, fordert es beim Modul

Verwaltung (Bild 6.5) die Dateinamen der zuverwendenden Schablonen, Anpassungen, Bibliotheken und Schnittstellenbeschreibungen der Funktionserweiterungen an. Der nachgeschaltete *Parser* liest die Dateien ein und erzeugt aus deren Inhalt eine interne Repräsentation. Die Steuergraphfabrik wurde unter Verwendung des Compiler Construction Systems GENTLE realisiert [144], weil sich damit die Module der Steuerfabrik einfach, schnell und effizient realisieren lassen. In GENTLE arbeitet man auf einer internen Repräsentation in Form von einer symbolischen Darstellung (Prädikate). Der *Addierer* führt auf dieser symbolischen Darstellung die Operation der Planaddition (Abschnitt 5.1.6, S. 61ff) aus. Der *Analysator* verwendet das Ergebnis der Planaddition für eine Kontextanalyse. Dazu gehört u. a. die Herstellung des Bezugs zwischen der Deklaration und der Verwendung eines Variablenamens, die Typprüfung zwischen den aktuellen und den formalen Parametern, und die Prüfung, ob für jeden ausgelassenen aktuellen Parameter, der zugehörige formale Parameter einen Initialisierungswert besitzt.

Aus dem Ergebnis des *Analysators* baut der *Konverter* unter Zuhilfenahme der *Knotenfabrik* den Steuergraphen auf. Der *Konverter* durchläuft die symbolische Darstellung und sammelt aus diesen die Informationen (z. B. Namen, Datentypen), die die *Knotenfabrik* für die Erzeugung der Steuerknoten, sowie der Initialisierungs- und Ausführungskanten benötigt.

Prozessor

Nachdem die *initiale Knotenverarbeitung* den Initialisierungspfad des Steuergraphen durchlaufen hat und sie dadurch alle globalen und planlokalen Variablen in der *globalen Variablenverwaltung* angelegt hat, durchläuft der *Prozessor* den Steuergraphen ein zweites Mal, um die Plananweisungen auszuführen. Der *Prozessor* besteht selbst wieder aus den Modulen: *Lokale Variablenverwaltung*, *Knotenverarbeitung*, *Ereignisverwaltung* und *Erweiterungsumsetzer* (Bild 6.9).

Der *Erweiterungsumsetzer* leitet die Dienstanforderungen der Erweiterungsschnittstelle an die Prozessormodule weiter - und umgekehrt. Die *lokale Variablenverwaltung* ist für die Organisation der Variablen (Anlegen, Archivieren, Abfrage) zuständig. In diesem Modul befindet sich ein Stack, auf dem die Gültigkeitsbereiche (Blöcke) der routinenlokalen Variablen in Form von Tabellen (Name, Wert) liegen. Des Weiteren enthält die *lokale Variablenverwaltung* eine Schnittstelle zum Variablenarchiv, um darin Variablen zu speichern (Abschnitt 5.1.5, S. 60ff).

Die *Knotenverarbeitung* ist für die Ausführung der Steuerknoten zuständig. Sie verarbeitet einen Knoten, wenn der *Taktgeber* den Dienst *Nächster Knoten* aufruft. Die noch zu verarbeitenden Knoten sind in einem Knotenstack abgelegt. In Abhängigkeit vom Typ des obersten Stackknotens beauftragt die *Knotenverarbeitung* andere Prozessormodule, z. B. wird die *lokale Variablenverwaltung* angewiesen, einen neuen Gültigkeitsbereich anzulegen (Dienst *Anlegen Block*). Jeden Knotentyp verarbeitet die *Knotenverarbeitung* auf eine andere Weise. Sie enthält neben dem Knotenstack einen *Expressionstack* für die Berechnung von im Plan auftretenden Ausdrücken und einen *Parameterstack* für die Parameterübergabe bei einem Routinenaufruf. Ein Element des Parameterstacks enthält jeweils den Wert und den Namen des zugehörigen formalen Parameters. Im Gegensatz dazu enthält der Expressionstack nur Werte.

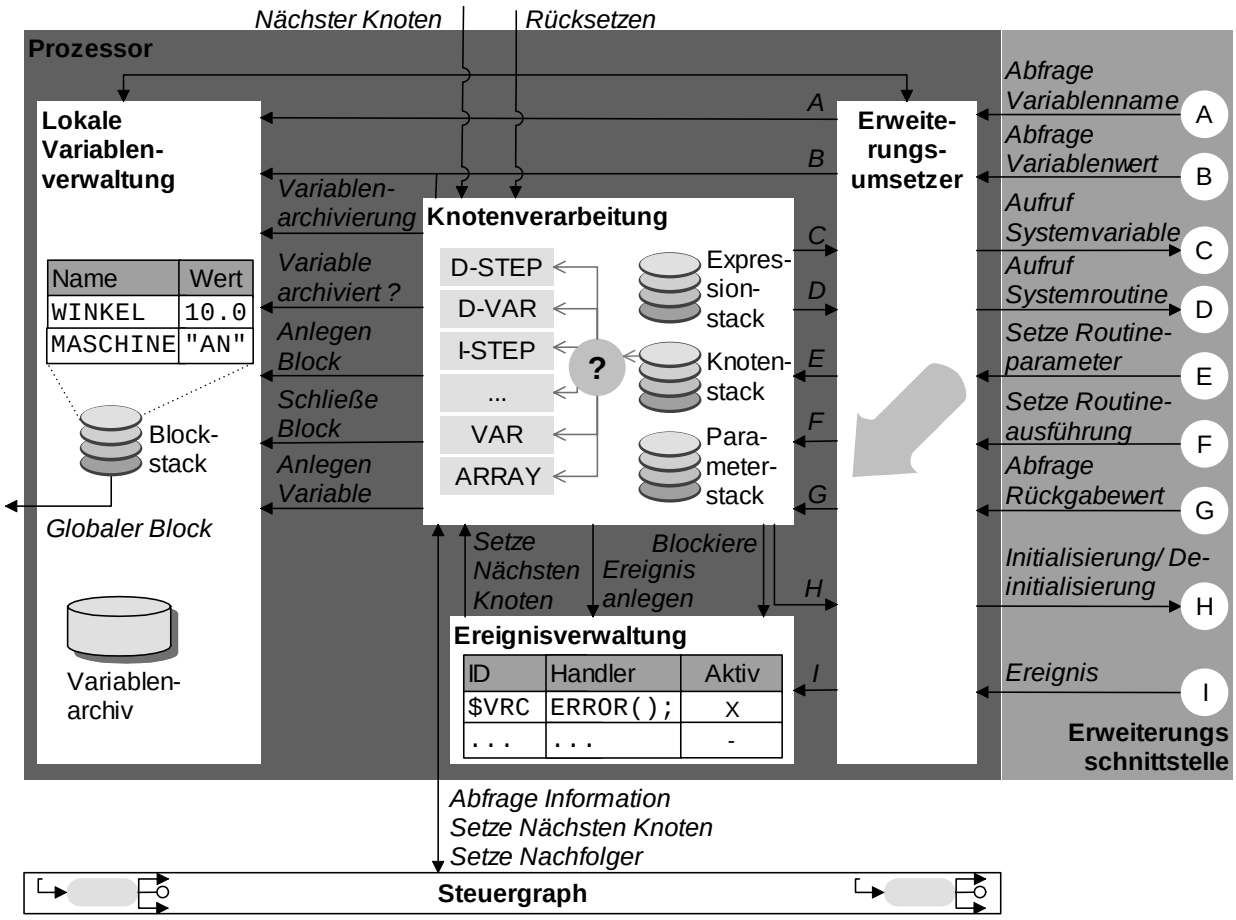


Bild 6.9: Aufbau des Prozessors

Die *Ereignisverwaltung* enthält eine Tabelle mit möglichen Ereignissen und den zugehörigen Routinenaufrufen für deren Behandlung (Abschnitt 5.4, S. 59ff). Die *Knotenverarbeitung* füllt diese Tabelle über den Dienst *Ereignis anlegen* bei der Verarbeitung des Knotens, der der Plananweisung `ON <Event> CALL <Routine>` entspricht. Bei Auftritt eines Ereignisses legt die *Ereignisverwaltung* den zu dem Routineaufruf gehörenden Knoten auf den Knotenstack der *Knotenverarbeitung*, die diesen beim nächsten Aufruf des Diensts *Nächster Knoten* verarbeitet.

Bild 6.10 zeigt den Ablauf der Knotenverarbeitung in der Ausführungsphase. Der Taktgeber beauftragt die *Knotenverarbeitung* durch Aufruf des Diensts *Nächster Knoten*. Diese entnimmt das oberste Knotenstackelement (Knoten 1) und fordert von diesem die zur Verarbeitung benötigen Informationen an. Je nach Knotentyp beauftragt die *Knotenverarbeitung* andere Module (z. B. *lokale Variablenverwaltung, Ereignisverwaltung, Erweiterungsumsetzer)*. Die *Knotenverarbeitung* trifft im Verlauf der Verarbeitung von V_i die Entscheidung, welche der Knoten V_k mit $Prio(E_{ik}^{Aus}) = 0$ als Nächstes verarbeitet wird (Knoten 2). Abschließend fordert sie Knoten 2 auf, dessen Nachfolger zu benennen (*Setze Nachfolger*). Knoten 2 legt zuerst sich auf den Knotenstack. Anschließend legt er die jeweiligen Knoten in der Reihenfolge absteigender Prioritätswerte $Prio(E_{ij}^{Aus})$ mit $Prio(E_{ij}^{Aus}) > 0$ auf den Knotenstack, auf die seine Ausführungskanten E_{ij}^{Aus} verweisen (*Setze Nächsten Knoten*). Somit ist die Verarbeitungsreihenfolge gewährleistet (Abschnitt 6.1.2, S. 80), dass die *Knotenverarbeitung* zuerst alle Knoten verarbeitet, die Knoten 2 für die Verarbeitung benötigt, bevor sie Knoten 2 verarbeitet.

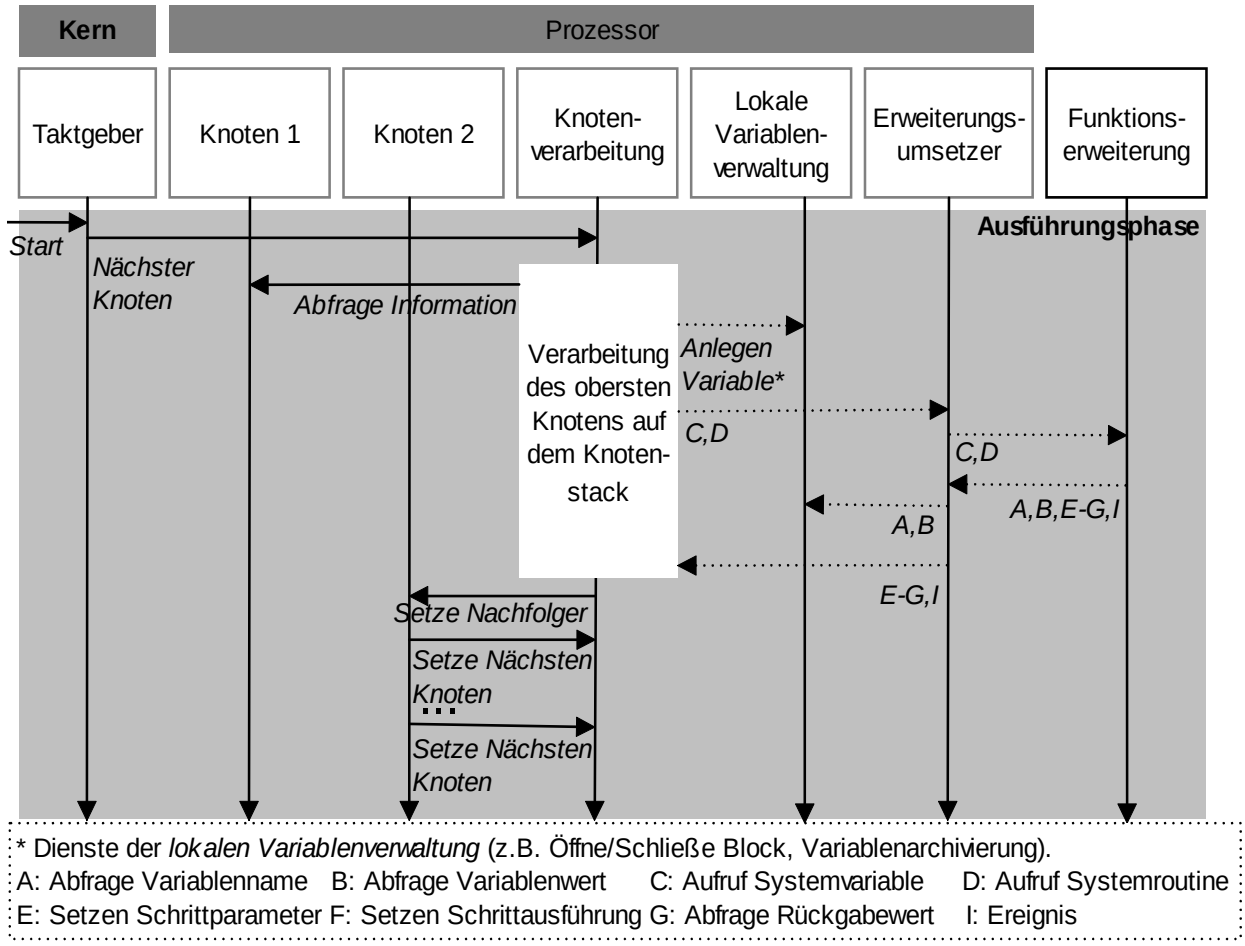


Bild 6.10: Sequenzdiagramm zur Bearbeitung des Steuergraphen in der Ausführungsphase

Beispiel Knotenverarbeitung

Im Folgenden soll die Arbeitsweise der *Knotenverarbeitung* an einem Beispiel erläutert werden (Bild 6.11). Dazu werden der betrachtete Generierungsplan (Bild 6.3, S. 81) und der zugehörige Steuergraph (Bild 6.4, S. 82) herangezogen. In Bild 6.11 werden die ersten drei Knotenverarbeitungen betrachtet (Abschnitt 6.1.3, S. 82).

Vor der Verarbeitung des ersten Knotens *Zugriff Variable* (VAR, `VARIANTE`), liegen die zu verarbeitenden Knoten unter Berücksichtigung der jeweiligen Prioritätswerte $Prio(E_{ij}^{Aus}) > 0$ der Ausführungskanten E_{ij}^{Aus} auf dem Knotenstack. Bei der Verarbeitung des ersten Steuerknotens *Zugriff Variable* (VAR, `VARIANTE`) ruft die *Knotenverarbeitung* den Dienst *Abfrage Variablenwert* von der *lokalen Variablenverwaltung* auf. Dieser Dienst erfordert die Angabe des zugehörigen Variablennamens (`VARIANTE`) und liefert deren Wert (`Produkt C`). Dazu schaut die *lokale Variablenverwaltung* in der Symboltabelle des obersten Blockstackelements nach (Bild 6.9). Nach Erhalt des Variablenwerts legt ihn die *Knotenverarbeitung* auf den Expressionstack. Damit ist die Verarbeitung des ersten Steuerknotens VAR beendet.

Vor der Verarbeitung des zweiten Steuerknotens *Aktueller Eingangsparameter* (A_IN, `WERKSTÜCK`), dessen Ausführungskante vom Knoten *Aufruf Routine* (C-STEP, `ZUGRIFF_SIMULATIONSMODELL`)

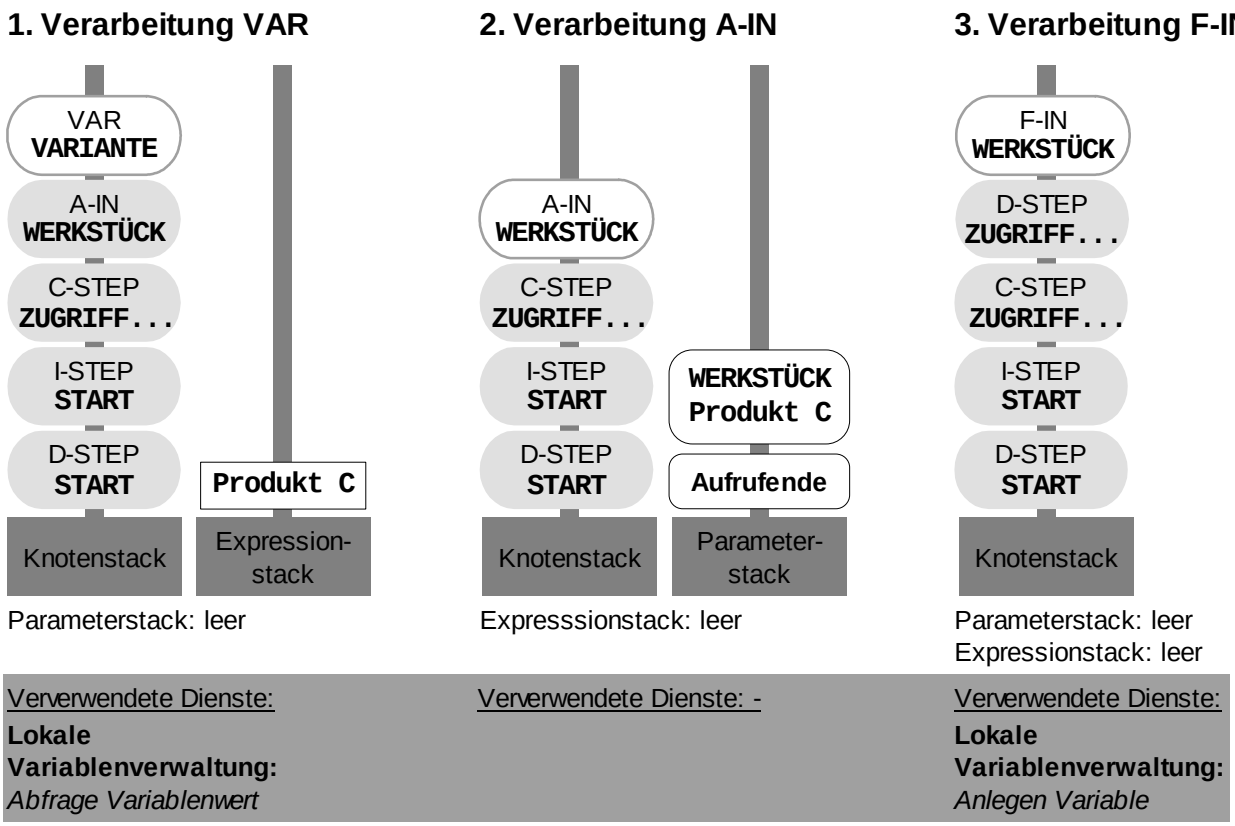


Bild 6.11: Beispiel zur Arbeitsweise der Knotenverarbeitung

die Priorität zwei besitzt, erfolgt die Verarbeitung eines nicht dargestellten Knotens, dessen Ausführungskaten vom Knoten C-STEP die Priorität eins besitzt. Darin wird das Aufrufende auf dem Parameterstack gekennzeichnet.

Bei der Verarbeitung des zweiten Steuerknotens A-IN nimmt die *Knotenverarbeitung* das oberste Element des Expressionstacks und legt einen neuen aktuellen Parameter mit dessen Namen (`WERKSTÜCK`) und Wert (`Produkt C`) auf den Parameterstack. Damit ist die Verarbeitung des zweiten Steuerknotens A-IN beendet.

Vor der Verarbeitung des dritten Steuerknotens *Formaler Eingangsparameter* (F_IN, `WERKSTÜCK`), dessen Ausführungskante vom Knoten *Deklaration Routine* (D-STEP, `ZUGRIFF_SIMULATIONS-MODELL`) die Priorität zwei besitzt, erfolgt die Verarbeitung eines nicht dargestellten Knotens, dessen Ausführungskaten vom Knoten D-STEP die Priorität eins besitzt. Darin wird unter Verwendung des Diensts *Anlegen Block* ein neuer Block auf dem Blockstack erzeugt, der die Symboltabelle für den Gültigkeitsbereich der Routine `ZUGRIFF_SIMULATIONSMODELL` enthält.

Bei der Verarbeitung des dritten Steuerknotens *Formaler Eingangsparameter* (F_IN, `WERKSTÜCK`) entfernt die *Knotenverarbeitung* das oberste Stackelement, das den Namen (`WERKSTÜCK`) des formalen Parameters besitzt. Die Suche auf dem Parameterstack endet spätestens bei einem Aufrufendeelement. Das gesuchte Element muss nicht existieren, da der CAR-Anwender einzelne Parameter bei Routinenaufrufen weglassen kann (Abschnitt 5.1.5, S. 58). In diesem Fall liegt der Initialisierungswert auf dem Expressionstack. Dieser Wert wurde aufgrund des adjazenten Knotens mit der Ausführungs-

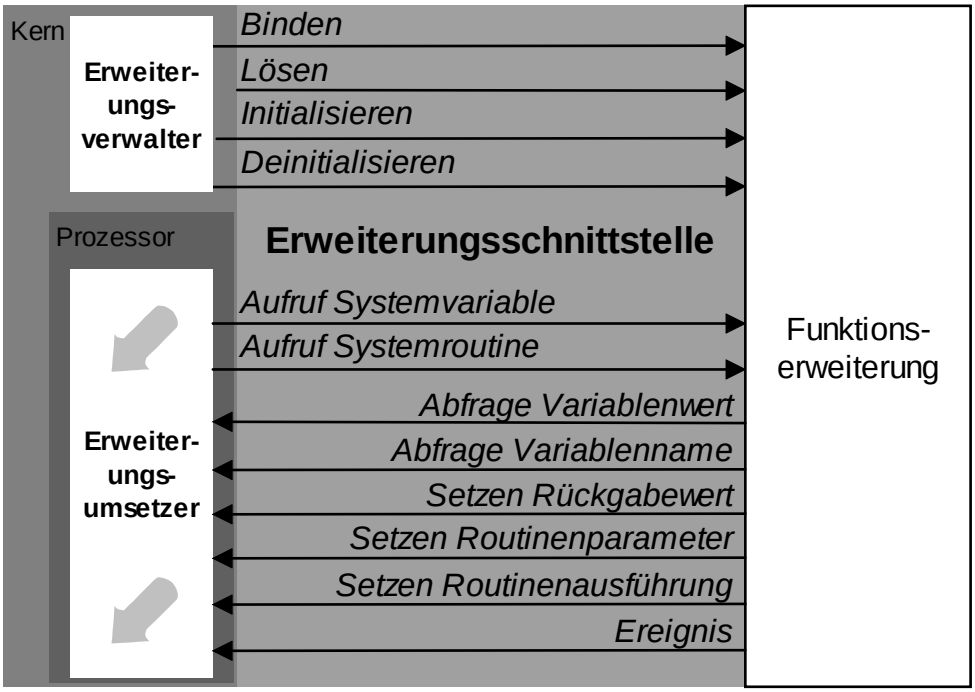


Bild 6.12: Erweiterungsschnittstelle der Generierungssteuerung

kante der Priorität eins des Steuerknotens F-IN berechnet. Abschließend legt die *Knotenverarbeitung* eine Variable `WERKSTÜCK` für den formalen Parameter unter Verwendung des Diensts *Anlegen Variable* bei der *lokalen Variablenverwaltung* an.

6.2.3 Erweiterungsschnittstelle

Die Erweiterungsschnittstelle dient dazu, Funktionserweiterungen an die Generierungssteuerung anzubinden, um dieser neue Basisfunktionen hinzuzufügen. Die Anforderungen, die an die Erweiterungsschnittstelle gestellt werden, sind:

1. Eine Erweiterung muss möglichst viele Kernfunktionen verwenden können, damit man eine neue Erweiterung ohne Eingriffe in den Kern entwickeln kann.

2. Der Zahl der Dienste muss so gering wie möglich sein, sodass ein Erweiterungsentwickler die Schnittstelle schnell erlernen kann.

Die in dieser Arbeit vorgeschlagene Schnittstelle (Bild 6.12) enthält einerseits Aufrufe aus dem Kern in die Funktionserweiterung (z. B. *Aufruf Systemvariable, Aufruf Systemroutine*) und andererseits aus der Erweiterung in den Kern (z. B. *Abfrage Variablenwert*, *Setzen Routinenausführung*).

Zur Anbindung einer Erweiterung an den Kern ruft der Erweiterungsverwalter im Kern die Funktion *Binden* auf. Entsprechend existiert eine Funktion *Lösen* zum Lösen der Bindung. Beim Dienst *Binden* kann die Funktionserweiterung Initialisierungen (z. B. Speicherallokation) durchführen, die sie für

alle Planausführungen benötigt. Entsprechend dient der Dienst *Lösen* zum Aufräumen dieser Initialisierungen. Es existieren Aufrufe *Initialisieren* und *Deinitialisieren*, die der Erweiterungsverwalter vor und nach einer Planausführung aufruft, damit die Erweiterung Initialisierungen und Deinitialisierungen für jede Planausführung durchführen kann.

Wird für die Planausführung ein lesender oder schreibender Zugriff auf eine in der Funktionserweiterung enthaltene Systemvariable benötigt, so ruft die Knotenverarbeitung im Prozessor über den Erweiterungsumsetzer den Dienst *Aufruf Systemvariable* auf. Entsprechend existiert für benötigte Systemroutinen der Dienst *Aufruf Systemroutine*. Während der Ausführung des Diensts *Aufruf Systemvariable* oder *Aufruf Systemroutine* kann die Erweiterung folgende Dienste verwenden:

1. *Abfrage Variablenwert*: Der Erweiterungsumsetzer liefert den Wert zum Namen einer Variablen unter Verwendung der lokalen Variablenverwaltung. Die lokale Variablenverwaltung fordert für die globalen oder planlokalen Variablen den Variablenwert von der globalen Variablenverwaltung an. In beiden Variablenverwaltungen sind die Zuordnungen zwischen Wert und Name einer Variable in den Symboltabellen der Blöcke enthalten.

2. *Abfrage Variablenname*: Ein Variablenwert besitzt in den Symboltabellen neben den eigentlichen Wert einen systeminternen Bezeichner. Der Erweiterungsumsetzer liefert zu einem systeminternen Bezeichner den zugehörigen Variablennamen. Der Erweiterungsumsetzer fordert den Name bei der lokalen Variablenverwaltung an, die gegebenenfalls die Anforderung an die globale Variablenverwaltung weiterleitet.

3. S*etzen Rückgabewert*: Dieser Dienst teilt für Systemroutinen mit Rückgabewert den Rückgabewert der Knotenverarbeitung mit. Die Knotenverarbeitung legt den Rückgabewert zur Weiterverarbeitung auf den Expressionstack.

4. *Setzen Routinenparameter*: Dieser Dienst legt einen Parameter zur Weiterverarbeitung durch den Dienst *Setzen Routinenausführung* auf den Parameterstack der Knotenverarbeitung.

5. *Setzen Routinenausführung*: Dieser Dienst ruft die angegebene Routine auf. Die zugehörigen Parameter müssen auf dem Parameterstack über den Dienst *Setzen Routinenparameter* gelegt worden sein. Der Dienst kann sowohl Systemroutinen als auch Routinen aufrufen, die der CAR-Anwender in einem Plan implementiert hat.

6. *Ereignis*: Dieser Dienst löst ein Ereignis aus. Der Erweiterungsumsetzer leitet das Ereignis an die Ereignisverwaltung weiter. Diese schaut nach, ob eine Routine aufgerufen werden muss. Falls ja, wird ein entsprechender Steuerknoten *Aufruf Routine* auf den Knotenstack der Knotenverarbeitung über den Dienst *Setzen Nächster Knoten* gelegt. Die Knotenverarbeitung verarbeitet diesen Knoten beim nächsten Aufruf von Dienst *Nächster Knoten*.

Ein Beispiel für den Dienst *Setzen Routinenausführung*, der eine vom Anwender in einem Plan implementierte Routine aufruft, ist der Aufruf der Routine `ZIELFUNKTION` von der Systemroutine `$OPTIMIZATION`, die bei der Schablone zur Layoutoptimierung zum Einsatz kommt (Bild 5.14, S. 73).

6.2.4 Funktionserweiterung

Die Funktionen, die die Generierungssteuerung dem CAR-Anwender in Form von Systemvariablen und Systemroutinen für die Verwendung in den Generierungsplänen zur Verfügung stellt, sind in so genannten *Funktionserweiterungen* realisiert. Eine Funktionserweiterung ist über die Erweiterungsschnittstelle an den Steuerungskern gekoppelt (Bild 4.5, S. 43).

Statische versus dynamische Anbindung

Für das Anbinden einer Funktionserweiterung an den Kern gibt es zwei Möglichkeiten: statische oder dynamische Anbindung.

Bei der *statischen Anbindung* kennt der Steuerungskern die in der Funktionserweiterung realisierten Systemvariablen und -routinen bereits zum Übersetzungszeitpunkt. Bei der *dynamischen Anbindung* kennt der Steuerungskern diese Variablen und Routinen im Vorfeld der Planausführung nicht. Die zweite Möglichkeit besitzt den Vorteil, dass man bei der Erweiterung der Generierungssteuerung den Steuerungskern nicht ändern muss. Ein weiterer Vorteil besteht darin, dass nicht benötigte Systemvariable und Systemroutinen, nicht an den Steuerungskern gebunden werden müssen. Damit wird die Generierungssteuerung skalierbar.

Wegen dieser beiden Vorteile wird hier die zweite Möglichkeit umgesetzt. Dazu muss aber eine Schnittstelle zwischen Erweiterung und Steuerungskern entworfen werden, über die jede Erweiterung mit dem Steuerungskern kommunizieren kann. Dies wurde mit dem Entwurf der Erweiterungsschnittstelle bereits erledigt.

Des Weiteren muss eine Beschreibung der in der Erweiterung realisierten Systemvariablen und -routinen entwickelt werden, damit der Steuerungskern diese vor einer Planausführung kennen lernt. Die Darstellung der Schnittstellenbeschreibung wird in Abschnitt 6.2.4 beschrieben.

Anbindungszeitpunkt

Für den Zeitpunkt der dynamischen Anbindung einer Funktionserweiterung an den Kern existieren zwei Möglichkeiten.

Bei der ersten Möglichkeit findet die Anbindung während der Ausführungsphase statt, d. h. zu dem Zeitpunkt, an dem der Kern die Funktionen aus der Erweiterung benötigt. Bei der zweiten Möglichkeit findet die Anbindung vor der Ausführungsphase statt.

Der Vorteil der Anbindung während der Planausführung besteht darin, dass der Kern die Anbindung nur für die benötigten Erweiterungen durchführen muss. Ein Nachteil besteht in einer längeren Ausführungsphase, da der Vorgang der Anbindung je nach durchgeführten Initialisierungen der Funktionserweiterung (Dienst *Binden*) eine gewisse Zeit dauern kann. Ein weiterer Nachteil besteht darin, dass der Kern das Fehlen von Erweiterungen erst in der Ausführungsphase feststellt. Diese beiden Nachteile bestehen bei der zweiten Möglichkeit nicht. Allerdings finden möglicherweise Anbindungen von Erweiterungen vor der Planausführung statt, die der Kern in der Ausführungsphase nicht

benötigt. Trotz dieses Nachteils wird aus den oben genannten Gründen die zweite Möglichkeit, d. h. die Anbindung der Funktionserweiterungen vor der Ausführungsphase, umgesetzt. Zur Beseitigung des letztgenannten Nachteils kann der CAR-Anwender im Modul *Verwaltung* der Anwenderschnittstelle (Bild 6.5, S. 83) neben den Plänen auch die für die Planausführung benötigten Erweiterungen angeben.

Aufbau

Eine Funktionserweiterung der Generierungssteuerung benötigt eine Schnittstellenbeschreibung und diese muss die folgenden Anforderungen erfüllen:

1. Für den CAR-Anwender soll unmittelbar erkennbar sein, welche Variablen und Routinen eine Erweiterung zur Verfügung stellt und wie er sie in seinen Plänen verwenden muss.
2. Der Steuerungskern muss aus der Schnittstellenbeschreibung ermitteln können, wie er die Systemvariablen und -routinen der Erweiterung aufrufen muss.

Um beide Anforderungen zu erfüllen, wird zur Formulierung der Schnittstellenbeschreibung die Syntax der Plansprache verwendet. Dadurch wird einerseits die Schnittstellenbeschreibung für den CAR-Anwender verständlich, weil er die Kenntnis der Plansyntax besitzt. Andererseits kann der Steuerungskern die Schnittstellenbeschreibung in einem für ihn verständlichen Format einlesen.

In der *Schnittstellenbeschreibung* sind die Deklarationen aller von dieser Erweiterung zur Verfügung gestellten Systemvariablen und Systemroutinen enthalten (Bild 6.13). Die *Funktionserweiterung* enthält die in der Schnittstellenbeschreibung zugehörigen Realisierungen.

In Bild 6.13 sind exemplarisch die Deklarationen von zwei Systemvariablen und zwei Systemroutinen dargestellt. Insgesamt kann eine Erweiterung folgende Elemente besitzen:

1. Lesbare und nicht beschreibbare Systemvariablen (Konstante, `$CURRENT_ROBOT_POSITION`)
2. Lesbare und beschreibbare Systemvariablen (`$CURRENT_ROBOT_PLACE`)
3. Systemroutinen ohne Rückgabewerte (`$GET_GEOMETRY`)
4. Systemroutinen mit Rückgabewerten (`$VARIATION`)

Der Kern ruft die Implementierung einer Konstante in der Funktionserweiterung auf, wenn im Plan auf die Konstante lesend zugegriffen wird. Die Implementierung der Konstante `$CURRENT_ROBOT_POSITION` ruft zunächst den Dienst *Abfrage Variablenwert* der Erweiterungsschnittstelle auf, um den zu der Konstante zugehörigen Variablenwert zu erhalten und belegt diesen mit der aktuellen Roboterposition aus dem Simulationsmodell. Letzteres liest die Implementierung aus dem Simulationsmodell über einen Dienst der Simulationsschicht aus.

Schnittstellenbeschreibung

```
-- Systemvariable (lesend, Konstante)
$POSITION CONST $CURRENT_ROBOT_POSITION;
-- Systemvariable (lesend und schreibend)
$FRAME $CURRENT_ROBOT_PLACE;
-- Systemroutine
$GET_GEOMETRY(CONST $OBJECT NAME => $VECTOR[][] GEOMETRIE);
-- Systemroutine mit Rückgabewert (Funktion)
$BOOL $VARIATION(=>$NUMBER[] PARAM);
```

Funktionserweiterung

Implementierung lesender Zugriff von Variable $CURRENT_ROBOT_POSITION

1. Abfrage Variablenwert $CURRENT_ROBOT_POSITION
2. Lesen der aktuellen Roboterposition aus dem Simulationsmodell
3. Schreibe Position in den Variablenwert $CURRENT_ROBOT_POSITION

Implementierung schreibender Zugriff von Variable $CURRENT_ROBOT_PLACE

1. Abfrage Variablenwert $CURRENT_ROBOT_PLACE
2. Lesen des Roboterstandorts aus dem Variablenwert
2. Ändern des Roboterstandorts im Simulationsmodell

Implementierung lesender Zugriff von Variable $CURRENT_ROBOT_PLACE

1. Abfrage Variablenwert $CURRENT_ROBOT_PLACE
2. Lesen des Roboterstandorts aus dem Simulationsmodell
3. Schreibe den Roboterstandort in Variablenwert von $CURRENT_ROBOT_PLACE

Implementierung der Routine $GET_GEOMETRY

1. Abfrage Variablenwert NAME, SURFACE
2. Abfrage der Geometriedaten des Objektes NAME aus Simulationsmodell
3. Schreibe Geometriedaten in Variablenwert von SURFACE

Implementierung der Routine $VARIATION

1. Abfrage Variablenwert PARAM[]
2. Schreibe neue Zellenänderung gemäß Formel 5.3 in Variablenwert PARAM[]
3. Setze Rückgabewert, ob weitere Zellenänderung gemäß Formel 5.3 erzeugbar sind

Bild 6.13: Beispielhafte Funktionserweiterung mit Schnittstellenbeschreibung

Anders als bei Konstanten erfordert eine Systemvariable zwei Implementierungen in einer Funktionserweiterung: eine Implementierung für den lesenden Zugriff (z. B. in einem Ausdruck) und eine Implementierung für den schreibenden Zugriff (z. B.auf der linken Seite der Zuweisung). Beide Implementierungen müssen zunächst den Variablenwert über den Dienst *Abfrage Variablenwert* der Erweiterungsschnittstelle beim Kern abfragen. Die Implementierung des lesenden Zugriffs der Systemvariablen `$CURRENT_ROBOT_POSITION` liest anschließend den Roboterstandort aus dem Simulationsmodell und schreibt ihn in den Variablenwert. Im Gegensatz dazu verändert die Implementierung des schreibenden Zugriffs den Roboterstandort im Simulationsmodell gemäß dem Variablenwert.

Für Routinen ist jeweils eine Implementierung in der Funktionserweiterung erforderlich. Die Implementierung von Systemroutinen mit Rückgabewert (z. B. `$VARIATION`) müssen an deren Ende den Rückgabewert dem Kern über den Dienst *Setzen Rückgabewert* der Erweiterungsschnittstelle bekannt geben.

6.2.5 Basisfunktionen

Die im Verlauf dieser Arbeit entstandenen Funktionserweiterungen (Bild 4.5, S. 43) werden in diesem Abschnitt kurz vorgestellt. Deren Systemroutinen und Systemvariablen legen den in Anforderung 2 (S. 30) geforderten Umfang an Basisfunktionen fest. Die Deklarationen aus den zugehörigen Schnittstellenbeschreibungen, mit einer kurzen Erläuterung über die erbrachte Funktion, sind in Anhang B (S. 169ff) abgedruckt. Die Funktionserweiterungen lauten wie folgt:

1. *Basis*: In dieser Erweiterung sind alle grundlegenden Funktionen enthalten, z. B. Operationen und Zugriffsfunktionen auf Felder, Erzeugung von zusammengesetzten Elementen (z. B. Vektoren, Positionen) und grundlegende mathematische Funktionen (z. B. Sinus, Cosinus).

2. *Parameter*: In dieser Erweiterung sind Funktionen für den Zugriff auf den Eingabeassistenten enthalten, z. B. Parameteranforderung eines oder mehrerer Parameter.

3. *Optimierung*: Diese Erweiterung enthält Optimierungsverfahren, z. B. Abstiegsverfahren zur Parameteroptimierung, Lösungsverfahren für das Rundreiseproblem.

4. *Datei*: Mit dieser Erweiterung ist die Verwendung von maschinenlesbaren Daten aus einer Datei, z. B. Funktionen zur Dateiverwaltung, Funktionen zur Manipulation von Zeichenketten.

5. *Geometrie*: Diese Erweiterung enthält Funktionen mit geometrischen Algorithmen zur Berechnung von zwei- und dreidimensionalen geometrischen Elementen, z. B. Berechnung von Schnittpunkten beim Schnitt zwischen Gerade und Polyeder, Strecken und Stauchen von Polygonen, Triangulierungsverfahren von Polyedern.

6. *Synthese*: Diese Erweiterung enthält Funktionen zum Versenden von Programm- und Steuerbefehlen an das Modul Programmsynthese. Damit ist einerseits die Festlegung des Programmsinhalts und die Programmsyntax möglich. Andererseits kann damit die Programmsynthse selbst gesteuert werden. Dies wird im nächsten Kapitel genauer betrachtet.

7. *Simulation*: Diese Erweiterung enthält Funktionen zur Ausführung des Simulationslaufs, z. B. Starten und Stoppen des Simulationslaufs, Laden von Programmen in die virtuelle Robotersteuerung, Lesen der Ausführungszeit des zuletzt durchgeführten Laufs.

8. *Modell*: Diese Erweiterung enthält Funktionen zum Lesen und Schreiben des Simulationsmodells, z. B. Erzeugen, Löschen und Verschieben von Objekten, Greif- und Greiferpunkten, Lesen der Geometrie- und Roboterdaten.

9. *Visualisierung*: Diese Erweiterung dient zum Ein- und Ausblenden von geometrischen Zwischenergebnissen während der Planausführung, z. B. Schnittebenen, Geraden oder Koordinatensysteme.

Die vier letztgenannten Erweiterungen rufen Dienste der unterlagerten Simulationsschicht zur Erbringung ihrer Funktion auf. Sie sind damit abhängig vom unterlagerten CAR-System COSIMIR®. Bei den anderen Funktionserweiterungen besteht diese Abhängigkeit nicht.

7 Synthese der Roboterprogramme

Dieses Kapitel beschreibt den Aufbau und den Befehlsumfang der generierbaren Roboterprogramme. Dazu werden die Anforderungen 4-10 (S. 31ff) berücksichtigt. Das Kapitel beschreibt anschließend die in dieser Arbeit entstandene Lösung zur Trennung der Informationen über den Programminhalt (Programmspezifikation) von der Beschreibung der konkreten Syntax einer Industrierobotersteuerung (Syntaxdefinition). Des Weiteren wird die Frage nach der Zusammenführung dieser Informationen im Modul Programmsynthese beantwortet.

7.1 Funktionen der Industrierobotersteuerungen

Das System muss aus derselben steuerungsneutralen Programmspezifikation inhaltlich äquivalente, steuerungsspezifische Roboterprogramme (Anforderung 4, S. 31) für unterschiedliche Steuerungen erzeugen können (Anforderung 5, S. 31). Aufgrund dieser Anforderungen ist eine Untermenge an gemeinsamen Roboterbefehlen erforderlich, die möglichst alle industriellen Robotersteuerungen verarbeiten können. Diese Untermenge soll möglichst groß sein, damit der CAR-Anwender bei der Angabe der Roboterbefehle in den Generierungsplänen möglichst viele Funktionen heutiger Robotersteuerungen verwenden kann.

Um die größtmögliche Untermenge zu finden, wurden heutiger Industrierobotersteuerungen untersucht. Aus dieser Untersuchung wird danach abgeleitet, welche Roboterbefehle dem CAR-Anwender für die steuerungsneutrale Programmspezifikation zur Verfügung stehen.

7.1.1 Auswahl

Für die Untersuchung wurde eine Auswahl heutiger Industrierobotersteuerungen getroffen (Tabelle 7.1). Diese Auswahl enthält die von COSIMIR® unterstützten Sprachen [116] der führenden europäischen Roboterhersteller [118, 88]. Die Sprachen sind von unterschiedlichem Charakter. In der Auswahl finden sich Pascal-ähnliche Sprachen (KUKA KRL [92], ABB Rapid [6]), eine BASIC-ähnliche Sprache (Mitsubishi MELFA BASIC IV [106]) und Sprachen mit einem proprietären Stil (Bosch BAPS [27], Adept V+ [8]).

Tabelle 7.1: Analysierte Steuerungen mit deren Programmiersprachen

Hersteller	KUKA	ABB	Bosch	Adept	Mitsubishi
Steuerung	KR C1/2	S4	rho4	Adept	CR2
Sprache	KRL	Rapid	BAPS	V+	MELFA BASIC IV

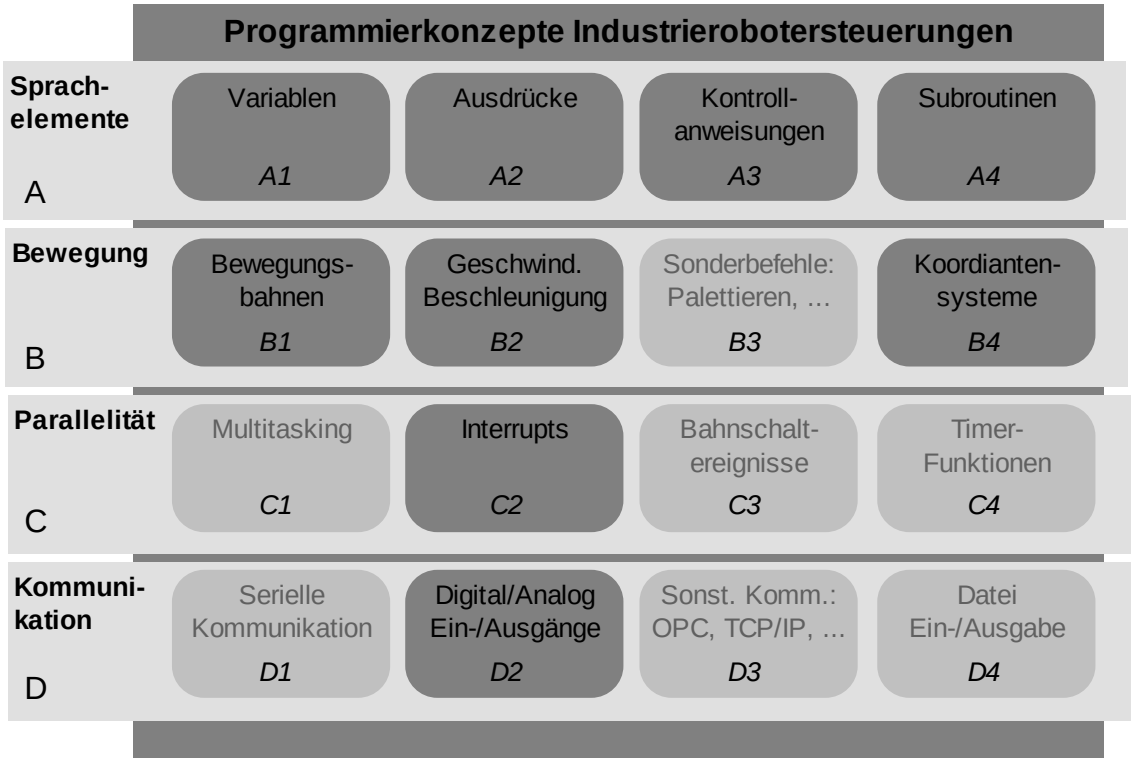


Bild 7.1: Einteilung der Programmierkonzepte heutiger Industrierobotersteuerungen

7.1.2 Programmierkonzepte

Da es viele Programmierkonzepte der untersuchten Industrierobotersteuerungen gibt, erfolgt zunächst eine Einteilung der Programmierkonzepte in vier Hauptkategorien A-D mit jeweils vier Unterkategorien 1-4 (Bild 7.1), die jeweils zusammengehörende Programmierkonzepte zusammenfassen.

Die Hauptkategorie A (Sprachelemente) enthält die Konzepte, die aus höheren Programmiersprachen bekannt sind. Dazu gehören Variablen, Ausdrücke, Kontrollanweisungen und Subroutinen.

Die Hauptkategorie B (Bewegung) enthält die Konzepte für die Programmierung der Roboterbewegungen. Dazu gehören die Vorgaben der Interpolationsart (z. B. PTP, linear, zirkular), der Geschwindigkeit und der Beschleunigung für die jeweilige Interpolationsart und die Möglichkeit zur Veränderung spezifischer Koordinatensysteme (z. B. Roboterbasis, Tool Center Point, Werkstück). Manche Steuerungen unterstützen außerdem die Roboterprogrammierung für ein bestimmtes Einsatzgebiet (z. B. Palettierbefehle der Mitsubishi Steuerung CR2).

Die Hauptkategorie C (Parallelität) enthält die Konzepte, die eine parallele Verarbeitung auf der Steuerung erfordern. Dazu zählen die parallele Verarbeitung von Roboterprogrammen (Multitasking), die Möglichkeit zur Definition von Unterbrechungen (Interrupts), Timer-Funktionen (Anstoßen einer Verarbeitung nach einer definierten Zeit) und Bahnschaltereignisse (Anstoßen einer Verarbeitung aufgrund von erfüllten Bewegungseigenschaften). Ein Beispiel für Letzteres ist das Setzen eines Ausgangs in einem vorgegebenen Abstand (räumlich, zeitlich) vor Erreichen einer Roboterzielstellung.

Die Hauptkategorie D (Kommunikation) enthält alle Konzepte für die Programmierung eines Datenaustauschs mit externen Geräten. Das wichtigste Konzept in dieser Kategorie ist, da es in der Praxis

Bild 7.2: Untermenge gemeinsamer Steuerungsfunktionen

am häufigsten vorkommt, die Kommunikation über digitale und analoge Ein-/Ausgänge. Daneben unterstützen Robotersteuerungen weitere Kommunikationsmechanismen wie die serielle Kommunikation, Kommunikation über Dateien oder sonstige Kommunikationsmechanismen (z. B. OPC, CAN-Bus, Profibus, Interbus-S, ArcNet, TCP/IP).

7.1.3 Ergebnis

Als Ergebnis der Untersuchung zeigt sich, dass die dunkel markierten Kategorien aus Bild 7.1 Funktionen zur gemeinsamen Untermenge beisteuern. Funktionen aus den hell markierten Kategorien konnten nicht in die gemeinsame Untermenge aufgenommen werden, weil sich deren Realisierung auf den Steuerungen als zu unterschiedlich herausgestellt hat. Damit die generierten Programme trotzdem diese Funktionen verwenden können, müssen manuell erstellte Programme die zugehörigen Befehle kapseln. Die generierten Programmen rufen zur Verwendung der Funktionen aus den hell markierten Kategorien die manuell erstellten Programme auf (Anforderung 7, S. 32).

In Bild 7.2 ist die Untermenge gemeinsamer Steuerungsfunktionen zusammengefasst. Die Steuerungen besitzen globale Variablen und globale eindimensionale Felder, die von einem vordefinierten Datentypen sind. Als Datentypen kommen dabei reelle Zahlen und Zeichenketten vor. Zusätzlich besitzen die Steuerungen einen Datentyp, der Angaben zu einer kartesischen Roboterposition enthält, und einen Datentypen, indem Achskoordinaten einer Roboterstellung abgelegt sind.

Die Steuerungen erlauben arithmetische und logische Ausdrücke, Vergleiche und besitzen alle die Möglichkeit zur Berechnung der homogenen Transformation. Als Kontrollanweisungen stellen die

Steuerungen die bedingte Verzweigung, bedingte Schleifen, die Zählschleife und die Zuweisung zur Verfügung. Des Weiteren ist es möglich, dass sich Programme gegenseitig aufrufen. Die betrachteten Steuerungen besitzen alle einen Befehl, um die Programmausführung eine gewisse Zeitspanne warten zu lassen. Subroutinen sind ebenfalls auf allen Steuerungen programmierbar. Allerdings ist bei der Steuerung Mitsubishi CR2 eine Parameterübergabe für Subroutinen nicht möglich.

Zur Bewegungsprogrammierung bieten alle Steuerungen die Interpolationsart PTP, die lineare und zirkulare Interpolationsart an. Auch überschliffene Bewegungen sind möglich. In Abhängigkeit der Interpolationsart können Beschleunigung und Geschwindigkeit angegeben werden. Des Weiteren sind Bezugskoordinatensystem für die Bewegungen (Roboterbasis, Werkzeug) änderbar.

Die betrachteten Steuerungen erlauben die Programmierung von Interrupts. Diese kommen insbesondere in Notsituationen (z. B. Drücken des Not-Aus-Schalters) zur Anwendung. Dazu sind eine Interrupt-Definition und eine Routine zur Interrupt-Behandlung erforderlich. Die Steuerungen bieten die Möglichkeit, Interrupts mit unterschiedlichen Prioritätswerten zu versehen.

Bei der Kommunikation mit externen Geräten konnte ausschließlich die Kommunikation über digitale E/A in die gemeinsame Untermenge aufgenommen werden. Bei den andern Kommunikationsmöglichkeiten hinderten insbesondere einzuhaltende Protokolle und Textformatierungen die Aufnahme in die Untermenge gemeinsamer Steuerungsfunktionen.

7.2 Steuerungsneutrale Programmspezifikation

Aufgrund der festgelegten Untermenge gemeinsamer Steuerungsfunktionen wird im Folgenden die Leistungsfähigkeit der generierbaren Roboterprogramme festgelegt. Anschließend wird dargestellt, wie die damit verbundene Programmspezifikation in die Generierungspläne integriert ist. Die Schwierigkeit bei der Definition der Leistungsfähigkeit generierbarer Roboterprogramme liegt in der Erfüllung der Anforderungen 6 und 7 (S. 32ff), insbesondere im Datenaustausch zwischen generierten und manuell erstellten Roboterprogrammen.

7.2.1 Struktur generierbarer Roboterprogramme

Zur Lösung dieser Schwierigkeit müssen die beiden folgenden Teilprobleme gelöst werden: Aufbau und Variablen der generierbaren Roboterprogramme.

Aufbau der generierbaren Roboterprogramme

Die Programmsynthese kann mehrere Roboterprogramme erzeugen, die zu einem Programmprojekt zusammengefasst sind (Bild 7.3). Diese Programme können sich gegenseitig aufrufen. Das Hauptprogramm kennzeichnet dabei den Beginn der Ausführung. Neben den Programmen wird auch eine Liste von globalen Variablen mit Initialisierungswerten erzeugt. Auf diese Variablen können alle

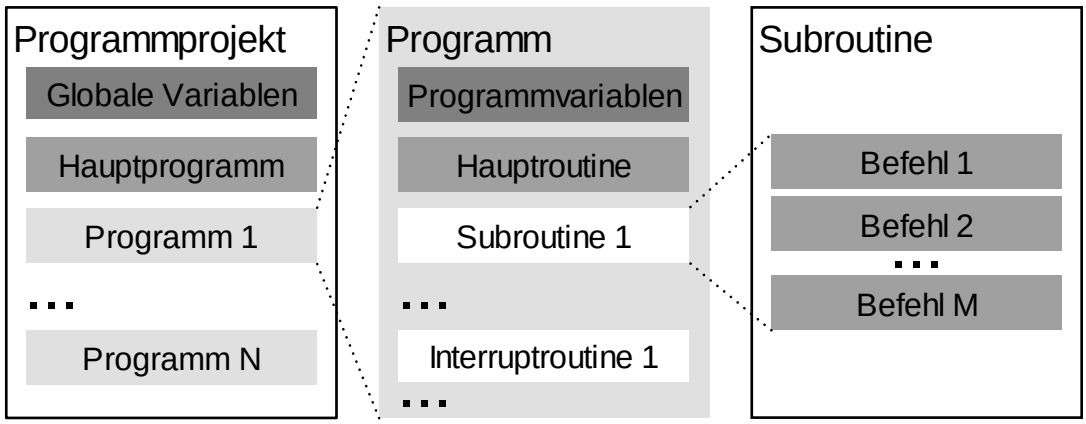


Bild 7.3: Aufbau generierbarer Roboterprogramme

Programme der Steuerung, auch manuell erstellte Programme, lesend und schreibend zugreifen. Ein Programm besteht aus genau einer Hauptroutine und aus Subroutinen, Interruptroutinen und Programmvariablen. Letztere sind nur für die Routinen in einem Programm gültig. Die Hauptroutine kennzeichnet den Bearbeitungsanfang des Programmaufrufs.

Insgesamt wird auf eine Parameterübergabe zwischen Programmen und zwischen einzelnen Routinen verzichtet. Die Grund dafür ist, dass jede Robotersteuerung eigene Konventionen und Möglichkeiten bei der Typkonvertierung besitzt. Der Datenaustausch zwischen Programmen und Routinen wird deshalb ausschließlich über Variablen durchgeführt. Die globalen Variablen dienen dem Datenaustausch zwischen Programmen. Die Programmvariablen dienen dem Datenaustausch zwischen Routinen. Es wird ebenso wird auf die Definition von lokalen Variablen in Subroutinen verzichtet, weil dieses Konzept nicht von allen betrachteten Steuerungen unterstützt wird (z. B. Mitsubishi CR2).

Die Möglichkeit zur Erzeugung von Subroutinen in einem Programm steht zur Verfügung, um wiederkehrende Befehlssequenzen zusammenzufassen. Der Verzicht auf Subroutinen hätte zur Folge, dass man dieselbe Befehlssequenz mehrmals hintereinander erzeugen müsste, was zu unnötig langen Programmen führt. Die Unterstützung von Interruptroutinen macht es möglich, in einem generierten Roboterprogramm auf Notsituationen (z. B. Drücken des Not-Aus-Schalters) zu reagieren.

Variablen in den generierbaren Roboterprogrammen

Der CAR-Anwender kann Programme spezifizieren, die Variablen und eindimensionale Felder (im Folgenden nur als Variable bezeichnet) vom Typ Zahl, Zeichenkette, kartesische und achsspezifische Roboterkoordinaten enthalten. Ein generiertes Programm unterscheidet die Variablenarten:

1. Globale Variable
2. Externe Variable
3. Explizite Programmvariable
4. Implizite Programmvariable

Die *globalen Variablen* sind für alle Programme auf der Steuerung gültig und werden im Programmprojekt angelegt und initialisiert. Dagegen ist eine *externe Variable* eine globale Variable, die ein manuell erstelltes Programm anlegt und initialisiert. Generierte Programme können externe Variablen verwenden, aber sie enthalten nicht deren Deklaration und Initialisierung. Programmvariablen sind nur in dem Roboterprogramm gültig, in dem sie deklariert sind. Ein generiertes Programm unterscheidet zwischen *expliziten* und *impliziten Programmvariablen*, je nachdem, ob der CAR-Anwender deren Name explizit vorgibt oder die Programmsynthese den Variablennamen automatisch zuweist.

Bei den ersten drei Variablenarten muss der CAR-Anwender den Variablennamen festlegen, damit sie in einem generierten oder in einem manuell erstellten Roboterprogramm identifizierbar sind. Ohne diese Identifizierbarkeit ist die Spezifikation von Programmen nicht möglich, die Informationen verarbeiten, die erst zur Laufzeit des Roboterprogramms verfügbar sind. Die Identifizierbarkeit ist in den Polierprogrammen aus dem einleitenden Beispiel (Bild 1.1, S. 2) nötig, wenn eine speicherprogrammierbare Steuerung (SPS) oder ein Maschinenführer der Robotersteuerung erst unmittelbar vor der Programmausführung mitteilt, wie oft der Roboter den Rand einer Produktvariante polieren soll.

7.2.2 Programmspezifikation in den Generierungsplänen

Die Spezifikation der zu generierenden Programme erfolgt im Generierungsplan. Für die Programmspezifikation stehen dem CAR-Anwender vordefinierte Plananweisungen aus der Funktionserweiterung *Synthese* zur Verfügung (Bild 7.4). Die Integration der Programmspezifikation in die Pläne hat den Vorteil, dass alle Angaben, die für die Programmgenerierung erforderlich sind, in einem Plan zusammengefasst sind. Die Integration der Programmspezifikation in die Pläne erfordert die Lösung folgender Probleme: Abbildung der Struktur (Aufbau, Variablen) generierbarer Roboterprogramme in die Pläne und die Auswahl bereitgestellter Roboterbefehle.

Abbildung von Planvariablen in das Roboterprogramm

Für die Deklaration einer Variable oder eines eindimensionalen Felds (Variablenart 1-3, S. 101), die jeweils vom CAR-Anwender einen Namen erfordert, gibt man eine entsprechende Planvariable an. Erlaubt sind Variablen der Typen `$NUMBER`, `$TEXT`, `$POSITION`, `$AXIS`, weil allen betrachteten Steuerungen diese Typen unterstützten. Aus dieser Planvariable erzeugt das System eine Variable im Programmprojekt mit möglichst demselben Namen (unter Einhaltung von Namenskonventionen der jeweiligen Sprache), mit demselben Datentyp und, falls nötig, mit demselben Variablenwert zur Initialisierung. Durch die zugehörigen Plananweisungen in Bild 7.4 fügt die Programmsynthese dem Roboterprogrammprojekt z. B. eine globale Variable A, eine externe Variable B und eine explizite Programmvariable C mit deren ursprünglichen Datentypen hinzu. Die Variablen A, B oder C können in einem Plan als planlokale oder routinenlokale Variable deklariert sein. Diese Variablendeklarationen müssen nicht in der Programmspezifikation erfolgen. Bei Verwendung dieser Variablen als Parameter in einer Plananweisung der Programmspezifikation tauchen sie in der zugehörigen steuerungsspezifischen Befehlssequenz im Roboterprogramm ebenfalls als Parameter auf.

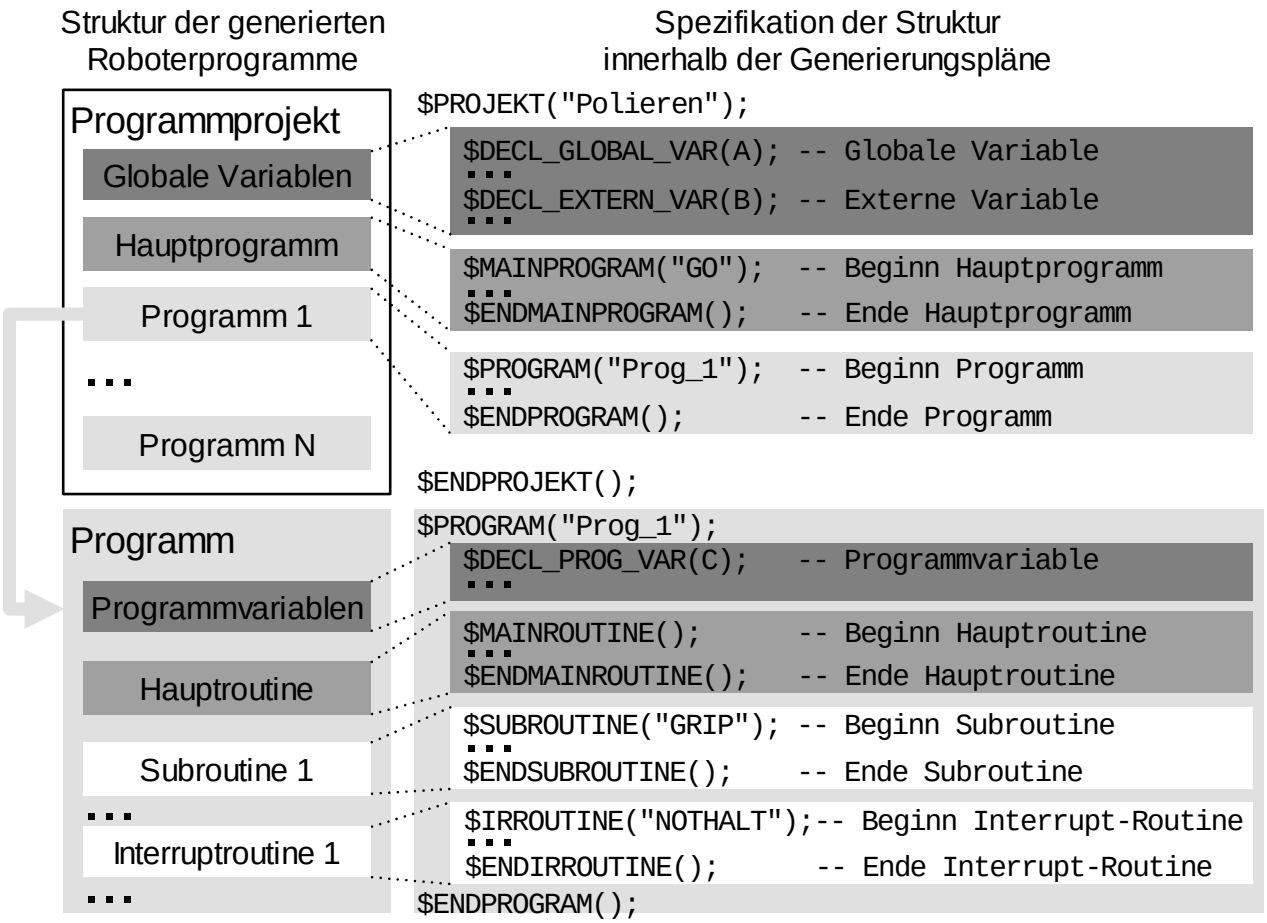


Bild 7.4: Spezifikation der Struktur der generierbaren Roboterprogramme in den Plänen

Die Vorteile, die durch diese Abbildung von Planvariablen in das generierte Roboterprogramm ergeben, bestehen einerseits darin, den Bezug zwischen Plan und generiertem Roboterprogramm zu verdeutlichen. Andererseits wird ein weiteres spezifisches Variablenkonzept für die generierbaren Roboterprogramme vermieden. Dieses Variablenkonzept müsste man als weiteres Sprachkonzept in die Pläne aufnehmen. Für dieses spezifische Variablenkonzept wären weitere Plananweisungen zur Definition und zur Übertragung der Inhalte aus den Planvariablen notwendig. Dieser Zusatzaufwand wird durch die Verwendung von Planvariablen in der Programmspezifikation eingespart.

Auswahl der Roboterbefehle

Die Roboterbefehle, die der CAR-Anwender zur Programmspezifikation verwenden kann, ergeben sich auf der durchgeführten Untersuchung der Industrierobotersteuerungen. Einen Auszug dieser Roboterbefehle zeigt Tabelle 7.2. Für die Kategorien der Untermenge gemeinsamer Steuerungsfunktionen aus Bild 7.2 sind jeweils entsprechende Roboterbefehle angegeben. Die Plananweisungen `$DECL_GLOBAL_VAR` und `$DECL_PROG_VAR` stellen keine Roboterbefehle dar, sie sind aber zur Vervollständigung aufgeführt.

Eine steuerungsübergreifende Abbildung von vollständigen Ausdrücken (Kategorie A2) erweist sich als schwierig, weil man zahlreiche Faktoren wie Typprüfung, Schachtelungstiefe und Typkonver-

Tabelle 7.2: Auszug unterstützter Programmbefehle

Kat.	Befehl der Programmspezifikation	Erklärung
A1	`$DECL_GLOBAL_VAR(VOID NAME)`	Deklaration globale Variable
A1	`$DECL_PROG_VAR(VOID NAME)`	Deklaration expl. Programmvariable
A2	`$GT($NUMBER A,B =>$NUMBER C)`	Vergleich (größer als)
A2	`$ADD($NUMBER A,B =>$NUMBER C)`	Addition
A3	`$IF($NUMBER NR);...$THEN();...$ENDIF()`	Bedingte Verzweigung
A3	`$TIMES($NUMBER N);...$ENDTIMES()`	Wiederholung einer Befehlssequenz
A3	`$CALL_PROGRAM($TEXT NAME)`	Aufruf Programm
A3	`$CALL_SUBROUTINE($TEXT NAME)`	Aufruf Subroutine
A3	`$WAIT_TIME($NUMBER SEC)`	Warten (Sekunden)
A4	`$SUBROUTINE($TEXT NAME)`	Subroutine
B1	`$PTP($POSITION NAME)`	PTP-Bewegung
B1	`$LINEAR($POSITION NAME)`	Linearbewegung
B1	`$PATH($POSITION[] NAME)`	Überschliffene Bewegung
B2	`$PTP_SPEED($NUMBER PERCENT)`	PTP-Geschwindigkeit (in %)
B2	`$PATH_SPEED($NUMBER MM_S)`	Bahngeschwindigkeit (in $\frac{mm}{s}$)
B2	`$PTP_ACCELERATION($NUMBER PERCENT)`	PTP-Beschleungigung (in %)
B2	`$PATH_ACCELERATION($NUMBER MM_S2)`	Bahnbeschleunigung (in $\frac{mm}{s^2}$)
B4	`$BASE($POSITION ROBOT)`	Setzen der Roboterbasis
B4	`$TOOL($POSITION ROBOT)`	Setzen des TCP
C2	`$INTERRUPT($NUMBER IN, $TEXT ROUTINE, $NUMBER PRIO, $BOOL FLANK)`	Definition Interrupt bei bei Eingangsflanke (0/1, 1/0) mit Priorität
D2	`$SET_DIG_OUTPUT($NUMBER NR)`	Setzen digitaler Ausgang
D2	`$RESET_DIG_OUTPUT($NUMBER NR)`	Rücksetzen digitaler Ausgang
D2	`$WAIT_UNTIL_INPUT_SET($NUMBER NR)`	Warten bis Eingang anliegt
-	`$COMMENT($TEXT LINE);`	Kommentarzeile

tierung beachten muss. Deshalb wird auf geschachtelte Ausdrücke verzichtet. Statt dessen werden elementare Ausdrücke auf jeweils eine Plananweisung abgebildet (z. B. `$GT`, `$ADD`) und tauchen im Roboterprogramm als Zuweisung, vergleichbar mit Dreiadressbefehlen, auf.

Für die Generierung von Kontrollstrukturen (A3) stehen dem CAR-Anwender bedingte Schleifen (`$WHILE`, `$REPEAT`) und die bedingte Verzeigung (`$IF`) zur Verfügung. Eine Generierung einer Zählschleife, wie sie in jeder analysierten Roboterprogrammiersprache auftaucht, erweist sich als schwierig. Dies liegt an der zugehörigen Zählvariablen. Zum einen muss diese Variable bei manchen Sprachen (Rapid, KRL und MELFA BASIC IV) vom Typ Integer sein. Dieser Typ steht aber in den Plänen nicht zur Verfügung, sondern nur der Typ reelle Zahl. Zum anderen muss bei manchen Sprachen die Zählvariable explizit deklariert werden (KRL, MELFA BASIC IV), bei anderen Sprachen (Rapid) darf diese Variable nicht deklariert werden, sie wird impliziert deklariert. Da die Zählschleife oftmals zur Wiederholung einer Befehlssequenz eingesetzt wird, wird die Schleife `$TIMES` zur Verfügung gestellt.

In der Kategorie B1 stehen die Interpolationsarten PTP, lineare, zirkulare und überschliffene Bewegung zur Verfügung. Dabei wird eine überschliffene Bewegung aus einzelnen Linearbahnen zusam-

Programmbefehl	$PATH_SPEED($NUMBER MM_S)	$SET_DIGITAL_OUTPUT($NUMBER NR)
KUKA KRL	$VEL.CP = #MM_S#/1000.0	SIGNAL #DO:NR# $OUT[#NR#] ... #DO:NR# = TRUE
ABB Rapid	PERS speeddata LinSpeed ... LinSpeed.v_tcp := 100;	SetDO #DO:NR#, 1;
Bosch BAPS	V = #MM_S#	AUSGANG BINAER: #NR#=#DO:NR#, ... #DO:NR#=1
Adept V+	SPEED #MM_S# MMPS ALWAYS	SIGNAL #NR#
Mitsubishi MELFA BASIC IV	SPD #MM_S#	M_OUT(#NR#) = 1

#MM_S#: Bahngeschwindigkeit #NR#: Ausgangsnummer #DO:NR#: Ausgangsname

Bild 7.5: Beispiel für Umsetzung der Roboterbefehle

mengesetzt und mit der Plananweisung $PATH angegeben. Für die jeweilige Interpolationsart ist die Geschwindigkeit und Beschleunigung änderbar (B2). In der Kategorie B4 sind das Werkzeug- und das Roboterbasiskoordinatensystem änderbar.

Kategorie C2 enthält Befehle in Zusammenhang mit Interrupts ($INTERRUPT). Dabei werden ausschließlich Interrupts zugelassen, die die Robotersteuerung bei Änderung von digitalen Eingangswerten auslöst. Die Verwendung von Interrupts für andere Einsatzgebiete, z. B. aufgrund einer Variablenänderung, ist zu eingeschränkt auf einzelne Steuerungen (z. B. KUKA KR C1).

In der Kategorie D2 stehen Befehle zum Setzen und Rücksetzen von digitalen Ausgängen zur Verfügung. Ebenso ist das Warten auf Eingangswerte zugelassen.

Als Beispiel zur Umsetzung der Programmbefehle stellt Bild 7.5 für die beiden Programmbefehle $PATH_SPEED (Setzen der Bahngeschwindigkeit) und $SET_DIGITAL_OUTPUT (Setzen eines digitalen Ausgangs) die Steuerungssyntax der Befehle für die betrachteten Steuerungen dar. Bemerkenswert ist, dass manche Programmbefehle Ergänzungen an verschiedenen Stellen im Roboterprogramm erfordern, z. B. ist die Voraussetzung für das Setzen eines digitalen Ausgangs bei BAPS oder KRL die Deklaration einer Variable, die dem Ausgang der Robotersteuerung entspricht.

7.3 Modul Programmsynthese

Die Aufgabe der Programmsynthese ist die Erzeugung steuerungsspezifischer Roboterprogramme aus der steuerungsneutralen Programmspezifikation. Dazu benötigt dieses Modul das Wissen über die Steuerungssyntax. Für die Umsetzung gibt es zwei Möglichkeiten (Bild 7.6).

Bei der ersten Möglichkeit enthält die Programmsynthese das Wissen über die spezifische Sprachsyntax. Bei der zweiten Möglichkeit ist dieses Wissen außerhalb der Programmsynthese in so genann-

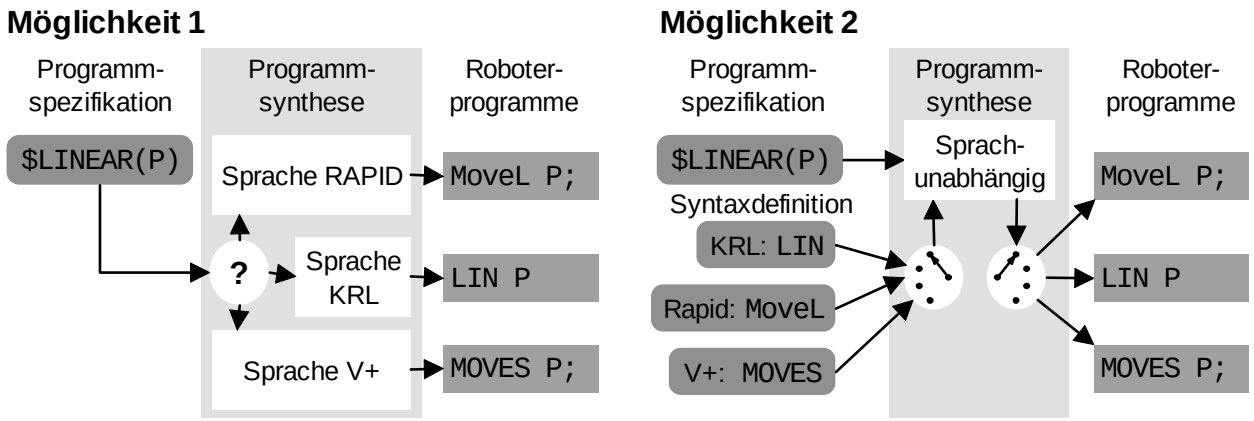


Bild 7.6: Möglichkeiten für den Entwurf des Moduls Programmsynthese

ten *Syntaxdefinitionen* enthalten. Dabei arbeitet die Programmsynthese vollständig steuerungsneutral. Der Vorteil der zweiten Möglichkeit besteht darin, dass Änderungen der Steuerungssyntax bei neuen Steuerungsversionen nicht zu einer Änderung der Programmsynthese führen. Etwaige Änderungen an der Steuerungssyntax kann man mit geringem Aufwand durch einen Eintrag in die Syntaxdefinition berücksichtigen. Ein weiterer Vorteil ist der geringe Aufwand für die Ergänzung neuer Roboterbefehle, weil diese ebenso keine Änderungen an der Programmsynthese erfordern. Aus diesen Gründen und aufgrund von Anforderung 10 (S. 34) scheidet die erste Möglichkeit aus.

Bei der Umsetzung der zweiten Möglichkeit müssen folgende Bereiche berücksichtigt werden:

1. Entwurf einer neutralen, modulinternen Repräsentation der zu generierenden Programme
2. Struktur und Inhalt der Syntaxdefinitionsdateien
3. Entwurf einer steuerungsneutralen Architektur der Programmsynthese

7.3.1 Interne Repräsentation der Roboterprogramme

Die Programmsynthese erhält den Inhalt der Programmspezifikation aus den Programmbefehlen, die die Generierungssteuerung ihr sendet (Bild 4.7, S. 46). Die Summe aller empfangenen Programmbefehle bildet den Inhalt des zu generierenden Programms. Beim Empfang jedes einzelnen Programmbefehls trägt die Programmsynthese den Inhalt an einer oder mehreren Stellen in der internen Repräsentation ein. Für z. B. das Setzen der digitalen Ausgänge in BAPS oder KRL (Bild 7.5) ist neben dem eigentlichen Befehl zum Setzen, die Deklaration einer Ausgangsvariablen erforderlich. Erst nachdem die interne Repräsentation vollständig gefüllt ist, gibt die Programmsynthese das Programm aus.

Die steuerungsneutrale interne Repräsentation der generierbaren Roboterprogramme besteht aus unterschiedlichen Containern (Bild 7.7): einem Container für die globalen und externen Variablen und jeweils einem Container für jedes zu generierende Programm. Jeder Container enthält Fächer und jedes Fach enthält Elemente. Ein Fach z. B. im Container für das Hauptprogramm ist das Fach, das alle Deklarationen (Elemente) von digitalen Ausgängen enthält. Ein Element wäre dann die Deklaration

eines digitalen Ausgangs. Die Anweisungen aus den Programmbefehlebn sind in den Elementen der Fächer Hauptroutine, Subroutinen und Interruptroutinen der Programmcontainer abgelegt.

Tritt z. B. in der Spezifikation der Hauptroutine des Hauptprogramms der Befehl `$SET_DIGITAL_OUTPUT` zum Setzen eines digitalen Ausgangs auf, so nimmt die Programmsynthese zwei Eintragungen in der internen Repräsentation vor. Erstens trägt sie ein neues Element im Fach ”Deklaration digitaler Ausgang” mit Namen und Nummer des Ausgangs ein, wobei der Ausgangsname vom Roboter aus dem Simulationsmodell stammt. Zweitens fügt sie eine Anweisung zum Setzen des Ausgangs am Ende der Anweisungen der Hauptroutine im Container des Hauptprogramms an.

Auf diese Weise verteilt die Programmsynthese die in jedem Befehl enthaltene Information auf unterschiedliche Fächer. Der Vorteil, der sich durch den Aufbau dieser Repräsentation ergibt, ist die Verteilung der Informationen und deren anschließende Verwendung durch die Syntaxdefinitionen der jeweiligen Robotersptachen.

Ein Beispiel, wie die Programmsynthese die interne Repräsentation der Roboterprogramme aufbaut, zeigt Bild 7.8. Die darin enthaltene Programmspezifikation legt eine Programmvariable `AUFNAHME` an und enthält drei Roboterbefehle in ihrer Hauptroutine. Der Roboterbefehl zum Setzen des Ausgangs bewirkt zwei Eintragungen in der Repräsentation. Die drei Roboterbefehle trägt die Programmsynthese in das Fach Hauptroutine des Containers des Hauptprogramms ein.

7.3.2 Struktur und Inhalt der Syntaxdefinition

Eine Syntaxdefinition enthält die Informationen über die Syntax einer Robotersteuerung. Für jede Robotersteuerung existiert mindestens eine Syntaxdefinitionsdatei. Dabei können mehrere Syntaxdefinitionen für eine Steuerung notwendig sein, wenn sie unterschiedliche Roboter (z. B. Zahl der Roboterachsen) unterstützt. In diesem Fall ändern sich u. a. der Achsdatentyp und die Befehle zur Angabe der Achsgeschwindigkeiten und der Achsbeschleunigungen. Eine Syntaxdefinition besteht aus drei Teilen (Bild 7.9):

1. Globale Einstellungen
2. Aufbauaktionen
3. Syntaxaktionen

Globale Einstellungen

Die *globalen Einstellungen* legen die grundlegenden Eigenschaften der Steuerungssyntax fest. Zu diesen Einstellungen gehört u. a. die Angabe, ob die Steuerungssyntax eine Zeilennummerierung benötigt (`LINE_NUMBER`, Zeile 1), die maximale Länge von globalen Variablennamen (`GLOBAL_VAR_MAX_LENGTH`, Zeile 2), die Anzahl der Nachkommastellen für Positionsangaben (`DECIMAL_PLACES_POSITION`, Zeile 3), die Kennzeichnung eines Kommentars (`COMMENT_BEGIN`, Zeile 4),

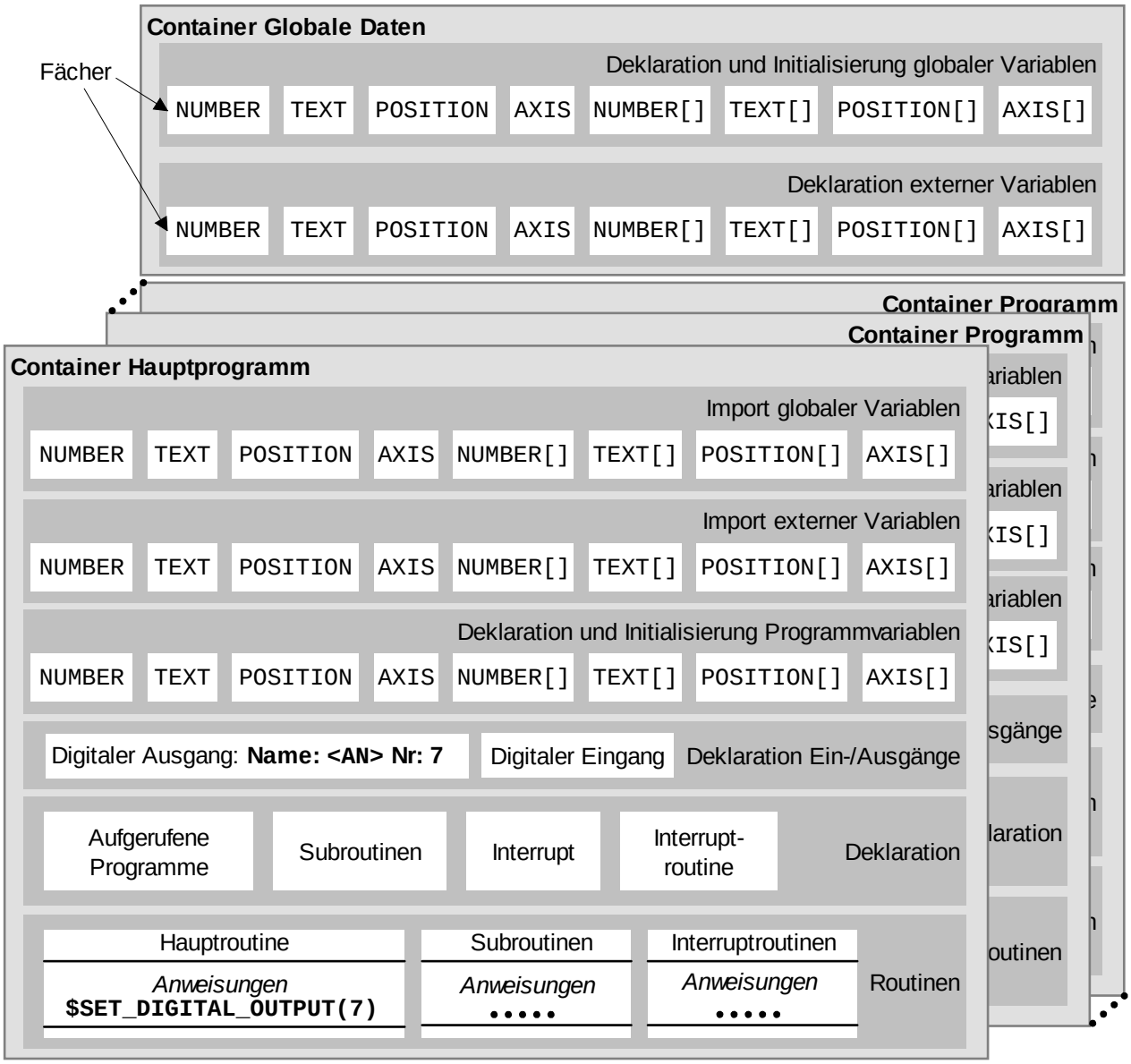


Bild 7.7: Repräsentation der generierbaren Roboterprogramme im Modul *Programmsynthese*

der Index (`ARRAY_START_INDEX`, Zeile 5), bei dem ein Array beginnt und das Präfix für implizite Programmvariablen des Typs `$NUMBER` (`PREFIX_IMPLICIT_PROG_VAR_NAME_NUMBER`, Zeile 6).

Des Weiteren enthält eine Syntaxdefinition Aufbauaktionen und Syntaxaktionen. Jede Aktion besitzt einen Namen, der in eckigen Klammern angegeben ist.

Syntaxaktion

Eine *Syntaxaktion* (Bild 7.9) enthält die Syntax in Form von Programmzeilen (`LINE`). Sie gibt die Syntax für genau einen Ausschnitt aus dem Roboterprogramm an, z. B. Deklaration einer Ausgangsvariablen (`DECL-OUTPUT-DIGITAL`, Zeile 28), die Deklaration einer Programmvariablen (`DECL-PROGRAM-VARIABLE-POSITION`, Zeile 30) des Typs `$POSITION` und die Initialisierung dieser Programmvariablen (`INIT-PROGRAM-VARIABLE-POSITION`, Zeile 32). Eine Syntaxaktion bezieht ihre

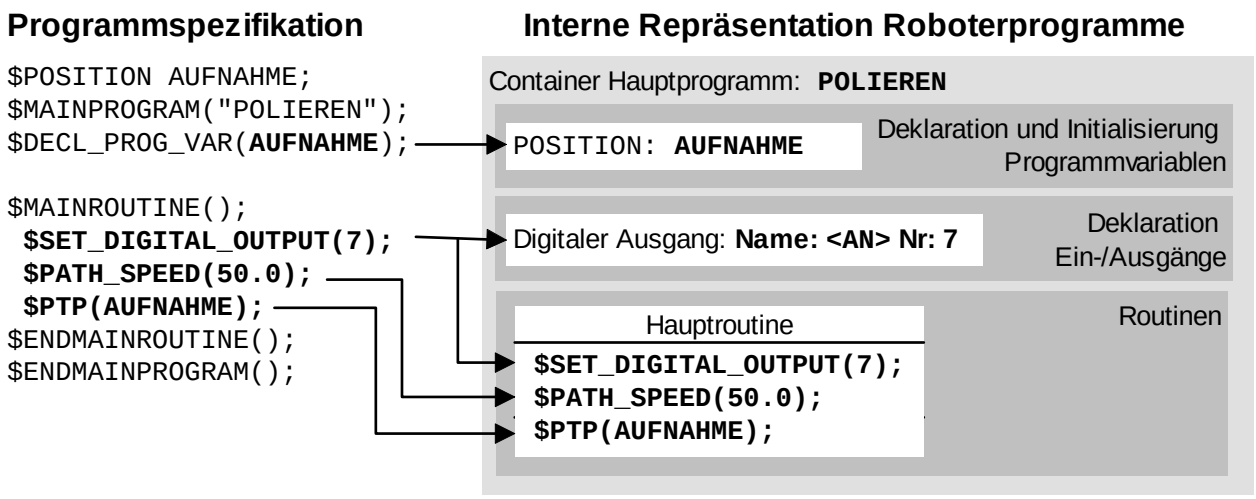


Bild 7.8: Beispiel zum Aufbau der internen Repräsentation

Informationen aus einem Fach der internen Repräsentation der Roboterprogramme (Bild 7.7) und wird für alle Elemente des Fachs angewendet. Die Zuordnung, aus welchem Fach die Syntaxaktion ihre Informationen bezieht ist vordefiniert und der Programmsynthese bekannt.

In Abhängigkeit von der Robotersprache können Syntaxaktionen leer sein, z. B. `DECL-PROGRAM-VARIABLE-POSITION` (Zeile 30) für KUKA KRL, da die Deklaration und Initialisierung einer Programmvariablen in derselben Syntaxaktion `INIT-PROGRAM-VARIABLE-POSITION` (Zeile 32) erfolgt. Leere Syntaxaktionen müssen nicht in der Syntaxdefinition aufgeführt sein.

Syntaxaktionen enthalten vordefinierte Variablen (eingerahmt durch ”#”) zur Verfügung. Deren Werte entsprechen denen der Elemente aus dem jeweiligen Fach. Die Syntaxaktion in Zeile 32 enthält z. B. die Variablen `#NAME#` (Programmvariablenname) und `#X#`, `#Y#`, `#Z#`, `#ROLL#`, `#PITCH#`, `#YAW#` (Initialisierungswerte, Zeile 33).

Syntaxaktionen existieren u. a. für die Deklaration aller Variablenarten (S. 101), für Variableninitialisierungen, -import, -zugriff und für jeden Roboterbefehl (Tabelle 7.2).

Aufbauaktion

Die *Aufbauaktionen* (Bild 7.9) ordnen die Haupt-, Sub- und Interruptroutinen sowie der Deklarationen für Variablen, Importe und Ein-/Ausgänge einer konkrete Robotersprache an.

Eine Aufbauaktion enthält eine Liste von Aktionen (`ACTION`). Diese Aktionen sind entweder Syntaxaktionen, weitere Aufbauaktionen oder vordefinierte Schlüsselwörter. Die Programmsynthese startet bei der Aufbauaktion `START` (Zeile 7) und führt deren Liste von oben nach unten aus. Trifft sie auf eine Aufbauaktion, so wird diese als Nächstes verarbeitet. Trifft sie auf eine Syntaxaktion, so wird diese Syntaxaktion für jedes Element des zugehörigen Fachs angewendet. Trifft sie auf ein vordefiniertes Schlüsselwort, so werden spezifische Funktionen ausgeführt. Ein Beispiel für ein vordefiniertes Schlüsselwort ist `MAINROUTINE-STMT` (Zeile 21), das die Syntaxaktionen für alle Roboterbefehle aus dem Fach Hauptroutine des Programmcontainers ausführt.

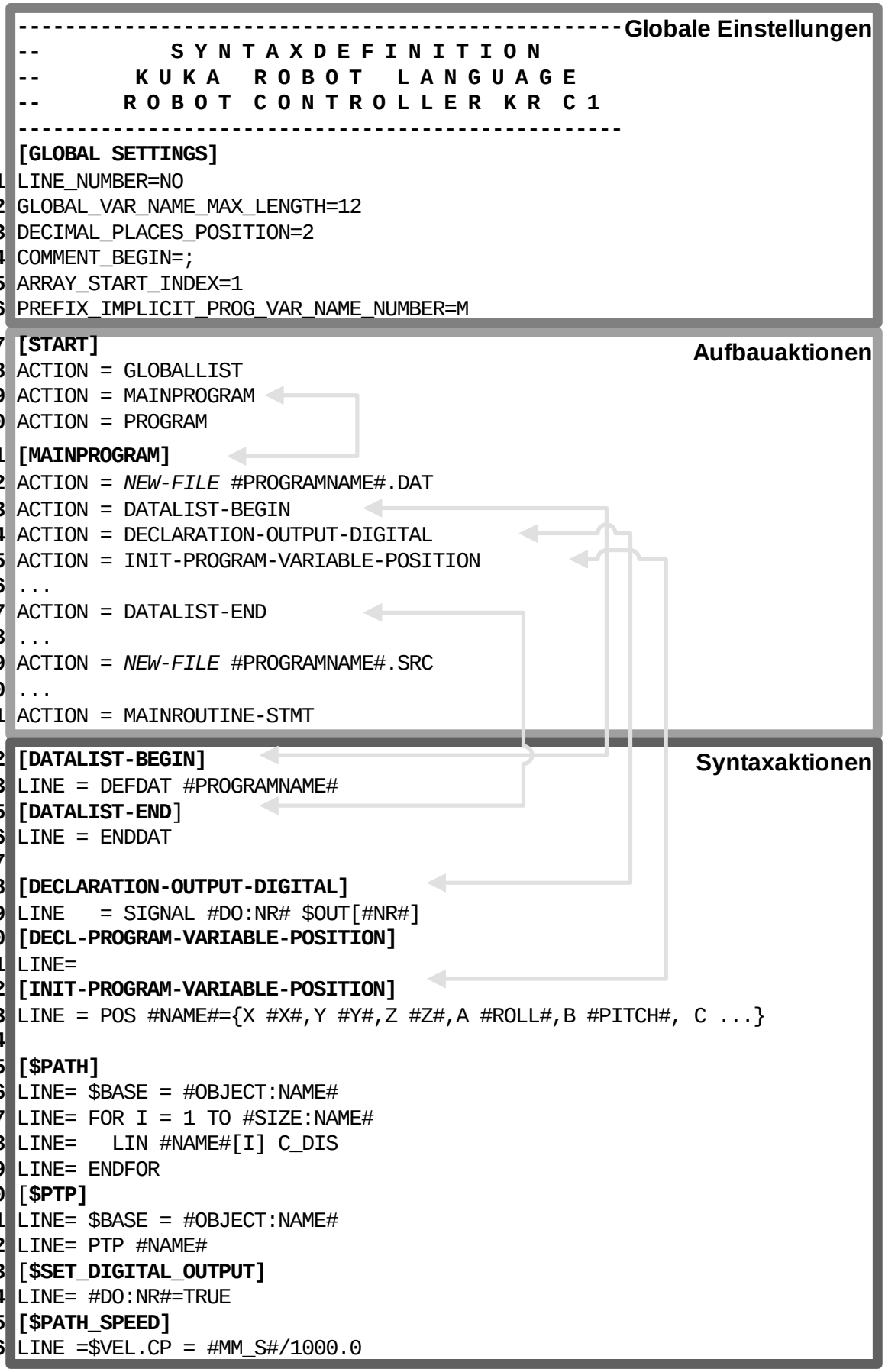


Bild 7.9: Auszug aus der Syntaxdefinition für die Sprache KRL von KUKA

Des Weiteren kann in der Liste einer Aufbauaktion das vordefinierte Schlüsselwort `NEW-FILE` (Zeilen 12, 19) verwendet werden. Diese gibt an, dass die Programmsynthese eine neue Roboterprogrammdatei mit dem angegebenen Namen erzeugen soll. In dieser Datei stehen bis zum nächsten Auftreten von `NEW-FILE` alle Roboterprogrammzeilen der im weiteren Verlauf bearbeitenden Syntaxaktionen.

Beispiel einer Syntaxdefinition

Ein Auszug aus der Syntaxdefinition für die Steuerung KUKA KR C1 mit der Sprache KRL ist in Bild 7.9 dargestellt. Bei KRL ist es üblich, alle Variablen eines Roboterprogramms mit deren Initialisierungswerten in einer Datenliste (DAT-Datei) zusammenzufassen und die Roboterbefehle in eine Programmdatei (SRC-Datei) zu schreiben. Durch die beiden Schlüsselwörter `NEW-FILE` (Zeilen 12, 19) legt die Programmsynthese die DAT- und die SRC-Datei an. Für die Datenliste erzeugt die Syntaxaktion `DATALIST-BEGIN` (Zeile 13, 22) einen Kopf und die Syntaxaktion `DATALIST-END` (Zeile 17, 25) eine Schlusszeile. Nach dem Erzeugen der SRC-Datei (Zeile 19) legt die Aufbauaktion `MAINPROGRAM` u. a. die Anweisungen der Hauptroutine (`MAINROUTINE-STMT`, Zeile 21) an. Diese Aktion ruft für alle Anweisungen aus dem Fach Hauptroutine (Bild 7.7) die Syntaxaktionen für die entsprechenden Roboterbefehle (z. B. `$PATH`, `$PATH_SPEED`, `$SET_DIG_OUTPUT`, Zeilen 35ff) auf.

Für jeden Roboterbefehl (Tabelle 7.2, S. 104) existiert exakt eine Syntaxaktion mit demselben Namen. In den Syntaxaktionen der Roboterbefehle kann man Variablen (eingerahmt durch "#") verwenden, die denselben Namen besitzen wie der formale Parameter des zugehörigen Roboterbefehls (z. B. `#NAME#` in `$PATH`, Zeile 38). Ist der formale Parameter ein Feld, so existiert eine zusätzliche Variable, die die Größe des Arrays angibt (z. B. `#SIZE:NAME#`, Zeile 37). Ist der formale Parameter vom Typ `$POSITION`, so existiert eine Variable in der Syntaxaktion (z. B. `#OBJECT:NAME#`, Zeile 36, 41), die einen Zugriff auf eine globale Variable im Roboterprogramm enthält, in der der Standort des Bezugsobjekts aus dem Simulationsmodell enthalten ist. Bei Befehlen, die auf digitale Ein-/Ausgänge zugreifen (z. B. `$SET_DIG_OUTPUT`, Zeile 43), steht neben der Nummer des Ein-/Ausgangs (`#NR#`, Zeile 29) auch dessen Name aus dem Simulationsmodell zur Verfügung (`#DO:NR#`, Zeile 29, 44).

Die Ausführung der Aufbauaktion `MAIN-PROGRAM` der Syntaxdefinition aus Bild 7.9 ist in Bild 7.10 dargestellt. Darin wird verdeutlicht, wie die Programmsynthese während der Ausführung einzelner Aufbau- und Syntaxaktionen auf die Repräsentation der Roboterprogramme aus Bild 7.8 zugreift. Es ist auch zu erkennen, welche der Syntaxaktionen welchen Ausschnitt des generierten Programms erzeugen.

Bibliotheken

Es gibt Robotersteuerungen, z. B. Bosch rho4 und Mitsubishi CR2 aus der Untersuchung in Abschnitt 7.1 (S. 97), deren Sprachsyntax einzelner Befehle und Datenstrukturen von der Achszahl des Roboters abhängt. Dies hat zur Folge, dass die notwendigen Syntaxdefinitionen größtenteils identische Aufbau- und Syntaxaktionen besitzen. Sind Änderungen in den identischen Teilen der Syntaxdefinitionen notwendig, müssen alle zugehörigen Syntaxdefinitionen einzeln geändert werden. Um

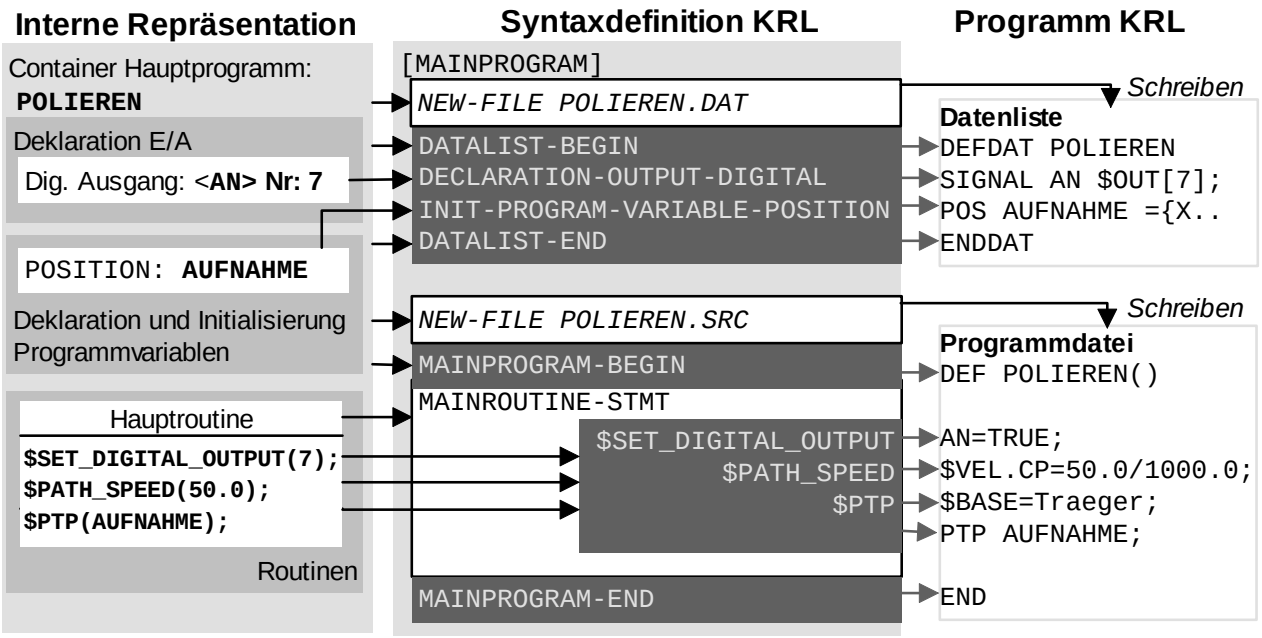


Bild 7.10: Beispiel zur Ausführung einer Syntaxdefinition

einerseits die Wartung der Syntaxdefinitionen zu vereinfachen und andererseits die Möglichkeit zu schaffen, neue Syntaxdefinitionen auf Basis bestehender Syntaxdefinitionen zu erstellen, wurde das Konzept der Bibliotheken für Syntaxdefinition entwickelt (Bild 7.11).

Durch die Anweisung `[IMPORT]` besteht für eine Syntaxdefinition die Möglichkeit, eine andere Syntaxdefinition vollständig zu importieren. Dadurch werden alle globalen Einstellungen, Aufbau- und Syntaxaktionen in der importierenden Syntaxdefinition gültig. Enthalten beide Syntaxdefinitionen Aktionen oder Einstellungen mit demselben Namen, so werden die Einstellungen und Aktionen der importierten Syntaxdefinition von der importierenden Syntaxaktion überschrieben. Auf diese Weise enthalten die importierenden Syntaxdefinitionen nur den kinematikspezifischen Teil der Sprache.

7.3.3 Architektur

Die Programmsynthese besteht aus den Modulen *Steuerbefehls-* und *Programmbefehlsempfänger*, *Parser*, *Syntaxdefinitionsinterpretierer*, *Roboterprogramm-* und *Syntaxdefinitionsfabrik* (Bild 7.12).

Der *Steuerbefehlsempfänger* erhält die Steuerbefehle (Tabelle 7.3) von der Funktionserweiterung Synthese (Bild 4.7, S. 46) und ist für die Befehlsausführung verantwortlich.

Der *Programmbefehlsempfänger* erhält die Programmbefehle (Tabelle 7.2, S. 104) ebenfalls von der Funktionserweiterung Synthese. Er prüft unter Verwendung eines endlichen Automaten, ob der CAR-Anwender die Befehle in der richtigen Reihenfolge (z. B. `$PROGRAM` und `$ENDPROGRAM`, Bild 7.4) angegeben hat, bevor er sie an die *Roboterprogrammfabrik* weiterleitet. Diese ist für den Auf- und Abbau der internen Repräsentation der steuerungsneutralen Roboterprogramme zuständig. Der Aufbau erfolgt über die erhaltenen Programmbefehle und der Abbau über den Steuerbefehl `$RESET`.

Nachdem der *Parser* den *Steuerbefehlsempfänger* beauftragt hat, die Syntaxdefinition einzulesen

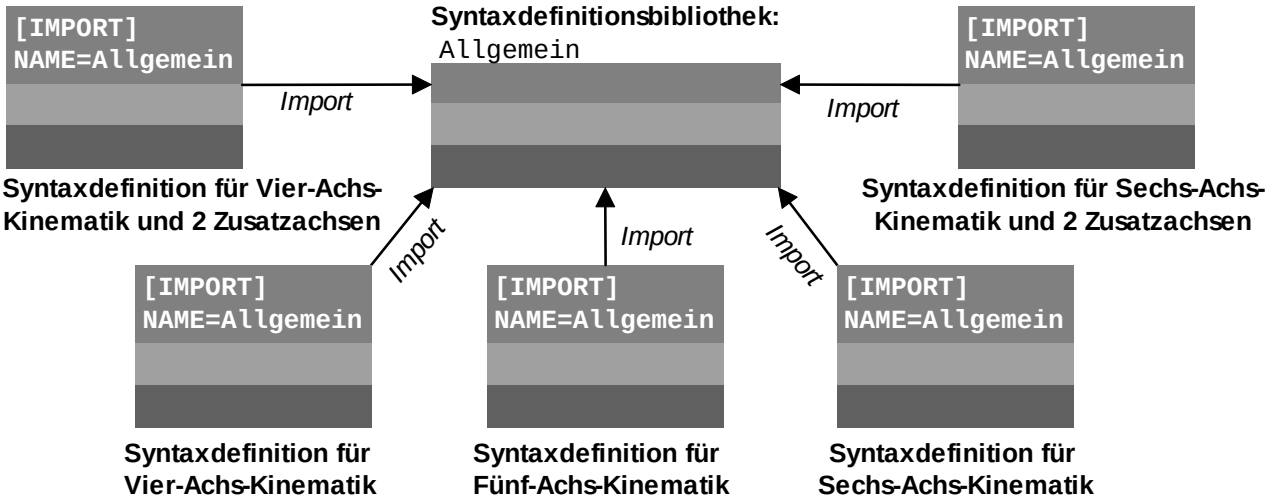


Bild 7.11: Bibliotheken für Syntaxdefinitionen

Tabelle 7.3: Steuerbefehle der Programmsynthese

Steuerbefehl	Empfängermodul	Erklärung
`$LANGUAGE`	Parser	Festlegung Steuerungssyntax
`$LOAD_SYNTAX`	Parser	Einlesen Syntaxdefinition
`$RESET`	Roboterprogrammfabrik	Löschen Roboterprogramm
`$GENERATE`	Syntaxdefinitionsinterpretierer	Ausgabe Roboterprogramm

(Steuerbefehl `$LOAD_SYNTAX`), baut der *Parser* mit der *Syntaxdefinitionsfabrik* in Abhängigkeit vom Variablenwert `$LANGUAGE` die Repräsentation der Syntaxdefinition auf. Diese Repräsentation entspricht dem Aufbau der zugehörigen Dateien und wird deshalb nicht näher betrachtet.

Der *Steuerbefehlsempfänger* fordert über den Steuerbefehl `$GENERATE` den *Syntaxdefinitionsinterpretierer* auf, die Programme auszugeben. Daraufhin führt Letzterer die in der Syntaxdefinition angegebenen Aktionen aus. Er besitzt einen *Aktionsstack*, auf dem alle noch zu verarbeitenden Aktionen liegen. Die Aktionsverarbeitung führt immer die Aktion aus, die oben auf dem Aktionsstack liegt. Handelt sich dabei um eine Aufbauaktion, so schreibt sie die darin enthaltenen Aktionen auf den Aktionsstack. Handelt es sich um ein vordefiniertes Schlüsselwort (z. B. `NEW-FILE`), so führt sie die zugehörige Spezialfunktion aus. Handelt es sich um eine Syntaxaktion, so führt die Aktionsverarbeitung diese für jedes Element des zugehörigen Fachs aus. Die Ausführung enthält im Einzelnen:

1. Extraktion aller Variablennamen aus den Programmzeilen der Syntaxaktion (Bild 7.13a)
2. Berechnung der zugehörigen Werte aus der Repräsentation der Roboterprogramme (Bild 7.13b)
3. Ersetzen der Variablennamen durch die berechneten Werte in den Programmzeilen (Bild 7.13c)
4. Eintragen der resultierenden Programmzeilen in den Programmspeicher

Zur Erfüllung von Anforderung 8 (S. 33) kann die Programmsynthese die Planaufrufe als Kommentare in das Roboterprogramm schreiben. Damit ist für den CAR-Anwender im Roboterprogramm

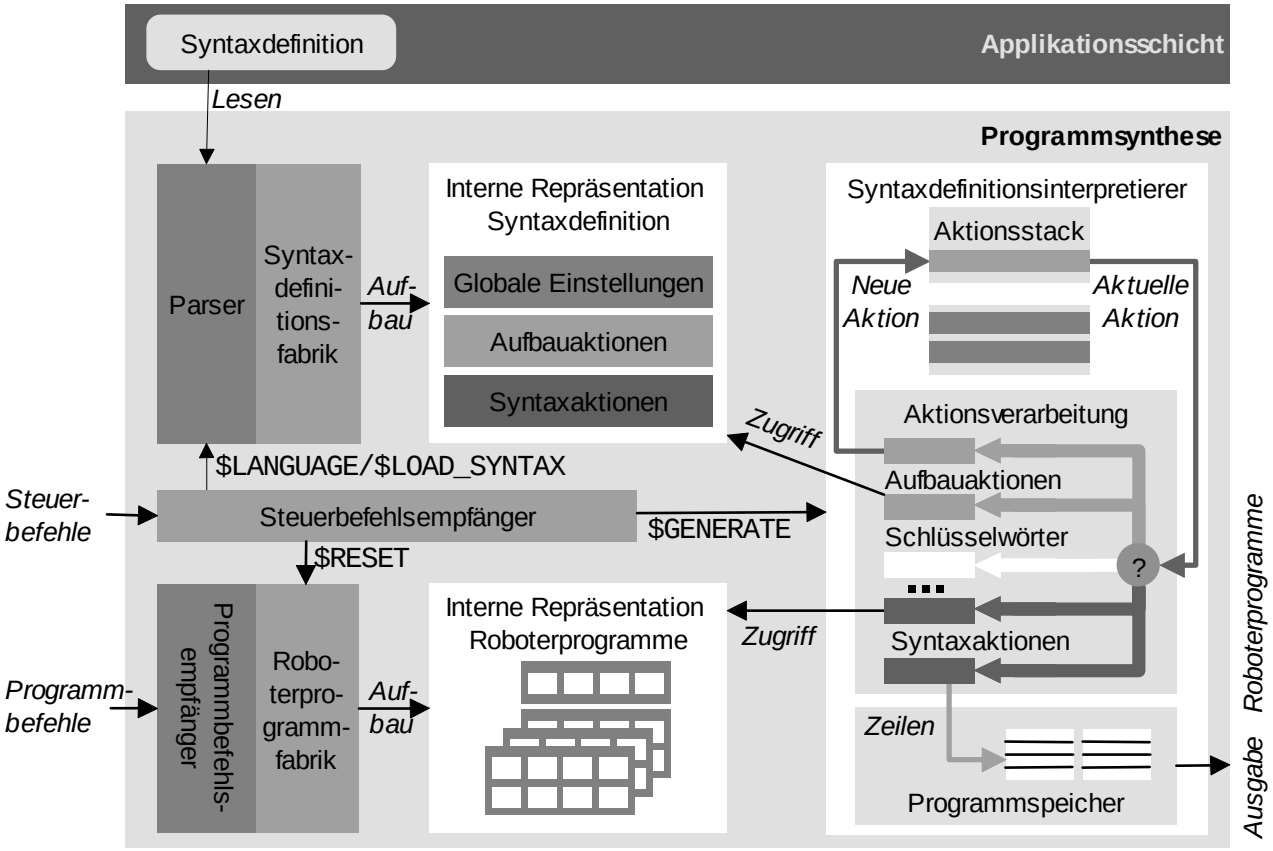


Bild 7.12: Architektur der Programmsynthese

erkennbar, welcher Programmbefehl für die nachfolgenden Anweisungen im Roboterprogramm verantwortlich ist. Problematisch ist die Tatsache, dass die Funktionserweiterung Synthese dazu keinen direkten Zugriff auf den Steuergraph besitzt, sondern sie kann die notwendigen Informationen nur über die Erweiterungsschnittstelle auslesen und darüber den Planaufruf, insbesondere dessen aktuelle Parameter, rekonstruieren.

Die Rekonstruktion des Planaufrufs findet wie folgt statt (Bild 7.14): Nachdem der Kern der Generierungssteuerung die Funktionserweiterung Synthese über z. B. `$SET_DIGITAL_OUTPUT` aufgerufen hat, ist zunächst nur der Namen der formalen Parameter (`NR`) bekannt. Über den Dienst *Abfrage Variablenwert* der Erweiterungsschnittstelle kann sich die Funktionserweiterung Synthese den internen

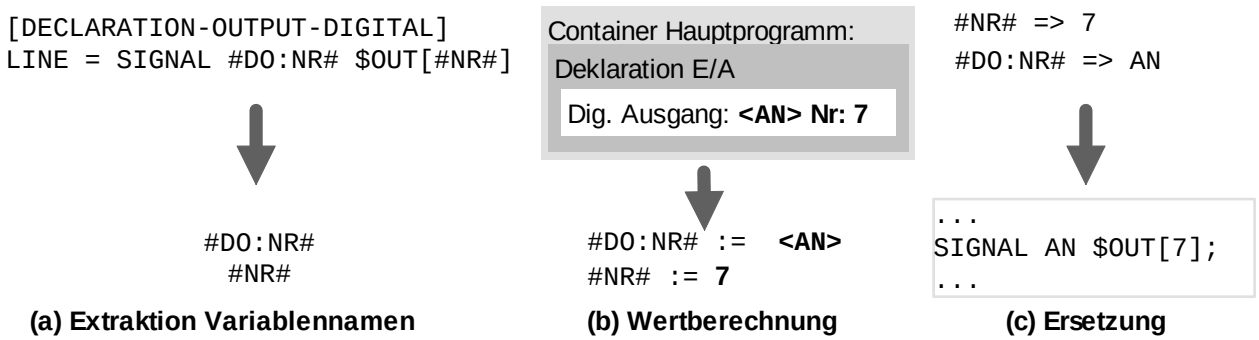


Bild 7.13: Verarbeitung einer Syntaxaktion

Ziel: Planaufrufe in Roboterprogrammen

Generierungsplan

```
POLIERMASCHINE=7
...
$SET_DIGITAL_OUTPUT(NR=POLIERMASCHINE);
...
```

Roboterprogramm (KRL)

```
...
; $SET_DIGITAL_OUTPUT(NR=POLIERMASCHINE)
AN=TRUE
...
```

Ablauf:

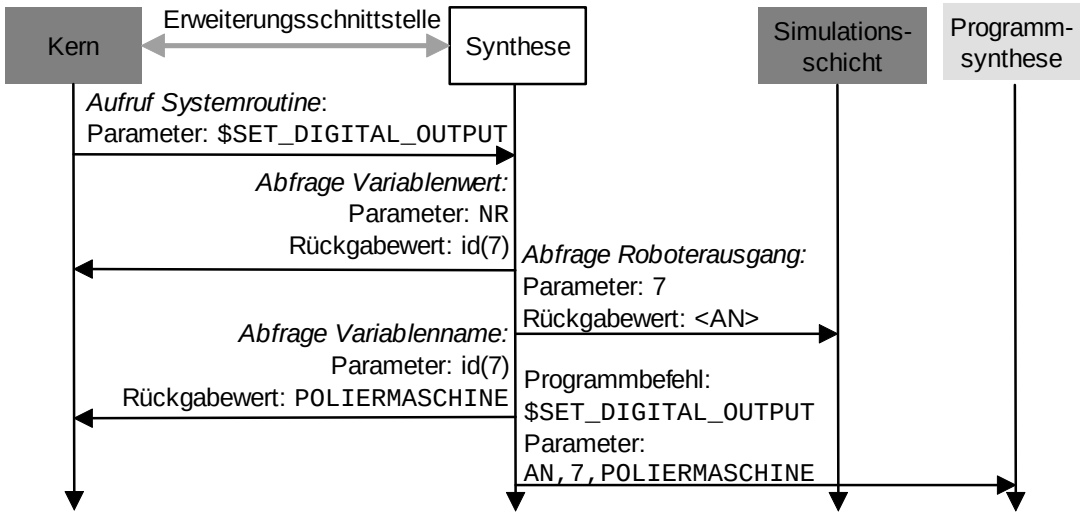


Bild 7.14: Rekonstruktion des Planaufrufs der Funktionserweiterung Synthese

Bezeichner des Werts des aktuellen Parameters (`id(7)`) besorgen. Ist der aktuelle Parameter eine Variable, so kann die Funktionserweiterung Synthese über den Dienst *Abfrage Variablenname* den Namen (`POLIERMASCHINE`) aus der globalen Variablenverwaltung des Kerns (Bild 6.6, S. 84) oder der lokalen Variablenverwaltung des Prozessors (Bild 6.9, S. 88) erhalten. Ist der aktuelle Parameter keine Variable, so scheitert der Dienst *Abfrage Variablenname* und die Funktionserweiterung Synthese ergänzt den Wert des aktuellen Parameters zur Rekonstruktion des Planaufrufs. Zur vollständigen Zusammenstellung des Programmbefehls für `$SET_DIGITAL_OUTPUT` ist der Name des digitalen Ausgangs aus dem Wert des aktuellen Parameters erforderlich. Dies erhält die Funktionserweiterung Synthese aus der Simulationsschicht über *Abfrage Roboterausgang*.

Der Ablauf aus Sicht der Funktionserweiterung Synthese ist in Bild 7.14 dargestellt. Entscheidend dabei ist der Dienst *Abfrage Variablenwert* der Erweiterungsschnittstelle, der zu einem internen Bezeichner eines Werts (`id(7)`), falls möglich, den zugehörigen Variablennamen (`POLIERMASCHINE`), ermittelt. Scheitert dies, wird statt des Variablennamens der Wert verwendet. Im Programmbefehl, den die Funktionserweiterung Synthese an die Programmsynthese schickt, sind somit alle Informationen zum Schreiben des Planaufrufs in das Roboterprogramm vorhanden. Somit ist Anforderung 8 (S. 33) erfüllt.

8 Simulation der Roboterprogramme

Nur die Programmausführung in einem Simulationslauf erlaubt deren Bewertung ohne den Einsatz der realen Roboter-Fertigungszelle (Anforderung 11, S. 34). Im Gesamtsystem übernimmt diese Aufgabe die Simulationsschicht (Bild 4.1, S. 37), die dazu eine virtuelle Robotersteuerung benötigt, für die es es vier Anbindungsmöglichkeiten gibt (Bild 4.9, S. 49). Mit der ersten Möglichkeit ist ein herstellerübergreifender Vergleich der Programmausführung möglich. Der wesentliche Nachteil dabei ist der hohe Realisierungsaufwand, weil pro Sprache ein Übersetzer erforderlich ist. Das hier beschriebene Framework zur Entwicklung solcher Compiler ein (Bild 4.10, S. 51) senkt diesen Aufwand erheblich.

Um Ausführungsfehler im Vorfeld eines Einsatzes in der realen Fertigungszelle aufzuspüren, muss die Simulationsschicht entsprechende Komponenten zur Fehlererkennung besitzen. Dazu erfolgt zunächst eine Einteilung der Fehlerarten. Daran schließt sich eine Beschreibung der Komponenten an.

Die Bewertungsergebnisse der Programmausführung im Simulationslauf können dann zur Verbesserung der Roboterprogramme herangezogen werden (innere Rückkopplung, Bild 4.1, S. 37). Dafür bieten sich Optimierungsverfahren an, deren hier Einsatz erläutert wird.

Abschließend erfolgt eine Übersicht über die Dienste der Simulationsschicht, die überlagerte Funktionserweiterungen verwenden.

8.1 Framework für die Programmtransformation

Nur bei der ersten Möglichkeit zur Anbindung einer virtuellen Robotersteuerung (Bild 4.9, S. 49) treten zwei aufeinander folgende Konvertierungen mit zwei unterschiedlichen neutralen Repräsentationen der Roboterprogramme auf.

Zunächst konvertieren die Generierungssteuerung und die Programmsynthese die steuerungsneutrale Programmspezifikation in ein steuerungsspezifisches Roboterprogramm. Danach konvertiert ein Compiler das steuerungsspezifische Programm in einen neutralen Zwischencode (IRDATA [4]). Die Konvertierung von einer neutralen Repräsentation (Programmspezifikation) über den Umweg der steuerungsspezifischen Roboterprogramme in eine andere neutrale Repräsentation (Zwischencode) findet aus folgenden Gründen statt:

1. Der CAR-Anwender soll beim Wechsel der Robotersteuerung die Programmierung nicht wiederholen müssen (Anforderung 5, S. 31). Dies erfordert die Programmspezifikation.

2. Das System muss steuerungsspezifische Roboterprogramme erzeugen, damit eine industrielle Robotersteuerung diese ausführen kann. Dies erfordert die erste Konvertierung.

3. Die steuerungsneutrale Programmspezifikation enthält nur eine Untermenge der Funktionen heutiger Robotersteuerungen. Um aber die erforderliche Gesamtmenge abbilden zu können, ist eine umfassendere Repräsentation, der einheitliche Zwischencode, erforderlich.

4. Damit die generierten Programme den vollständigen Funktionsumfang der Robotersteuerung verwenden können, muss das System auch manuell erstellte Roboterprogramme prüfen (Anforderung 7, S. 32). Deshalb reicht die Prüfung der Programmspezifikation nicht. Folglich ist die zweite Konvertierung erforderlich.

5. Um Fehler in der Syntaxdefinition zu finden, muss man die steuerungsspezifischen Programme prüfen. Diese Prüfung erfordert ebenfalls die zweite Konvertierung.

Des Weiteren ist das Abstraktionsniveau der beiden neutralen Repräsentationen unterschiedlich. Während die Programmspezifikation das Abstraktionsniveau einer prozeduralen Hochsprache besitzt, ist der Zwischencode mit Assembler vergleichbar.

8.1.1 Architektur

Im Gegensatz zur Synthese von Roboterprogrammen, bei der es möglich war, ein vollständig steuerungsunabhängiges Modul (Programmsynthese) für die Erzeugung der Roboterprogramme zu realisieren, ist dies bei den Compilern nicht möglich. Dies liegt an der Tatsache, dass bei der Synthese für die Erreichung eines bestimmten Roboterverhaltens exakt eine Beschreibung in der Sprache der Robotersteuerung ausreicht. Dagegen muss der Compiler alle Beschreibungsmöglichkeiten einer Sprache berücksichtigten. Dies erfordert eine spezifische Behandlung jeder Robotersprache. Bisher wurde für jede Robotersprache jeweils ein vollständig neuer Compiler entwickelt (z. B. für Bosch BAPS, Unimation VAL II, Mitsubishi Movemaster Command und Mitsubishi MELFA BASIC III), ohne dabei bestehende Realisierungen wieder zu verwenden [51, 138]. Die Schwierigkeit der Wiederverwendung liegt an den starken Abhängigkeiten zwischen der compilerinternen Repräsentation der Roboterprogramme (Datenstrukturen) und deren sprachspezifischen Übersetzungsvorschriften (Algorithmen). Erschwerend kommt hinzu, dass industrielle Robotersprachen aufgrund ihrer Historie selten eine einheitliche Struktur besitzen. Dies führt bei der Realisierung von Compilern zu einer Fülle von zu behandelnden Ausnahmen in den Übersetzungsvorschriften.

Das hier vorgeschlagene Framework enthält die Phasen eines Compilers [10] in jeweils einem Modul gekapselt (Bild 8.1): Parser, Kontextanalyse, Umwandlung, Codeerzeugung und Linker. Jedes Modul liest Eingangsdaten, verarbeitet diese und erzeugt Ausgangsdaten, die es entweder in Dateien (z. B. symbolischer IRDATA-Code) oder in einer internen Repräsentation ausgibt.

Im Gegensatz zum Aufbau vieler Übersetzer, die aus Gründen der Speicherkapazität das Roboterprogramm nur soweit im Speicher halten, wie sie es für die Übersetzung und die Codeoptimierung benötigen, lesen die mit dem Framework entwickelten Compiler das Roboterprogramm vollständig ein, bevor sie es übersetzen. Erst dadurch wird eine Trennung der Phasen der Übersetzung in eigenständige Module möglich.

Der *Parser* liest Dateien der folgenden Kategorien ein und baut aus diesen eine interne Repräsentation des Roboterprogramms, den abstrakten Syntaxbaum (AST), auf (Bild 8.1):

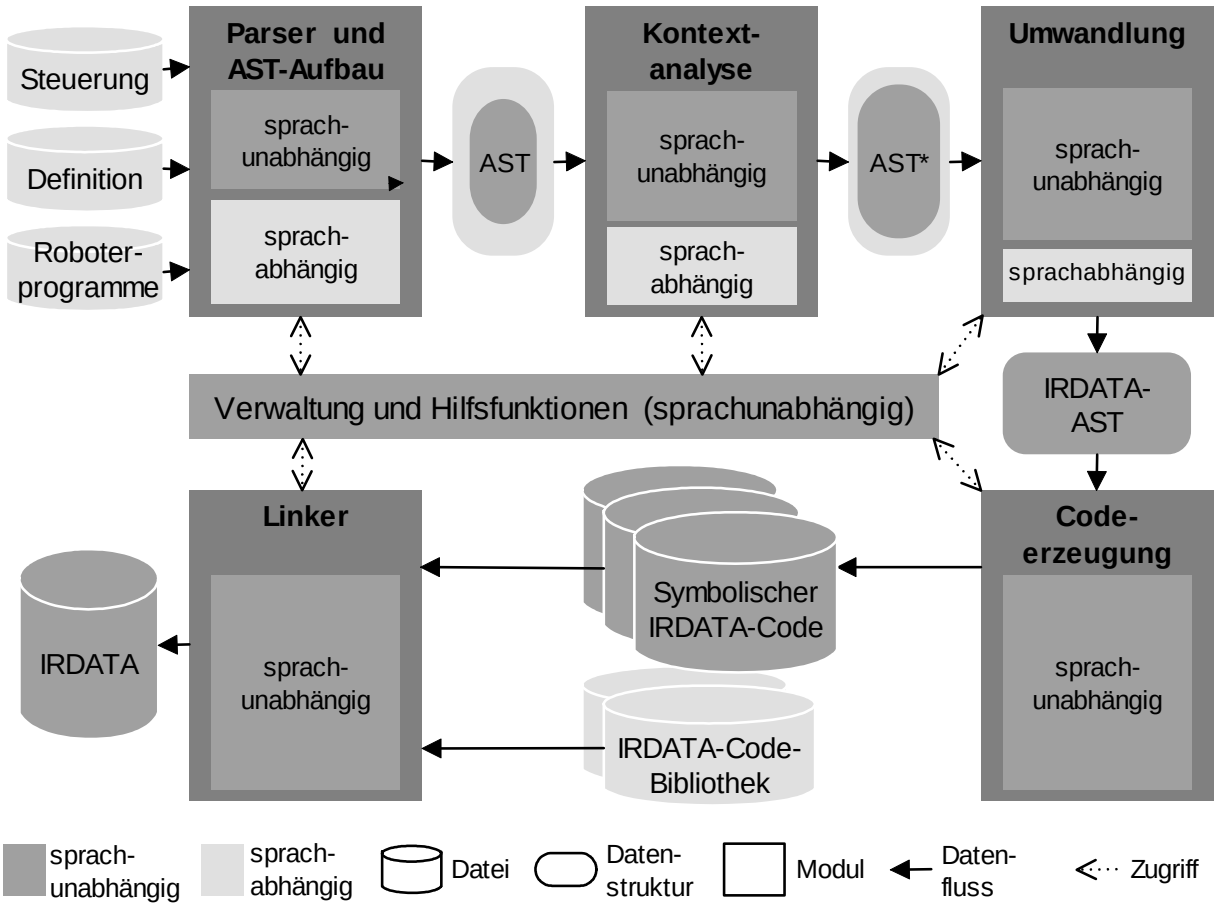


Bild 8.1: Framework zum Aufbau eines Programmtransformators

1. Steuerung: Dies Dateien stammen von der realen Robotersteuerung. Sie enthalten Deklarationen und Initialisierungen von Variablen, Befehlen und Datentypen (z. B. die Datei $CONFIG.DAT der Steuerung KUKA KR C1 oder KR C2).

2. Definition: Diese Dateien enthalten ebenfalls Deklarationen und Initialisierungen vordefinierter Variablen, Befehlen und Datentypen. Diese sind der Robotersteuerung bekannt, aber nicht explizit in der ersten Kategorie vorhanden. Diese Datei muss der Compilerentwickler anlegen.

3. Roboterprogramme: Dies sind die zu übersetzenden Roboterprogramme, die im Simulationslauf ausgeführt werden sollen.

Die *Kontextanalyse* prüft den abstrakten Syntaxbaum auf Korrektheit und Konsistenz, z. B. korrekte Datentypen bei Parameterübergabe der Roboterbefehle. Sie modifiziert und ergänzt den eingelesenen abstrakten Syntaxbaum in einen erweiterten abstrakten Syntaxbaum (AST*) um weitere Informationen (z. B. den Datentyp eines aktuellen Parameters oder Typkonvertierungsfunktionen).

Die *Umwandlung* baut dann den erweiterten abstrakten Syntaxbaum in einen abstrakten Syntaxbaum der Zielsprache IRDATA um (IRDATA-AST). Bevor die *Codeerzeugung* den IRDATA-Syntaxbaum in Dateien ausgibt. Im Gegensatz zur Norm DIN 66314 [4], bei der die IRDATA-Befehle als Zahlen-

codes dargestellt sind, werden aus Gründen der Lesbarkeit symbolische Bezeichner für die IRDATA-Befehle ausgegeben (symbolischer IRDATA-Code).

Die *Umwandlung* und *Codeerzeugung* erzeugen für jeden Roboterbefehl einen Aufruf in eine so genannte *IRDATA-Code-Bibliothek*, die die Implementierung des Roboterbefehls in der Syntax des symbolischen IRDATA-Codes enthält. Die IRDATA-Code-Bibliotheken wurden eingeführt, um den zu übersetzenden Sprachumfang ohne Eingriff in die Compilermodule erweitern und modifizieren zu können. Der *Linker* führt abschließend die symbolischen IRDATA-Dateien zusammen, die die virtuelle Robotersteuerung (z. B. von KEIBEL [84]) dann ausführen kann.

Insgesamt realisiert das Framework - mit Ausnahme der Grammatik im *Parser* - einen vollständigen Übersetzer für eine prozedurale Hochsprache. Sowohl die Übersetzungsvorschriften der Module *Parser*, *Kontextanalyse* und *Umwandlung* als die abstrakten Syntaxbäume (AST, AST*) sind in einen sprachabhängigen und in einen sprachunabhängigen Teil geteilt. Der sprachunabhängige Teil stellt ein Standardverhalten zur Verfügung, das für die jeweilige Sprache im sprachabhängigen Teil überschrieben werden kann. Dadurch kann sich der Compilerentwickler vollständig auf die spezifischen Elemente seiner Sprache konzentrieren, ohne dass er dabei das Standardverhalten jeweils neu implementieren muss.

Insgesamt reduziert das Framework den Realisierungsaufwand eines Compilers erheblich (Tabelle 8.1). Die angegebenen Entwicklungszeiten beziehen sich auf die Entwicklungszeit des ohne das Framework entwickelten Compilers [138] für IRL (Industrial Robot Language [2]). Alle mit dem Framework entwickelten Compiler sind vom Umfang her vergleichbar mit dem IRL-Compiler, deshalb ist für alle Compiler die geschätzte Entwicklungszeit 100 %. Die Varianz der erzielten Reduktion zwischen den KRL/Rapid-Compilern auf der einen Seite und den V+/MELFA BASIC IV-Compilern auf der anderen Seite begründet sich dadurch, dass der KRL-Compiler, der Rapid-Compiler und das Framework parallel entwickelt wurden. Dadurch enthalten die Entwicklungszeiten des KRL- und des Rapid-Compilers die Entwicklungszeit des Frameworks. Die Synergieeffekte, die durch die parallele Entwicklung entstanden, führten dazu, dass die Entwicklungszeiten des KRL- und des Rapid-Compilers auf unter 100 % reduziert wurden. Bei einer Entwicklung nur eines Compilers zusammen mit dem Framework wäre eine Entwicklungszeit deutlich über 100% zu erwarten. Unter der Voraussetzung, dass das Framework für den KRL- und den Rapid-Compiler bereits vorhanden gewesen wäre, wäre eine Reduktion vergleichbar mit den Sprachen V+ und MELFA BASIC IV zu erwarten.

Tabelle 8.1: Entwicklungzeit der Compiler (in % der Entwicklungzeit des IRL-Compilers[138])

Sprache	Geschätzte Entwicklungszeit	Tatsächliche Entwicklungszeit	Erzielte Reduktion
KRL	100 %	75 %	-25 %
Rapid	100 %	83 %	-17 %
V+	100 %	29 %	-71 %
MELFA BASIC IV	100 %	17 %	-83 %

Sprachunabhängiger Teil

```
'type' STATEMENT
  -- Allgemeine Anweisungen
 while  (...,
  Cond  : EXPRESSION,
  Body  : STATEMENTS
 )
  -- Spezifische Anweisungen
 specific_stmt(...,
  Stmt: SPECIFIC_STMT
 )
```

Sprachabhängiger Teil

```
'type' SPECIFIC_STMT

 -- linear movement
 mba4_mvs(...,
   Target: EXPRESSION,
   ...
 )
```

Bild 8.2: Teilung des abstrakten Syntaxbaums am Beispiel von Anweisungen

Neben der Reduktion des Entwicklungsaufwands besitzt das Framework einen weiteren Vorteil, der sich durch die Dateien der ersten beiden Kategorien und den IRDATA-Code-Bibliotheken ergibt. Bei neuen Roboterbefehlen oder Änderungen an bestehenden Roboterbefehlen braucht man die Compiler nicht zu modifizieren, sofern der neue Roboterbefehl dieselbe Struktur besitzt wie die anderen Befehle. Man muss lediglich die entsprechende Deklaration in einer Datei aus den ersten beiden Kategorien und die zugehörige Implementierung in einer der IRDATA-Code-Bibliotheken ergänzen oder ändern. Praktische Erfahrungen mit diesem Konzept haben gezeigt, dass sich der Wartungsaufwand erheblich reduziert [51].

8.1.2 Teilung des abstrakten Syntaxbaums

Das Framework wurde mit GENTLE aufgebaut [144]. Dabei sind die abstrakten Syntaxbäume in Form von Prädikaten und die Übersetzungsvorschriften in Form von Regeln realisiert. Die Regelausführung erfolgt ähnlich wie in Programmiersprache Prolog [31].

Bild 8.2 zeigt die Teilung des abstrakten Syntaxbaums anhand von Anweisungen. Er enthält neben allgemeinen Anweisungen (z. B. `while`) ein besonderes Prädikat zur Kennzeichnung von sprachspezifischen Roboterbefehlen (`specific_stmt(...)`). Darin werden alle sprachspezifischen Roboterbefehle aufgenommen (z. B. `mba4_mvs`), die nicht einer Datei der Kategorien ”Steuerung” oder ”Definition” zugeordnet werden können. Die Teilung der abstrakten Syntax ist notwendig, weil nicht alle Roboterbefehle dieselbe formale Struktur besitzen, und somit manche Befehle eine spezifische Übersetzungsvorschrift erfordern, wie z. B. der folgende Befehl in MELFA BASIC IV [107]:

```
MVS P1, -100 WTHIF M_IN(4)=1, M_OUT(6)=1 DLY 0.5
```

Dieser Befehl fährt linear zu einer Position, die 100 mm von der Position P1 entfernt ist. Wenn während dieser Bewegung der Eingang 4 gesetzt wird, so setzt die Steuerung den digitalen Ausgang 6 für 0.5 Sekunden und setzt ihn anschließend zurück. Bei diesem Befehl sind die Parameter mit mehreren

Sprachunabhängige Regeln

```
'action' AnalyzeStmts(STATEMENT -> STATEMENT)
 -- Regel 1
'rule' AnalyzeStmts(Stmt -> A_Stmt):
    -- Überschreiben der Regeln für allgemeine Anweisungen
    Pre_AnalyzeStmts(Stmt -> A_Stmt)
 -- Regel 2
'rule' AnalyzeStmts(while(…) -> while(…)):
    -- Standardverhalten für WHILE
 -- Regel 3
'rule' AnalyzeStmts(specific_stmt(S)->specific_stmt(A_S)):
   -- Regeln für spezifische Anweisungen
    Post_AnalyzeStmts(S->A_S)
```

Sprachabhängige Regeln (Implementierung vom Compilerentwickler)

```
'action' Pre_AnalyzeStmts(STMT -> STMT)
  -- Überschreiben der Regeln für allgemeine Anweisungen

'action' Post_AnalyzeStmts(SPECIFIC_STMT -> SPECIFIC_STMT)
'rule' Post_AnalyzeStmts(mba4_mvs(…) -> mba4_mvs(…)):
  -- Regeln für spezifische Anweisungen
```

Bild 8.3: Teilung der Übersetzungsvorschriften am Beispiel der *Kontextanalyse*

befehlsspezifischen Schlüsselwörtern (`WTHIF`, `M_IN`, `M_OUT`, `DLY`) verbunden, die eine formale Beschreibung in einer Datei der Kategorie ”Steuerung” oder ”Definition” verhindert, weil sie von keiner Grammatikregel des Parsers erfasst wird.

8.1.3 Teilung der Übersetzungsvorschriften

Die Übersetzungsvorschriften in den Modulen *Parser*, *Kontextanalyse* und *Umwandlung* besitzen ebenfalls eine Aufteilung in einen sprachunabhängigen und in einen sprachabhängigen Teil. Bild 8.3 zeigt die Struktur der Regeln in der *Kontextanalyse*. Die Anwendung der Regeln auf den abstrakten Syntaxbaum erfolgt von oben nach unten. Ist eine Regel anwendbar (Unifikation der Regelbedingung mit den Prädikaten des abstrakten Syntaxbaums [10, 31]), so finden nachfolgende Regeln keine Berücksichtigung mehr.

Im sprachunabhängigen Teil existieren drei Arten von Regeln:

1. Regeln zum Überschreiben der allgemeinen Anweisungen (Regel 1)
2. Standardverhalten (Regel 2)
3. Regeln für sprachspezifische Anweisungen (Regel 3)

In Abhängigkeit von der zu bearbeitenden Anweisung wird eine dieser Regeln auf die Prädikate angewendet. Handelt es sich um eine allgemeine Anweisung, bei der man das Standardverhalten des

Frameworks überschreiben muss, so muss der Compilerentwickler Regel 1 für jede dieser Anweisung implementieren. Für alle Anweisungen, für die er das Standardverhalten des Frameworks übernehmen will, braucht er nichts implementieren, da alle Regeln 2 im Framework realisiert sind. Handelt es sich um eine sprachspezifische Anweisung, so erfolgt deren Übersetzung in Regel 3, die der Compilerentwickler implementieren muss.

Die sprachabhängigen Regeln wenden dabei die Übersetzungsvorschriften für die sprachspezifischen Roboterbefehle an. Die Regelstruktur enthält somit eine klare Trennung zwischen sprachunabhängigen und sprachabhängigen Verhalten.

8.2 Prüfung der Roboterprogramme

Die Ausführung der steuerungsspezifischen Roboterprogramme in der Realität kann Fehler verursachen, sodass die Programme nicht in der realen Roboter-Fertigungszelle einsetzbar sind. Um solche Fehler im Vorfeld aufzuspüren, ist die Simulationsschicht für die Fehlererkennung zuständig. Sie sendet bei einem erkannten Fehler ein Ereignis an die überlagerte Generierungsschicht.

8.2.1 Fehlerarten

Die Fehler, die bei der Ausführung der Roboterprogramme in einer realen Roboter-Fertigungszelle auftreten können, lassen sich in vier Kategorien einteilen:

1. *Programmfehler*: Dieser Fehler tritt bei falscher Verwendung der Roboterprogrammiersprache auf. Dazu zählen u. a. syntaktische Fehler, mehrfache Definition von Variablen in einem Gültigkeitsbereich und Befehlsparameter mit inkorrekten Datentypen. Diesen Fehler kann die Simulationsschicht vor deren Ausführung erkennen. Solche Fehler treten vorwiegend in manuell erstellten Programmen oder bei fehlerhaften Eintragungen in den Syntaxdefinitionen auf.

2. *Laufzeitfehler*: Dieser Fehler tritt während der Programmausführung in der Robotersteuerung auf. Mögliche Laufzeitfehler sind u. a. unerreichbare Zielpositionen, Berechnungsfehler (z. B. Division durch Null, ein Konfigurationswechsel während einer Folge von Linearbewegungen, der Versuch singuläre Stellungen zu durchfahren, zu kurze Anhaltewege oder die Überschreitung von Geschwindigkeits- und Beschleunigungsvorgaben, sodass der Roboter eine Bewegung nicht durchführen kann.

3. *Kollisionsfehler*: Dieser Fehler tritt bei einer unerwünschten Berührung oder Durchdringung von mindestens zwei Komponenten in der Fertigungszelle auf.

4. *Störungen*: Eine Störung ist eine unerwartete Funktionsweise eines an der Robotersteuerung angeschlossenen externen Gerätes, z. B. Greifer oder Sensoren. Dieser Fehler tritt bei Kabelbrüchen, Kurzschlüssen oder Ausfällen der externen Geräte auf.

Neben unerwünschten Kollisionen können auch erwünschte Kollisionen zwischen Komponenten der realen Roboter-Fertigungszelle während der Ausführung der Roboterprogramme auftreten, wie z. B. der Kontakt einer Produktvariante mit der Polierscheibe aus dem einleitenden Beispiel (Bild 1.1, S. 2). Erwünschte Kollisionen führen nicht zu einem Fehler.

8.2.2 Komponenten zur Fehlererkennung

Um die genannten Fehlerarten im Vorfeld des Programmeinsatzes in der realen Roboter-Fertigungszelle aufzuspüren, besitzt die Simulationsschicht zur Fehlererkennung folgende Komponenten:

1. Compiler
2. Virtuelle Robotersteuerung
3. Kollisionserkennung
4. Störungsvorgabe

Die Compiler sind zur Erkennung von Programmfehlern und die virtuelle Robotersteuerung von KEIBEL [84] zur Erkennung von Laufzeitfehlern einsetzbar. Die an COSIMIR® angebundenen realen Robotersteuerungen (Möglichkeit 3, Bild 4.9, S. 49) sind nur eingeschränkt zur Fehlererkennung von Laufzeitfehler einsetzbar, da diese in der Regel in einen Fehlerzustand gehen, der nicht immer von außen sichtbar ist oder während der Planausführung rücksetzbar ist.

Die nicht im Rahmen dieser Arbeit entwickelte Kollisionserkennung von COSIMIR® dient zur Entdeckung von Kollisionsfehlern. Sie findet Kollisionen zwischen Gruppen von verschiedenen Objekten im Simulationsmodell (Bild 4.8, S. 48). Für die Unterscheidung, ob eine erwünschte oder eine unerwünschte Kollision vorliegen, ist eine Erweiterung dieser Kollisionserkennung erforderlich. Durch eine in dieser Arbeit entwickelte Erweiterung kann der CAR-Anwender Objektgruppenpaare in einem Generierungsplan angeben. Falls es zu einer Kollision zwischen den Gruppen eines angegebenen Paares kommt, betrachtet das System diese Kollision als erwünscht und führt demnach nicht zu einem Kollisionsfehler. Alle anderen Kollisionen führen zu einem Kollisionsfehler.

Zur Erkennung von Störungen wurde im Rahmen dieser Arbeit ein entsprechendes COSIMIR®-Modul zur Störungsvorgabe entwickelt. Die Simulationsschicht lässt Störungen ab einem einstellbaren Zeitpunkt für eine gewisse Dauer eintreten. Eine Störung wird dabei entweder als Trennung einer E/A-Verbindung von einem Roboterausgang zu einem externen Gerät oder durch ein Erzwingen von Werten auf die Robotereingänge modelliert. Dadurch kann geprüft werden, wie die generierten und manuell erstellten Roboterprogramme bei Eintritt einer Störung reagieren. Die Störungsvorgabe wird im nächsten Abschnitt genauer betrachtet.

Für jede Fehlerart wird ein Ereignis an die Generierungsschicht gesendet. Der CAR-Anwender kann in einem Plan durch die Verbindung einer Systemvariablen (Tabelle 8.2) für das jeweilige Ereignis mit einer Fehlerbehandlungsroutine auf diese Fehlerarten unterschiedlich im Plan reagieren.

Tabelle 8.2: Planvariblen der Ereignisse für Fehlerarten

Planvariable	Fehlerart
`$PROGRAM_ERROR`	Programmfehler
`$VRC`	Laufzeitfehler
`$COLLISION`	Kollisionsfehler
`$MALFUNCTION`	Störung

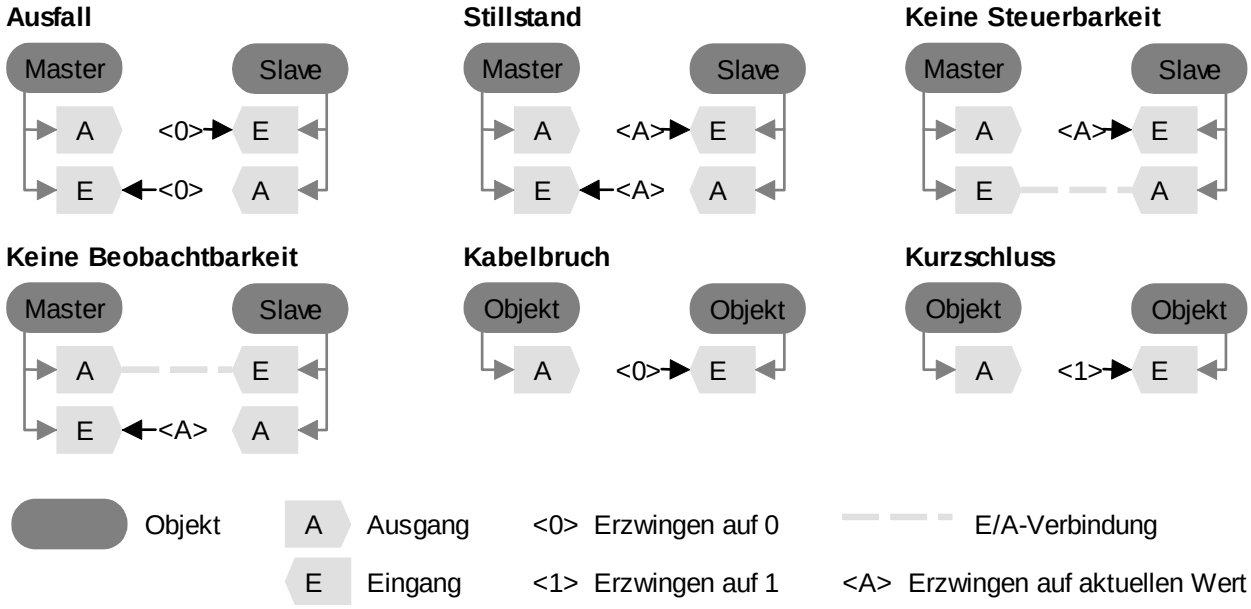


Bild 8.4: Störungstypen und ihre Abbildung im Simulationsmodell

8.2.3 Störungsvorgabe

Die Störungsvorgabe ändert am Simulationsmodell während des Simulationslaufs die E/A-Verbindungen zwischen den Objekten. Es wird zwischen verschiedenen Störungstypen (z. B. Stillstand) unterschieden (Bild 8.4). Eine Störung ist entweder objektbezogen (z. B. Ausfall) oder verbindungsbezogen (z. B. Kabelbruch). Bei objektbezogenen Störungen ist das Objekt, bei der die Störung auftritt der *Master*. Im Gegensatz dazu ist das Objekt, das eine Verbindung zu einem gestörten Objekt besitzt ein *Slave*. Tritt eine Störung beim Master-Objekt auf, so sind in der Regel mehrere Slave-Objekte davon betroffen. Je nach Störung kann der Roboter sowohl Master als auch Slave sein. Bei verbindungsbezogenen Störungen besteht diese Rollenverteilung nicht. Nach Störungsende stellt die Störungsvorgabe die ursprünglichen E/A-Verbindungen wieder her. Tritt z. B. ein Ausfall des Roboters, der dann das Master-Objekt ist, ein, so löst die Störungsvorgabe alle E/A-Verbindungen zu den mit dem Roboter verbundenen Objekten, die die Slave-Objekte darstellen. Stattdessen erzwingt sie die Eingangswerte des Roboters und alle Eingänge, die eine Verbindung zum Roboter hatten, auf Null.

Die Störungsvorgabe in der Simulationsschicht enthält eine Liste an Störungen, die während eines Simulationslaufs eintreten (Bild 8.5). Der CAR-Anwender gibt über eine Eingabemaske im Vorfeld an, welche Störungen während des Simulationslaufs eintreten sollen. Dabei legt er die Objekte fest,

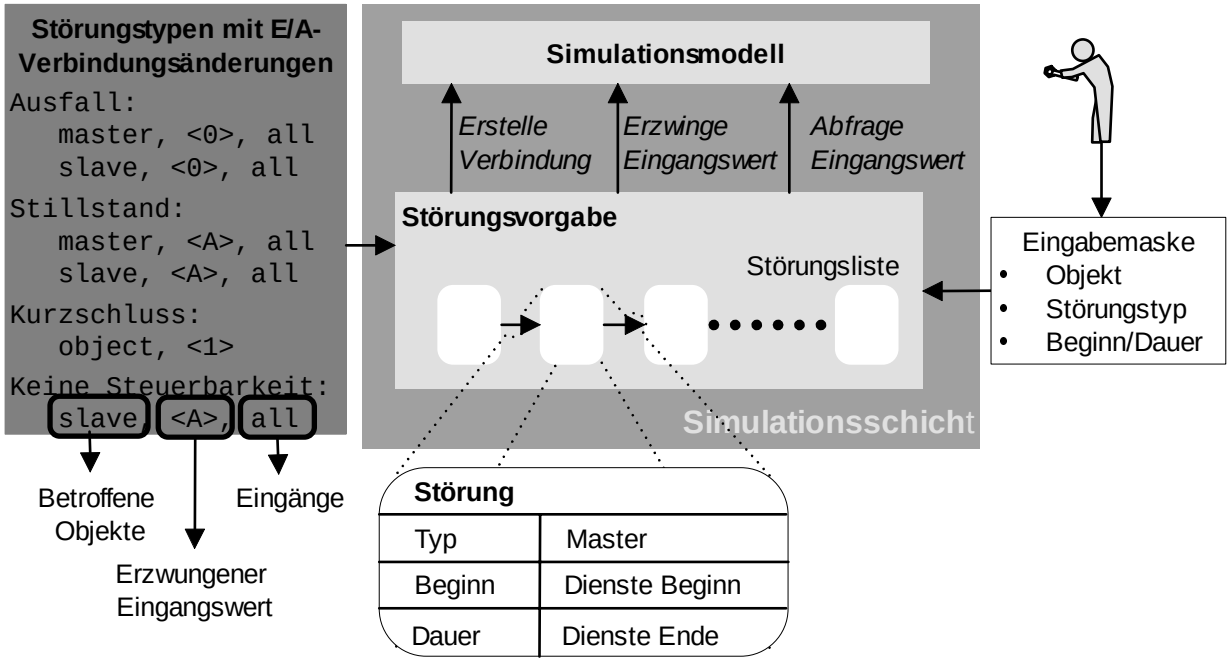


Bild 8.5: Funktionsweise der Störungsvorgabe

die (inkl. Beginn und Dauer) gestört sein sollen. Diese Störungen trägt die Störungsvorgabe in die Störungsliste ein. Jede Störung besteht aus ihrem Typ, dem Zeitpunkt des Störungsbeginns, der Störungsdauer und dem Master-Objekt. Außerdem enthält eine Störung jeweils eine Liste an Diensten, die sie jeweils bei Beginn und Ende einer Störung aufrufen muss. Die möglichen Dienste sind: Abfragen eines aktuellen Eingangswerts, Erzwingen eines Eingangswerts und das Erstellen einer E/A-Verbindung. Ein explizites Lösen einer Verbindung ist nicht nötig, weil dies implizit beim Aufruf des Diensts *Erzwinge Eingangswert* geschieht.

Die möglichen Störungstypen sind außerhalb der Störungsvorgabe in einer eigenen Datei eingetragen. Diese Datei enthält für jeden Störungstyp eine Liste an Anweisungen, die festlegen, was und wie die Störungsvorgabe beim Störungsbeginn im Simulationsmodell ändern muss. Jede Anweisung legt fest, welche Eingänge (alle oder einen speziellen Eingang) welcher Objekte (Master, Slave oder Objekt) die Störungsvorgabe auf welchen Eingangswert (<0> Null, <1> Eins oder <A> aktuellen Wert) erzwingen muss. Der Vorteil, der sich durch die Modellierung der Störungstypen in Dateien ergibt, besteht darin, dass neue Störungstypen modellierbar sind, ohne die Störungsvorgabe selbst zu ändern.

8.3 Rückkopplung von Simulationsergebnissen

Durch die Ausführung der Roboterprogramme in der Simulationsschicht entstehen Laufzeitinformationen (z. B. Ausführungszeit), die man für eine Bewertung der Roboterprogramme und damit der Planungsergebnisse heranziehen kann (innere Rückkopplung, Bild 4.1, S. 37). Um eine Verbesserung zu erzielen, kann der CAR-Anwender mit dem in dieser Arbeit entwickelten System viele Varianten von Roboterprogrammen nach dem Trial-and-Error-Prinzip ausprobieren und aus den untersuchten

```
   PROGRAMM Name                    ; Programmkopfzeile
   ...                              ; Variablendeklarationen
   ANFANG                           ; Programmstart
A  sys_zeit(StartZeit)              ; Lesen Sys.zeit bei Start
   ...                              ; Bewegungen und dig. E/A
B  sys_zeit(EndeZeit)               ; Lesen Sys.zeit bei Ende
C  Gesamtzeit=EndeZeit-StartZeit    ; Berechnung der Gesamtzeit
   PROGRAMM_ENDE                    ; Programmende
```

Bild 8.6: Befehle zur Messung der Ausführungszeit eines BAPS-Programms

Varianten diejenige mit der besten Bewertung bestimmen. Er kann aber auch Optimierungsverfahren einsetzen, die versuchen in ein (lokales) Minimum einer Zielfunktion abzusteigen.

8.3.1 Messung der Ausführungszeit

Da in der Praxis die Ausführungszeit eines Roboters als Zielfunktion zur Optimierung verwendet wird, wird in diesem Abschnitt deren Ermittlung näher betrachtet. Dazu ist die Ausführung eines steuerungsspezifischen Roboterprogramms mithilfe einer virtuellen Robotersteuerung erforderlich. In dieser Arbeit kommen einerseits die Standardsteuerung von COSIMIR® (KEIBEL [84]) zum Einsatz (Möglichkeit 1, Bild 4.9, S. 49). Andererseits wird für Roboterprogramme in der Syntax BAPS die Robotersteuerung Bosch rho4 verwendet, die BAUER [17] über OPC an COSIMIR® angebunden hat (Möglichkeit 3, Bild 4.9, S. 49).

Die Abfrage der Ausführungszeit bei der Standardsteuerung von COSIMIR® stellt kein Problem dar, weil diese Steuerung über eine COSIMIR®-interne Schnittstelle verfügt, über diese die Ausführungszeit ermittelbar ist [54]. Die Kommunikationszeit zwischen COSIMIR® und deren Standardsteuerung ist aufgrund dieser internen Schnittstelle vernachlässigbar.

Zur Messung der Ausführungszeit der BAPS-Programme wird die Systemuhr der Robotersteuerung Bosch rho4 verwendet (Bild 8.6). Dazu werden jeweils am Anfang (Zeile A) und am Ende des BAPS-Programms (Zeile B) Befehle zur Abfrage der internen Systemzeit und zur Berechnung der Ausführungszeit (Zeile C) generiert. Es ergibt sich die gemessene Ausführungszeit t_{Ausf} in der Variablen `Gesamtzeit`. Die Anweisungen zur Zeitmessung wurden in die Syntaxdefinition von BAPS aufgenommen, d. h. es war keine Änderung des Moduls "Programmsynthese" erforderlich.

Ein Problem bei der Bestimmung der Ausführungszeit eines Roboterprogramms auf einer realen Robotersteuerung, die an ein Simulationssystem gekoppelt ist, ist die Interaktion mit der simulierten Peripherie. Beispielsweise ist die Ansteuerung eines simulierten Greifers über digitale E/A erforderlich. Dabei wird der Greifer über den digitalen Ausgang `GreiferZu` geöffnet (`GreiferZu=0`) und geschlossen (`GreiferZu=1`). Bevor der Roboter nach dem Setzen oder Rücksetzen dieses digitalen Ausgangs seine Bewegung fortsetzt, muss er auf eine Rückmeldung vom Greifer bezüglich seines Zustand über den digitalen Eingang `GreiferIstZu` warten. Demnach müssen für das Schließen folgende BAPS-Programmbefehle zur Kommunikation mit dem Greifer auftauchen. Entsprechendes gilt auch für das Öffnen des Greifers:

```
GreiferZu=1               ; Ansteuerung: Greifer schliessen
WARTE BIS GreiferIstZu=1  ; Warten bis Greifer geschlossen ist
```

Hierbei entsteht folgendes Problem (Bild 8.7): Die Steuerung rho4 stellt ihre digitalen Ein- und Ausgangswerte als OPC-Items über ihren OPC-Server zur Verfügung. Das System COSIMIR® besitzt einen OPC-Client, der Nachrichten über Änderungen dieser OPC-Items erhält, und aktualisiert bei Empfang von Änderungen die internen Ein- und Ausgänge des Simulationsmodells (Bild 4.8, S. 48). Die Zeit t_W, die der Befehl `WARTE` bis zur Programmfortsetzung benötigt, setzt sich also zusammen aus der Zeit t_K zur Kommunikation mit dem Simulationssystem und der Zeit t_G für das Öffnen oder Schließen des Greifers. Es gilt:

$$t_W = t_K + t_G = t_{K_1} + t_{K_2} + t_G \tag{8.1}$$

In einem Roboterprogramm finden n Interaktionen ($n \geq 0$) mit der Peripherie statt. Die Zeit $t_{W,i}$ bezeichnet die Zeit t_W für den Interaktionsvorgang i und $t_{G,i}$ bezeichnet die Zeit t_G für den Vorgang i. Die Summe aller Zeiten $t_{W,i}$ wird als T_W bezeichnet. Es gilt:

$$T_W = \sum_{i=1}^{n} t_{W,i} = \sum_{i=1}^{n} (t_{K,i} + t_{G,i}) = \sum_{i=1}^{n} (t_{K_1,i} + t_{K_2,i} + t_{G,i}) \tag{8.2}$$

Das Problem entsteht durch Schwankungen in der Zeit $t_{K,i}$, die durch eine nicht echtzeitfähige Kommunikation (in diesem Fall OPC) verursacht wird. Folglich entstehen Schwankungen in der Zeit $t_{W,i}$ und damit auch in der gemessenen Ausführungszeit t_{Ausf}. Dies hat wiederum zur Folge, dass die Ausführungszeiten von zwei unterschiedlichen Roboterprogrammen bezüglich t_{Ausf} nicht mehr vergleichbar sind. Eine Optimierung ist mit dem Kriterium t_{Ausf} somit nicht möglich.

Um die Optimierung trotz der Schwankungen in t_{Ausf} zu ermöglichen, müssen diese beseitigt werden. Das bedeutet, dass mehrere Messungen der Ausführungszeit t_{Ausf} eines Roboterprogramms innerhalb einer kleinen Schwankungsbreite denselben Wert ergeben müssen. Um dies zu erreichen, gibt es zwei Ansätze, die beide auf der Überlegung beruhen, den konstanten Anteil t^C_{Ausf} der Ausführungszeit des Roboterprogramms zu bestimmen. Beide Ansätze gehen von der in der Praxis vorhandenen Annahme aus, dass jedes Öffnen und jedes Schließen des Greifers in einer realen Roboter-Fertigungszelle immer gleich lang dauert. Es gilt:

$$t_{Ausf} = t^C_{Ausf} + T_W \tag{8.3}$$

Dieser konstante Anteil t^C_{Ausf} erfasst die Ausführung aller Roboterprogrammbefehle (z. B. Bewegungsbefehle, E/A-Befehle) mit Ausnahme des Befehls zum Warten auf die Rückmeldung der Peripherie. Die Ansätze lassen sich aus Formel 8.3 ableiten und lauten:

1. *Peripherie ignorieren*: Es wird $T_W = 0$ angenommen und der Befehl `WARTE` für alle Interaktionen weggelassen.

2. *Messen der Zeit* T_W: Die Zeit $t_{W,i}$ wird für jede Interaktion i gemessen und die Summe T_W von der Ausführungszeit t_{Ausf} abgezogen.

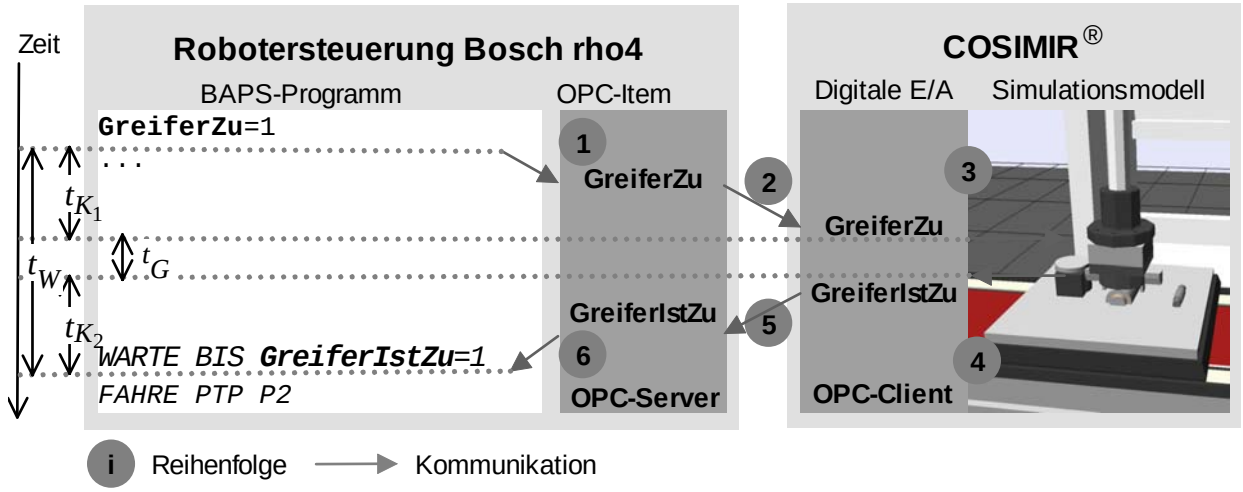


Bild 8.7: Kommunikationsvorgänge zur Greiferansteuerung

```
   ANFANG                                    ; Programmstart
A  SummeTw=0                                 ; Initialisierung Summe Tw
   sys_zeit(StartZeit)                       ; Lesen Sys.Zeit bei Start
   ...
   GreiferZu=1                               ; Greiferansteuerung
B  sys_zeit(TwVorWarte)                      ; Lesen Sys.Zeit vor Rückmeldung
   WARTE BIS GreiferIstZu=1                  ; Warten auf Rückmeldung
C  sys_zeit(TwNachWarte)                     ; Lesen Sys.Zeit nach Rückmeldung
D  SummeTw=SummeTw+TwNachWarte-TwVorWarte    ; Summieren aller Tw
   ...
   sys_zeit(EndeZeit)                        ; Lesen Sys.Zeit bei Start
   Gesamtzeit=EndeZeit-StartZeit-SummeTw E   ; Korrektur Gesamtzeit
   PROGRAMM_ENDE
```

Bild 8.8: Ergänzende Programmbefehle für den Ansatz *Messen der Zeit* T_W

Der erste Ansatz beseitigt die Schwankungen in der Ausführungszeit t_{Ausf} dadurch, dass er den für die Schwankung verantwortlichen Befehl `WARTE` weglässt. Ein Nachteil des ersten Ansatzes besteht darin, dass Inkonsistenzen im Simulationsmodell entstehen können. Z. B. kann es vorkommen, dass das Simulationssystem das Öffnen und Schließen des Greifers zu spät erkennt. Für manche Applikation sind solche Inkonsistenzen unwichtig, weil dadurch die Roboterbewegung nicht beeinflusst wird. Bei Applikationen, die z. B. Sensorik verwenden, sind solche Inkonsistenzen nicht akzeptabel.

Der zweite Ansatz *Messen der Zeit* T_W (Bild 8.8) liest vor und nach jedem Befehl `WARTE` (Zeile B, C) die Systemzeit der Steuerung aus, berechnet die Zeit $t_{W,i}$ als Differenz der beiden ausgelesenen Systemzeiten und addiert diese Differenz zur Variablen `SummeTw` (Zeile D). Die Summe T_W aller Zeiten $t_{W,i}$ wird am Programmende von der gemessenen Ausführungszeit t_{Ausf} abgezogen (Zeile E). Der Vorteil des zweiten Ansatzes besteht darin, dass die oben genannten Inkonsistenzen des Simulationsmodells nicht entstehen können, da auf die Rückmeldung vom Greifer gewartet wird. Ein Nachteil besteht darin, dass die gemessene Ausführungszeit t_{Ausf} wegen der zusätzlichen Programmbefehle zum Messen der Zeit t_W geringfügig erhöht ist ($\approx$ 1 bis 2 ms pro Interaktion mit dem Greifer).

Die Messergebnisse aus Tabelle 8.3 bestätigen die Wirksamkeit der beiden Ansätze. Darin sind die Ausführungszeiten t_{Ausf} bzw. t^C_{Ausf} angegeben, die sich durch den Einsatz der beiden Ansätze zur Vermeidung der Schwankungen in T_W ergeben. Für jeden Fall sind fünf Messungen aufgeführt. Das ausgeführte BAPS-Programm enthält insgesamt sechs Interaktionen mit einem Greifer (drei Mal Schließen, drei Mal Öffnen). Im ersten Fall in Tabelle 8.3 erkennt man das Problem, dass die Ausführungszeit t_{Ausf} schwankt, wenn keiner der beiden Ansätze zu deren Vermeidung eingesetzt wird. Im zweiten Fall sieht man die Wirksamkeit des Ansatzes *Peripherie ignorieren*, weil t^C_{Ausf} nahezu konstant ist. Im dritten Fall zeigt sich die Wirksamkeit des zweiten Ansatzes *Messen der Zeit* T_W, weil auch hier t^C_{Ausf} nahezu konstant ist. Im zweiten und dritten Fall treten so gut wie keine Schwankungen in t^C_{Ausf} auf. Man erkennt, dass bei Verwendung des Ansatzes *Messen der Zeit* T_W, die Ausführungszeit t^C_{Ausf} höher liegt als beim Ansatz *Peripherie ignorieren*. Dies liegt an den zusätzlichen Programmbefehlen zur Messung der Zeit T_W. Zur Vervollständigung aller Fälle wurden im vierten Fall fünf Messungen durchgeführt, bei denen der Befehl `WARTE` weggelassen wurde und trotzdem die Befehle zur Messung der Zeit T_W enthalten waren. Hier erkennt man, dass der Ansatz *Messen der Zeit* T_W, die Schwankungen in T_W gut erfasst, da die Werte von t^C_{Ausf} im dritten und vierten Fall in etwa gleich sind.

Tabelle 8.3: Ergebnisse bei unterschiedlichen Anssätzen zur Messung der Ausführungszeit t_W

	Fall 1		**Fall 2**		**Fall 3**		**Fall 4**	
Ansatz 1	**nein**		**ja**		**nein**		**ja**	
Ansatz 2	**nein**		**nein**		**ja**		**ja**	
Nr.	T_W	t_{Ausf}	T_W	t^C_{Ausf}	T_W	t^C_{Ausf}	T_W	t^C_{Ausf}
1	0	9304	0	7400	1614	7424	6	7424
2	0	9674	0	7394	2064	7424	12	7424
3	0	9104	0	7400	1644	7430	6	7430
4	0	9350	0	7400	1784	7430	6	7424
5	0	8192	0	7400	2052	7430	6	7430

Es kann festgehalten werden, dass unter der Voraussetzung, dass eine Interaktion mit der Roboterperipherie erforderlich ist, die Bestimmung der absoluten Werte für die Ausführungszeiten t_{Ausf} trotz des Einsatzes einer realen Robotersteuerung aufgrund von Schwankungen problematisch ist. Zur Bewertung von planerischen Varianten (z. B. Trägeranordnung) genügt eine korrigierte Ausführungszeit t^C_{Ausf}, in der die Schwankungen beseitigt sind. Erst dadurch werden Planungsvarianten vergleichbar.

8.3.2 Zielfunktion

Für die Optimierung einer Zielfunktion $F(\overrightarrow{x})$, d. h. die Bewertung eines Simulationslaufs, muss der CAR-Anwender eine solche Funktion in den Plänen angeben (Schritt *Bewertung*, Bild 5.15, S. 74). Der Vektor $\overrightarrow{x}$ legt die Größen fest, die die Roboterprogramme, das Simulationsmodell oder beides variieren. In der Schablone zur Layoutoptimierung ist z. B. als Standard für die Layoutbewertung die Ausführungszeit des Simulationslaufs vorgesehen. Mit einem Optimierungsverfahren berechnet ein

Plan ein (lokales) Optimum $F(\overrightarrow{x}^*)$ an der Stelle $\overrightarrow{x}^*$. Das Ergebnis ist ein optimiertes Roboterprogramm oder Planungsergebnis bezüglich der vom CAR-Anwender angegebenen Zielfunktion.

Ein Optimierungsproblem, das man mit diesem System lösen kann, lautet allgemein:

Minimiere $F(\overrightarrow{x})$
u. d. N. Fehlerfreie Ausführbarkeit des Simulationslaufs

Charakteristisch für diese Optimierungsprobleme ist, dass zur Berechnung der Zielfunktionswerte aus dem zulässigen Bereich M keine analytische Berechnung der Zielfunktion $F(\overrightarrow{x})$ und der Nebenbedingungen existiert, sondern dies ein Simulationslauf erfordert. Dabei muss der Simulationslauf fehlerfrei bezüglich denen in Abschnitt 8.2.1 aufgeführten Fehlerarten sein. Die Besonderheit des Systems ist, dass der Simulationslauf auf realen, steuerungsspezifischen Roboterprogrammen beruht. Insgesamt verursacht die Berechnung eines Zielfunktionswerts dadurch einen hohen Aufwand.

Betrachtet man die Layoutoptimierung, so sind deren Zielfunktionen in der Regel nichtlinear, nicht stetig und folglich auch nicht differenzierbar [145, 170]. Diese Aussagen bestätigt die in Bild 8.9 dargestellte Zielfunktion für die Optimierung der Poliermaschinenposition aus dem einleitenden Beispiel (Bild 1.1, S. 2). Der zulässige Bereich M ist beschränkt und die Zielfunktion enthält lokale Minima. Beispiele für Zielfunktionen zur Bewertung eines Simulationslaufs sind:

1. Ausführungszeit des Roboters [50]
2. Summe oder Maximum der Achsvariablendifferenzen [145, 170]
3. Räumliche Ausdehnung der Roboter-Fertigungszelle [48]
4. Prozessergebnis [135]

8.3.3 Verfahrensauswahl

Für die Optimierung sind geeignete Verfahren auszuwählen. Dabei sind die genannten Eigenschaften der Zielfunktion zu berücksichtigen. Für diese numerischen, mehrdimensionalen und nichtlinearen Parameteroptimierungsprobleme existieren viele Verfahren, die man wie folgt einteilen kann:

1. Suchverfahren (z. B. HOOKE-JEEVES [72])
2. Gradientenverfahren (z. B. FLETCHER-REEVES [43])
3. Newtonverfahren [113].

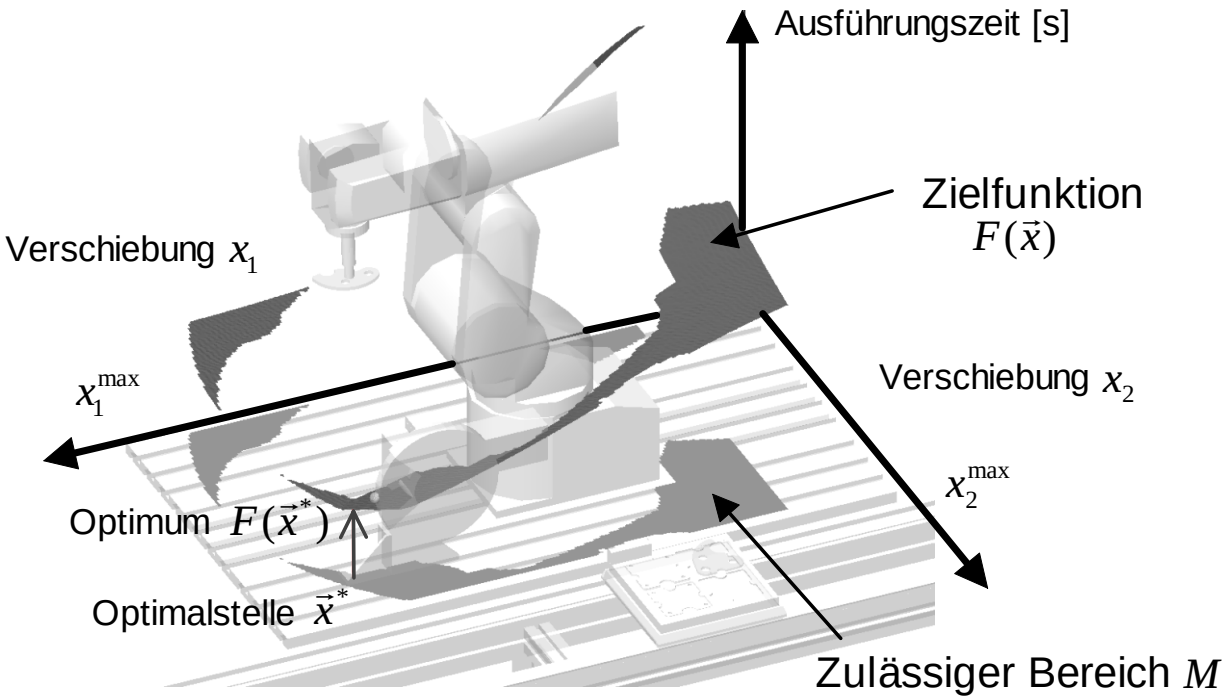


Bild 8.9: Zielfunktion "Ausführungszeit" bei Variation der Poliermaschine auf der Grundplatte für Produktvariante C: $F(\vec{x}) = Ausführungszeit(x_1, x_2)$

Gradientenverfahren und Newtonverfahren benötigen eine Ableitung der Zielfunktion. Weil zur Berechnung eines Funktionswertes der in dieser Arbeit betrachteten Zielfunktionen ein Simulationslauf notwendig ist, existiert keine analytische Form der Ableitung. Die numerische Berechnung der Ableitung führt zu einer Erhöhung der Anzahl der zu berechnenden Zielfunktionswerte, deren jeweilige Berechnung aufwendig ist. Aus diesen Gründen scheiden Gradientenverfahren und Newtonverfahren zur Lösung aus. Die Suchverfahren hingegen benötigen ausschließlich die Berechnung der Zielfunktion, nicht aber deren Ableitung(en).

SCHWINN [145] und WOENCKHAUS [170] haben verschiedene Verfahren für die Layoutoptimierung untersucht. Deren Untersuchungen ergaben, dass jedes analysierte Verfahren bessere Ergebnisse liefert, wenn die Suche von mehreren im Definitionsbereich verteilten Punkten gestartet wird. Aus diesem Grund wird in der Schablone zur Layoutoptimierung das Optimierungsverfahren ausgehend von verschiedenen Standorten (Formel 5.3, S. 72) angewendet.

Des Weiteren zeigen diese Untersuchungen, dass es nicht das beste Verfahren bezüglich Ergebnisgüte, Rechenzeit und Konvergenzgeschwindigkeit gibt. Deshalb stehen dem CAR-Anwender mehrere Verfahren zur Verfügung (u. a. Hooke-Jeeves, Simulated-Annealing), von denen er durch Setzen der Systemvariable `$STRATEGY` eines auswählen kann.

8.4 Dienste der Simulationsschicht

Die Simulationsschicht bietet Dienste, die die Funktionserweiterungen der Generierungssteuerung der überlagerten Generierungsschicht aufrufen. Die Simulationsschicht wird in dieser Arbeit von

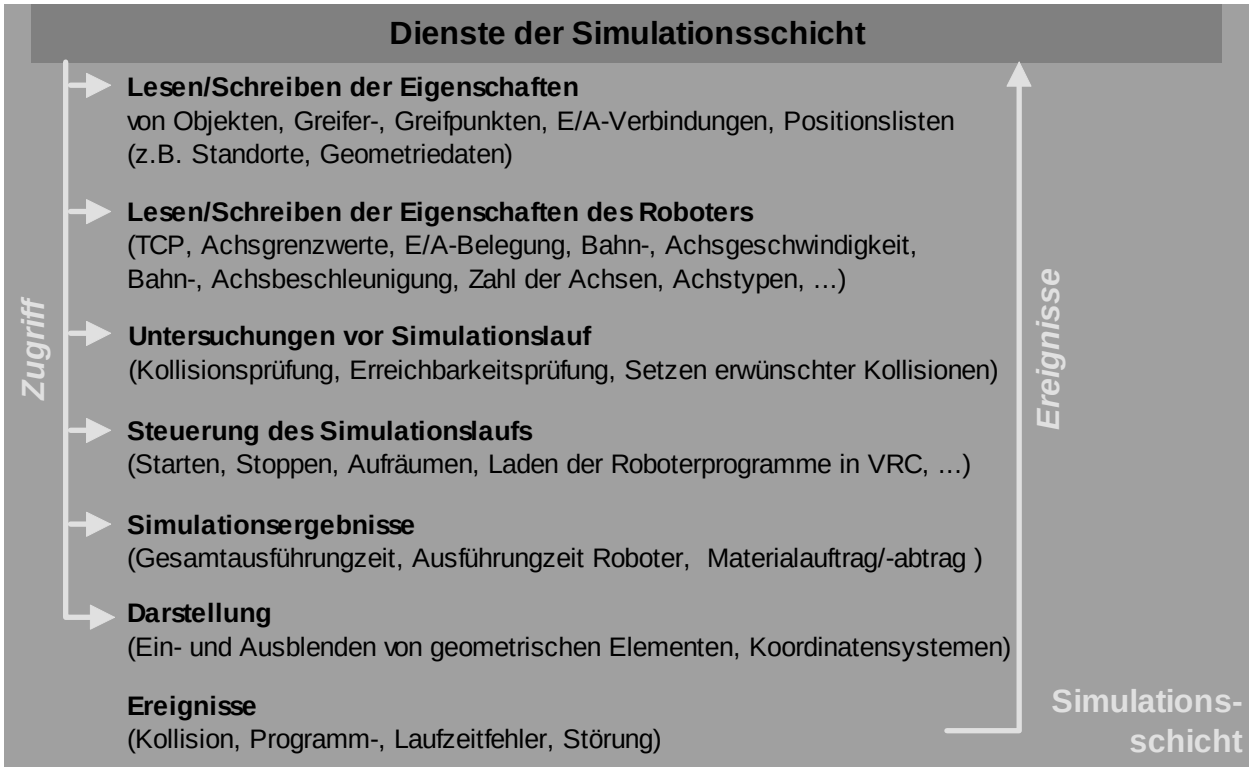


Bild 8.10: Dienste der Simulationsschicht

COSIMIR® gebildet, das die Rolle des Diensterbringers besitzt. Aufgrund des Dienstumfangs werden hier Gruppen gebildet, die im Folgenden kurz erläutert werden (Bild 8.10).

Es besteht die Möglichkeit, die Eigenschaften der Objekte im Simulationsmodell auszulesen und zu schreiben. Dazu zählen die Standorte der Objekte, ihre Geometrie, deren E/A-Verbindungen. Die Simulationsschicht erlaubt es, Greif- und Greiferpunkte zu erzeugen, auszulesen, zu verbinden und zu lösen. Außerdem besteht ein Zugriff auf vordefinierte Bahnstützpunktlisten für die Roboterbewegung.

Weil der Roboter im Simulationsmodell eine besondere Rolle spielt, werden die Dienste zum Lesen und Schreiben der Robotereigenschaften zusammengefasst. Die generierten Programmen benötigten diese Informationen, z. B. TCP. Außerdem werden diese Informationen, z. B. Zahl der Achsen, dazu verwendet, zu entscheiden, welche Syntaxdefinitionen die Programmsynthese einlesen soll.

Um möglichst viele Fehler vor einem Simulationslauf zu finden, stehen Dienste zur Kollisions- und Erreichbarkeitsprüfung zur Verfügung. Des Weiteren kann man den Simulationslauf selbst steuern und nach dessen Ende Ergebnisse abfragen. Da die Pläne zur Programmgenerierung für variantenreiche Produkte in der Regel geometrische Berechnungen für die Bahnstützpunkte enthalten, werden Dienste zur Verfügung gestellt, die es erlauben geometrische Elemente oder Koordinatensystem ein- und auszublenden. Dadurch erhält der CAR-Anwender eine Kontrolle über seine Berechnungen.

Alle bisher beschriebenen Dienste können im erweiterten Sinne als Zugriff auf das Simulationsmodell betrachtet werden. Darüber hinaus sendet die Simulationsschicht Ereignisse an die überlagerte Generierungsschicht, um diese über besondere Vorkommnisse während des Simulationslaufs zu informieren. Dadurch werden die Simulationsläufe bei Auftritt eines Fehlers nicht unnötig verlängert.

9 Applikationen und Ergebnisse

Dieses Kapitel beschreibt Applikationen, die die unterschiedlichen Einsatzmöglichkeiten des in dieser Arbeit entwickelten Systems aufzeigen. Dabei wird der Einsatz des Systems zur Erzeugung von Roboterprogrammen für Produktvarianten erläutert. Ebenso wird der Einsatz des Systems zur Beantwortung von Fragen, die bei der Planung von Roboter-Fertigungszellen auftauchen, dargestellt (Layoutoptimierung, Bestimmung von Reihenfolgen, Auswahl von Fertigungskomponenten).

Der Einsatz des Systems zur Automatisierung des Programmierung dient zur Aufwandreduktion bei zahlreichen Produktvarianten. Insbesondere wenn die Programmierung einer Produktvariante aufwendig ist, macht sich das System bezahlt, wie nachfolgende Applikationen verdeutlichen.

9.1 Programmgenerierung für die zementorientierte Hüftendoprothetik

In Zusammenarbeit mit der Klinik für Orthopädie und Rheumatologie der Philipps-Universität Marburg wurde ein Prototyp entwickelt, um Vorversuche den Robotereinsatz zur zementorientierten Hüftendoprothetik durchzuführen. Durch den Einsatz des in dieser Arbeit entwickelten Systems wird es für einen Mediziner möglich, die für die Versuche notwendigen Roboterprogramme ohne umfangreiche Roboterprogrammierkenntnisse zu erzeugen.

Motivation

Folgende Schritte stellen den Verlauf einer zementorientierten Hüftoperation verkürzt dar (Bild 9.1):

(a) Bestimmen der Schnittebene am Oberschenkelknochen

(b) Abschneiden und entfernen des Hüftkopfs

(c) Sichtbarwerden der Spongiosa des Oberschenkelknochens

(d) Einbringen der Raspel in die Spongiosa

(e) Manuelle Vor- und Zurückbewegung der Raspel in der Spongiosa

(f) Raspel entnehmen und füllen des entstandenen Hohlraums mit Zement

(g) Einführen des Prothesenschafts in den Zement

Durch die Raspelbewegung in der Spongiosa (schwammartige innere Knochenstruktur), das anschließende Füllen des entstandenen Hohlraums mit Zement und das abschließende Einführen des Prothesenschafts in den Zement, bildet sich ein Zementmantel um den Prothesenschaft. Eine definierte,

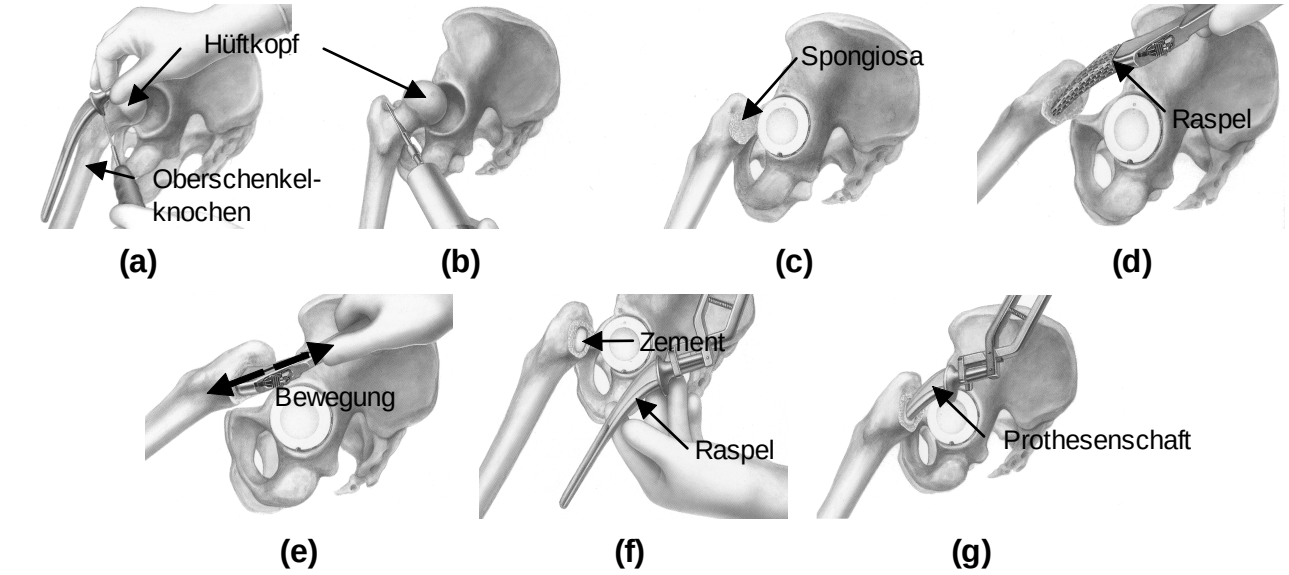


Bild 9.1: Verlauf bei der zementorientierten Hüftoperation (Quelle: W. Link GmbH [165])

gleichmäßige Zementmanteldicke (2-3 mm) ist für den Patienten von großem Vorteil, da dann der Sitz der Prothese am besten ist. Die Gleichmäßigkeit der Zementmanteldicke ist mit der heutigen manuellen Vorgehensweise (Bild 9.1e) nicht zu erreichen. Aus diesem Grund soll ein Roboter nach der Raspelentnahme den Hohlraum in die Spongiosa für den Zementmantel fräsen, damit sich ein Zementmantel mit einer gleichmäßigen, definierten Dicke um den Prothesenschaft bilden kann (Bild 9.2). Der Operationsverlauf mit Robotereinsatz ist, verglichen mit einer Operation ohne Robotereinsatz, nahezu identisch mit Ausnahme der Zeitspanne zwischen der Raspelentnahme und dem Einfüllen des Zements. Der Roboter soll in dieser Zeitspanne statt der manuellen Raspelbewegung (Bild 9.1e) den durch die Raspel entstandenen Hohlraum um 2-3 mm vergrößern.

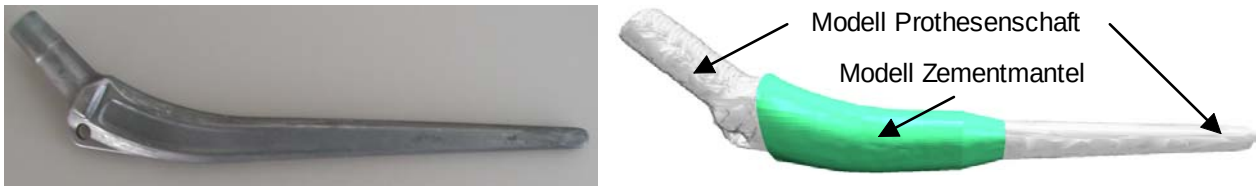
Der bisherige Robotereinsatz in der Hüftendoprothetik (CASPAR [123], ROBODOC [95]) konzentrierte sich ausschließlich auf die zementlose Hüftendoprothetik, obwohl in Europa 70 % aller Operationen für Hüftprothesen zementorientiert sind.

In medizinischen Fachkreisen besteht Uneinigkeit darüber, ob die zementlose Hüftendoprothetik, die nur durch den Robotereinsatz möglich ist, dem Patient einen Vorteil bringt oder ihm eher schadet [62, 95, 159]. Die Vorteile beim Robotereinsatz in der zementorientierten Hüftendoprothetik bestehen im Gegensatz zum Robotereinsatz bei der zementlosen Hüftendoprothetik darin, dass einerseits bei einem Roboterausfall die Operation wie bisher fortgesetzt werden kann und andererseits die Genauigkeitsanforderungen geringer sind.

Aus folgenden Gründen ist es sinnvoll, die Roboterprogramme für den Fräsvorgang zu generieren:

1. Für die bei der zementorientierten Hüftendoprothetik eingesetzten Prothesenschäfte existieren mehrere hundert Varianten.

2. Es werden auch in Zukunft Veränderungen an der Schaftgeometrie vorgenommen.



(a) Prothesenschaft **(b)** Geometriemodelle

Bild 9.2: Prothesenschaft und zugehörige Geometriemodelle mit Zementmantel

3. Die Fräsparameterwerte (z. B. Vorschub und Zustellung [58]) sind unbekannt und müssen durch Versuche ermittelt werden.

4. Der Fräsvorgang für eine Prothesenvariante erfordert mehrere tausend Roboterpositionen.

Insgesamt bedeutet dies, dass für jeden Prothesenschaft und für jede Änderung an den Fräsparameterwerten ein neues Roboterprogramm notwendig ist.

Realisierung

Zur Durchführung der Vorversuche für den Robotereinsatz in der zementorientierten Hüftendoprothetik wurden ein Roboter KUKA KR-6 und die Fräse des Systems CASPAR [123] der Firma U.R.S.-Ortho eingesetzt (Bild 9.3).

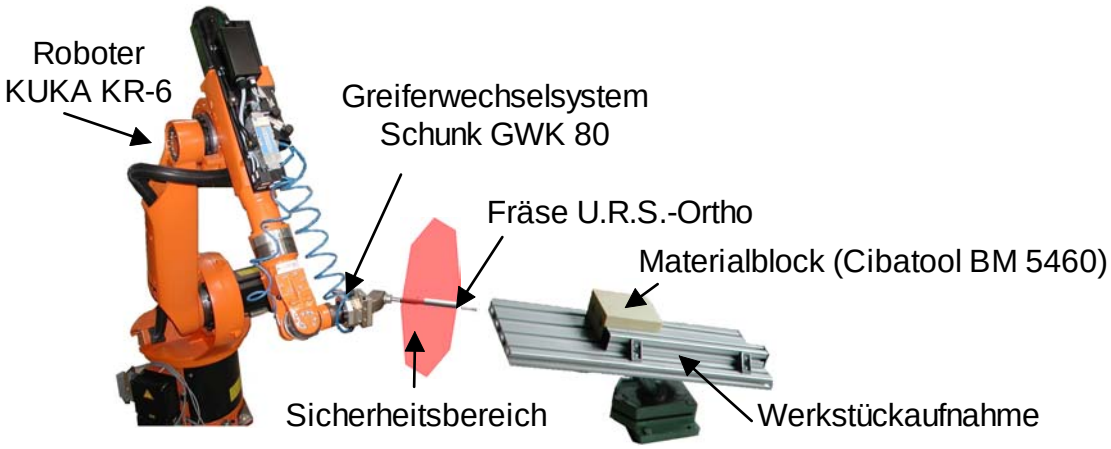


Bild 9.3: Aufbau zur Durchführung der Versuche

Die Programmgenerierung erfordert ein Geometriemodell des Prothesenschafts und des Zementmantels. Für Ersteres können dreidimensionale CAD-Daten des Herstellers zum Einsatz kommen. Falls die CAD-Daten, wie in diesem Fall, nicht verfügbar sind, lässt sich das Geometriemodell aus Computertomografiedaten (CT) berechnen [166]. Aus dem Geometriemodell des Prothesenschafts lässt sich, wie im Folgenden erläutert, das Geometriemodell des Zementmantels berechnen.

Mithilfe der Schablone zur Generierung variantenreicher Produkte (Abschnitt 5.2.2, S. 64ff) wurde ein Generierungsplan implementiert, der als Eingangsdaten die Fräsparameterwerte vom Anwender

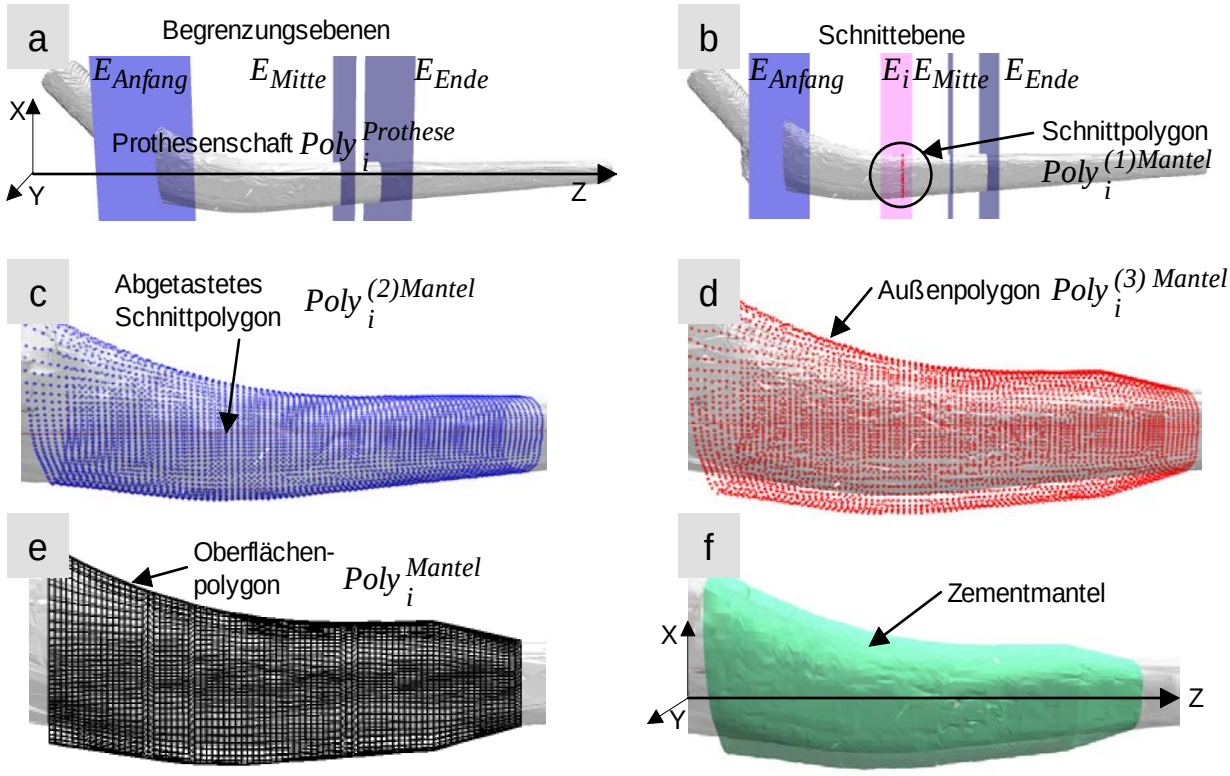


Bild 9.4: Schritte zur Erstellung des Zementmantelmodells

und das Geometriemodell des Prothesenschafts benötigt. Daraus erzeugt der Plan das Geometriemodell des Zementmantels und führt eine Fräsbahnberechnung durch. Das Ergebnis einer Planausführung ist ein KRL-Programm, das die Fräsbewegung des Roboters und die über digitale Ausgänge verbundene Fräse steuert. Im Folgenden werden die Berechnungen, die dieser Plan enthält, erläutert: Erzeugung des Geometriemodells des Zementmantels und Fräsbahnberechnung.

Erzeugung des Geometriemodells für den Zementmantel

Die Geometriemodelle des Prothesenschafts und des Mantels, d. h. deren Oberflächen, werden jeweils durch eine Polygonfolge $P^{Prothese} = \{Poly_i^{\mathrm{Prothese}} | 1 \leq i \leq N^{\mathrm{Prothese}}\}$ und $P^{Mantel} = \{Poly_i^{Mantel} | 1 \leq i \leq N^{Mantel}\}$ repräsentiert (Abschnitt 4.4.1, S. 47). Der aufgestellte Generierungsplan berechnet aus der Polygonfolge $P^{Prothese}$ und weiteren Eingangsparametern die Polygonfolge P^{Mantel} über Zwischenschritte, deren Ergebnis jeweils eine Polygonfolge $P^{(k)Mantel} = \{Poly_i^{(k)Mantel}\}_{i=1}^{N^{(k)Mantel}}$ ist. Bild 9.4 zeigt die im Folgenden erläuterten Zwischenschritte.

Die Parameter, die neben dem Prothesenmodell den Zementmantel definieren, gibt der Mediziner vor:

- Zementmanteldicke d_{Mantel}
- Begrenzungsebenen E_{Anfang}, E_{Mitte} und E_{Ende}
- Schnittebenenabstand d_S^{Mantel}

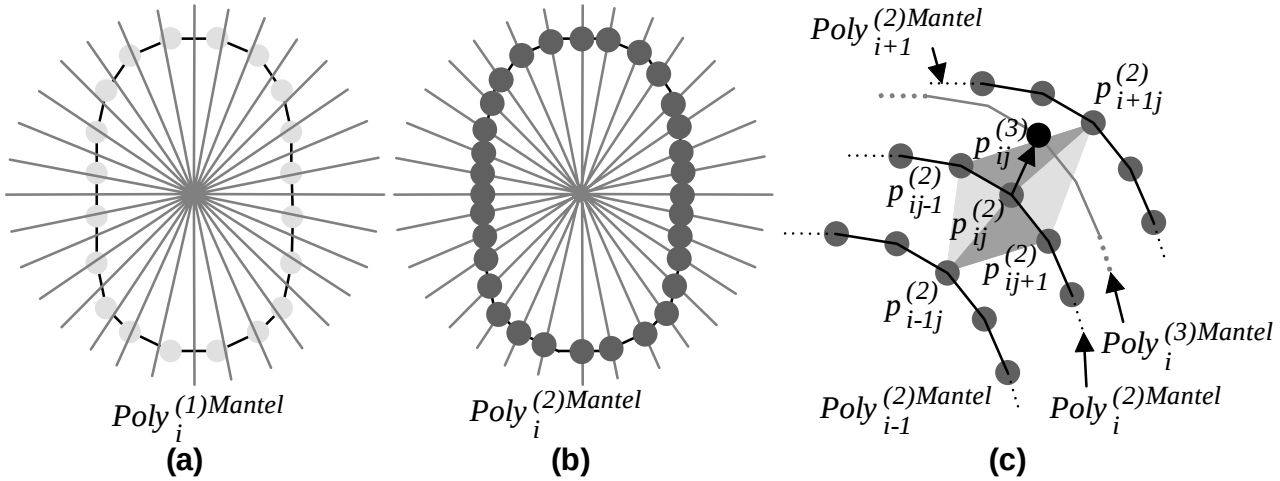


Bild 9.5: Zwischenschritte $Poly_i^{(1)Mantel}$ bis $Poly_i^{(3)Mantel}$

- Abtastwinkel α

Die Zementmanteldicke d_{Mantel} gibt den maximalen Abstand zwischen der gegebenen Oberfläche des Prothesenschafts und der zu erstellenden Oberfläche des Zementmantels an. Der Zementmantel soll zwischen den Ebenen E_{Anfang} und E_{Mitte} die maximale Zementmanteldicke d_{Mantel} besitzen. Der Zementmantel soll zwischen den Begrenzungsebenen E_{Mitte} und E_{Ende} linear von d_{Mantel} auf 0 verlaufen. Dabei zeigt der Normalenvektor der Begrenzungsebenen in die gleiche Richtung wie die Z-Achse des Geometriemodells des Prothesenschafts (Bild 9.4a). Der Abstand $d_{Anfang,Ende}$ ist der Abstand zwischen E_{Anfang} und E_{Ende} und $d_{Ende,Mitte}$ der Abstand zwischen E_{Mitte} und E_{Ende}.

Im Schnittebenenabstand d_S^{Mantel} berechnet der Plan eine Folge $E^{Mantel} = \{E_i^{Mantel}\}_{i=1}^{N_E}$ von äquidistanten Schnittebenen E_i^{Mantel} mit $N_E = \lceil \frac{d_{Anfang,Ende}}{d_s^{Mantel}} \rceil$. Der Plan schneidet E^{Mantel} mit $P^{Prothese}$, d. h. jedes E_i^{Mantel} wird mit jedem $Poly_i^{\mathrm{Prothese}}$ geschnitten (Bild 9.4b). Dazu steht die Systemroutine `$CUT_PLANE_POLYGONS` zur Verfügung (Anhang B, S. 171). Das Ergebnis ist eine Polygonfolge $P^{(1)Mantel} = \{Poly_i^{(1)Mantel}\}_{i=1}^{N_E}$ (Bild 9.5a). Jedes Polygon $Poly_i^{(1)Mantel}$ wird durch eine zyklische Folge $\{p_{ij}^{(1)Mantel}\}_{j=1}^{N_i}$ von Punkten $p_{ij}^{(1)Mantel}$ beschrieben. $Poly_i^{(1)Mantel}$ und $Poly_j^{(1)Mantel}$ mit $i \neq j$ können eine unterschiedliche Anzahl von Polygonpunkten besitzen, d. h, es kann gelten $N_i \neq N_j$. Da aber für die weitere Konstruktion der Zementmanteloberfläche ein Punkt des Schnittpolygons $p_{ij}^{(1)Mantel}$ eindeutig je einem Punkt der benachbarten Schnittpolygonpunkt $p_{i-1j}^{(1)Mantel}$ und $p_{i+1j}^{(1)Mantel}$ zugeordnet werden muss, wird jedes Schnittpolygon $Poly_i^{(1)Mantel}$ im Abtastwinkel α um seinen Schwerpunkt abgetastet (Systemroutine `$SCAN_POLYGON`, Anhang B, S. 171). Das Ergebnis ist eine Polygonfolge $P^{(2)Mantel} = \{Poly_i^{(2)Mantel}\}_{i=1}^{N_E}$ (Bild 9.4c, Bild 9.5b), deren Polygone $Poly_i^{(2)Mantel}$ jeweils die gleiche Anzahl $N_\alpha = \frac{2\Pi}{\alpha}$ an Punkten $p_{ij}^{(2)Mantel}$ besitzen ($1 \leq j \leq \frac{2\Pi}{\alpha}$).

Im nächsten Schritt (Bild 9.4d) konstruiert der Plan aus der Polygonfolge $P^{(2)Mantel}$ die Polygonfolge $P^{(3)Mantel} = \{Poly_i^{(3)Mantel}\}_{i=1}^{N_E}$ mit $Poly_i^{(3)Mantel} = \{p_{ij}^{(3)Mantel}\}_{j=1}^{N_\alpha}$. Die Punkte $p_{ij}^{(3)Mantel}$ sollen auf der Zementmanteloberfläche liegen (Außenpunkte, Bild 9.4d). Dazu werden alle Punkte $p_{ij}^{(2)Mantel}$ der Polygonfolge $P^{(2)Mantel}$ durch jeweils einen Richtungsvektor $\overrightarrow{n}_{ij}^{Mantel}$ nach außen verschoben (Bild 9.5c). Es gilt demnach:

$$p_{ij}^{(3)Mantel} = p_{ij}^{(2)Mantel} + \overrightarrow{n}_{ij}^{Mantel} \text{ für alle } 1 \leq i \leq N_E \text{ und } 1 \leq j \leq N_\alpha \tag{9.1}$$

Der Betrag $\left\| \overrightarrow{n}_{ij}^{Mantel} \right\|$ hängt von der Lage des Punkts $p_{ij}^{(2)Mantel}$ ab. Liegt $p_{ij}^{(2)Mantel}$ zwischen E_{Anfang} und E_{Mitte}, so gilt: $\left\| \overrightarrow{n}_{ij}^{Mantel} \right\| = d_{Mantel}$. Liegt $p_{ij}^{(2)Mantel}$ zwischen E_{Mitte} und E_{Ende}, so nimmt der Betrag linear ab, je näher sich $p_{ij}^{(2)Mantel}$ der Begrenzungsebene E_{Ende} nähert. Für die Richtung von $\overrightarrow{n}_{ij}^{Mantel}$ werden vier Ebenen gebildet und deren Normalenvektoren gemittelt. Die erste Ebene wird durch die Punkte $p_{ij}^{(2)Mantel}$, $p_{i+1j}^{(2)Mantel}$ und $p_{ij+1}^{(2)Mantel}$ festgelegt. Die zweite Ebene wird durch die Punkte $p_{ij}^{(2)Mantel}$, $p_{ij+1}^{(2)Mantel}$ und $p_{i-1j}^{(2)Mantel}$ festgelegt. Die dritte Ebene wird durch die Punkte $p_{ij}^{(2)Mantel}$, $p_{i-1j}^{(2)Mantel}$ und $p_{ij-1}^{(2)Mantel}$ festgelegt und die vierte Ebene wird durch die Punkte $p_{ij}^{(2)Mantel}$, $p_{ij-1}^{(2)Mantel}$ und $p_{i+1j}^{(2)Mantel}$ festgelegt.

Im vorletzten Schritt (Bild 9.4e) wird das angestrebte Zementmantelmodell, d. h. dessen Oberfläche als Polygonfolge $P^{Mantel} = \{Poly_i^{Mantel} | 1 \leq i \leq N^{Mantel}\}$ berechnet. Dazu berechnet der Plan aus benachbarten Außenpunkten $p_{ij}^{(3)Mantel}$ die Oberflächenpolygone $Poly_i^{Mantel}$. Ein Oberflächenpolygon $Poly_i^{Mantel}$ setzt sich durch vier Außenpunkte als zyklische Folge $Poly_i^{Mantel} = \{ p_{ij}^{(3)Mantel}, p_{i+1j}^{(3)Mantel}, p_{i+1j+1}^{(3)Mantel}, p_{ij+1}^{(3)Mantel} \}$ zusammen. Im letzten Schritt (Bild 9.4f) erhält der Zementmantel Darstellungseigenschaften (Farbe, Transparenz).

Bahnberechnung für die Fräsbewegung

Für die Bahnberechnung der Fräsbewegung kann der Mediziner die folgenden Parameter einstellen:

- Vorschub d_f und Vorschubgeschwindigkeit v_f
- Zustellung d_z und Zustellgeschwindigkeit v_z
- Sicherheitspolygon $Poly^{\text{Si}\,cherheit}$ und Sicherheitsabstand $d^{\text{Si}\,cherheit}$

Die Zustellung d_z mit der Zustellgeschwindigkeit v_z definiert den Abtrag bei der Bewegung des Fräskopfs in Richtung der Z-Achse des Zementmantels (Bild 9.6). Der Vorschub d_f mit der Vorschubgeschwindigkeit v_f definiert den Abtrag bei der Bewegung des Fräskopfs in der XY-Ebene des Zementmantels tangential zu dessen Oberfläche (Bild 9.6). Das Polygon $Poly^{\text{Si}\,cherheit}$ soll die Bewegungsfreiheit der Fräse aus Sicherheitsgründen einschränken. Die Fräse darf sich nur innerhalb von $Poly^{\text{Si}\,cherheit}$ bewegen (Sicherheitsbereich, Bild 9.3). Der Abstand $d^{\text{Si}\,cherheit}$ legt fest, dass die Fräse mit Ausnahme des Kopfs beim Fräsvorgang den Abstand $d^{\text{Si}\,cherheit}$ zur Spongiosa einhält.

Die Bahnberechnung erfolgt in zwei Phasen. In der ersten Phase werden die Stützpunkte (X-, Y- und Z-Werte) der Bahn und in der zweiten Phase wird die Orientierung (Roll-, Pitch- und Yaw-Werte) für jeden einzelnen Stützpunkt berechnet.

Berechnung der Stützpunkte

Zuerst wird eine Folge $E^{Bahn} = \{E_i^{Bahn}\}_{i=1}^{N^{Bahn}}$ von äquidistanten Schnittebenen E_i^{Bahn} im Abstand der Zustellung d_z mit $E_1^{Bahn} = E_{Anfang}$ und $E_{N_B}^{Bahn} = E_{Ende}$ berechnet. Es gilt: $N^{Bahn} = \lceil \frac{d_{Anfang,Ende}}{d_z} \rceil$. Diese Schnittebenen E_i^{Bahn} schneiden den Zementmantel P^{Mantel} (Systemroutine `$CUT_PLANE_POLYGONS`, Anhang B, S. 171). Beim Schnitt einer Ebene E_i^{Bahn} ergibt sich ein Polygon $Poly_i^{Mantel\,\mathrm{rand}}$, das sich als zyklische Folge von Punkten $\{p_{ik}^{Mantel\,\mathrm{rand}}\}_{k=1}^{N_i^{Rand}}$ darstellen lässt. Insgesamt ergibt sich eine Polygonfolge $\{Poly_i^{Mantel\,\mathrm{rand}}\}_{i=1}^{N_E}$. Der Vorteil dieser Vorgehensweise besteht darin, dass für die Bahnberechnung entweder der berechnete Zementmantel P^{Mantel} aus dem vorigen Abschnitt verwendbar ist oder der Mediziner einen selbst modellierten Mantel benutzen kann.

Um bei der Berechnung der Stützpunkte den Fräskopf als punktförmig annehmen zu können (Bild 9.6), werden die Polygone $Poly_i^{Mantel\,\mathrm{rand}}$ jeweils um den Radius des Fräsers $r_{Fräse}$ geschrumpft (virtueller Zementmantelrand, Anwendung der Systemroutine `$SHRINK_POLYGON`). Das Ergebnis ist eine Polygonfolge $P_1^{Bahn} = \{Poly_{i1}^{Bahn}\}_{i=1}^{N^{Bahn}}$. Die zugehörigen Punkte p_{i1k}^{Bahn} der zyklischen Punktfolge $Poly_{i1}^{Bahn} = \{p_{i1k}^{Bahn}\}_{k=1}^{N_{i1}^{Bahn}}$ bilden ein Teil der zu berechnenden Bahnstützpunkte.

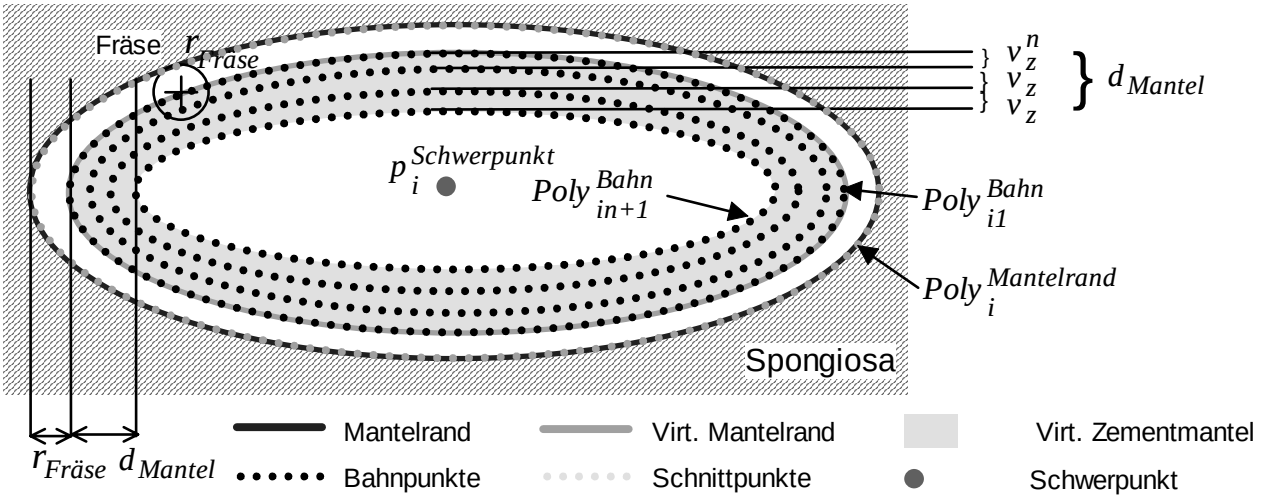


Bild 9.6: Schnitt durch das Geometriemodell des Zementmantels zur Berechnung der Bahnpunkte für die Roboterfräsbewegung

Aus jedem Polygon $Poly_{i1}^{Bahn}$ werden n weitere Bahnen (Umrundungen) durch Schrumpfen von $Poly_{i1}^{Bahn}$ um v_z bzw. v_z^n gebildet (Systemroutine `$SHRINK_POLYGON`), sodass gilt:

$$(n-1)v_z + v_z^n = d_f \text{ mit } v_z^n \leq v_z \text{ und } n \in \{1, 2, ...\} \tag{9.2}$$

Die äußerste Umrundung entspricht $Poly_{i1}^{Bahn}$. Bei inneren Umrundungen $\{Poly_{ij}^{Bahn}\}_{j=n+1}^{2}$ trägt der Fräskopf mit dem gesamten Vorschub v_z Material (Schruppen) ab. Bei der letzten, äußersten Umrundung $Poly_{i1}^{Bahn}$ trägt er nur mit dem Vorschub v_z^n ($v_z^n \leq v_z$) ab (Schlichten). Der Fräskopf startet und beendet seine Bewegung im Schwerpunkt $p_i^{Schwerpunkt}$ des Polygons $Poly_{i1}^{Bahn}$ (Systemroutine `$BALANCE_POINT`). Der Umlaufsinn eines Polygons $Poly_{ij}^{Bahn}$ ist entgegengesetzt zum Umlaufsinn des Fräskopfs. Die Bahnstützpunkte P^{Bahn} lassen sich insgesamt wie folgt beschreiben:

$$P^{Bahn} = \{\{Poly_{ij}^{Bahn}\}_{j=n+1}^{1}\}_{i=1}^{N_B} \cup \{p_i^{Schwerpunkt}\}_{i=1}^{N_B} \tag{9.3}$$

Berechnung der Fräsrichtung für jeden Stützpunkt

Die Orientierung für einen Stützpunkt p_{ij}^{Bahn} wird durch eine Gerade von einem Punkt auf der Ebene des Sicherheitspolygons $Poly^{Sicherheit}$ zu dem Stützpunkt festgelegt. Diese Geradenrichtung wird im Folgenden als Fräsrichtung für den Stützpunkt bezeichnet.

An das Verfahren zur Berechnung der Fräsrichtungen werden folgende Anforderungen gestellt:

1. Die Fräse darf nicht mit der Spongiosa kollidieren.
2. Die Fräse darf das Sicherheitspolygon $Poly^{Sicherheit}$ nicht verlassen (Bild 9.3).
3. Die Fräsachse soll, falls möglich, parallel zur Z-Achse des Zementmantels verlaufen. Die sich dabei ergebene Richtung wird als Vorzugsrichtung bezeichnet.
4. Falls die Fräse die Vorzugsrichtung nicht einnehmen kann, weil dies eine Verletzung der Anforderung 1 oder 2 bedeuten würde, soll die Fräse möglichst wenig geneigt werden, damit der Roboter nur geringe Orientierungsänderungen durchführen muss.

Das eingesetzte Verfahren durchläuft zwei Phasen. In der ersten Phase werden alle möglichen Richtungen zu einem Stützpunkt berechnet, die die ersten beiden Anforderungen erfüllen. In der zweiten Phase wird aus den Fräsrichtungen der ersten Phase diejenige bestimmt, die mit der Z-Achse des Zementmantels den kleinsten Winkel bildet, um die dritte und vierte Anforderung zu erfüllen.

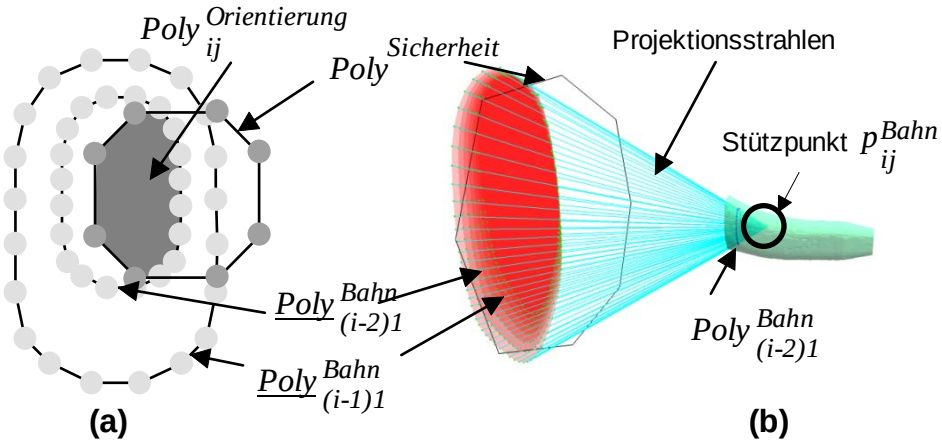


Bild 9.7: Berechnung der möglichen Fräsrichtungen

Für die Berechnung aller möglichen Fräsrichtungen (Bild 9.7a und b) in der ersten Phase werden für jeden Stützpunkt p_{ij}^{Bahn} einer Schicht i alle Polygone P_{kj}^{Bahn} auf dem virtuellen Zementmantelrand mit $1 \leq k < i$ auf das Sicherheitspolygon $P^{Sicherheit}$ mit dem Stützpunkt p_{ij}^{Bahn} als Projektionszentrum projiziert (Systemroutine `$CENTRAL_PROJECTION`). Das Ergebnis eines projizierten Polygons

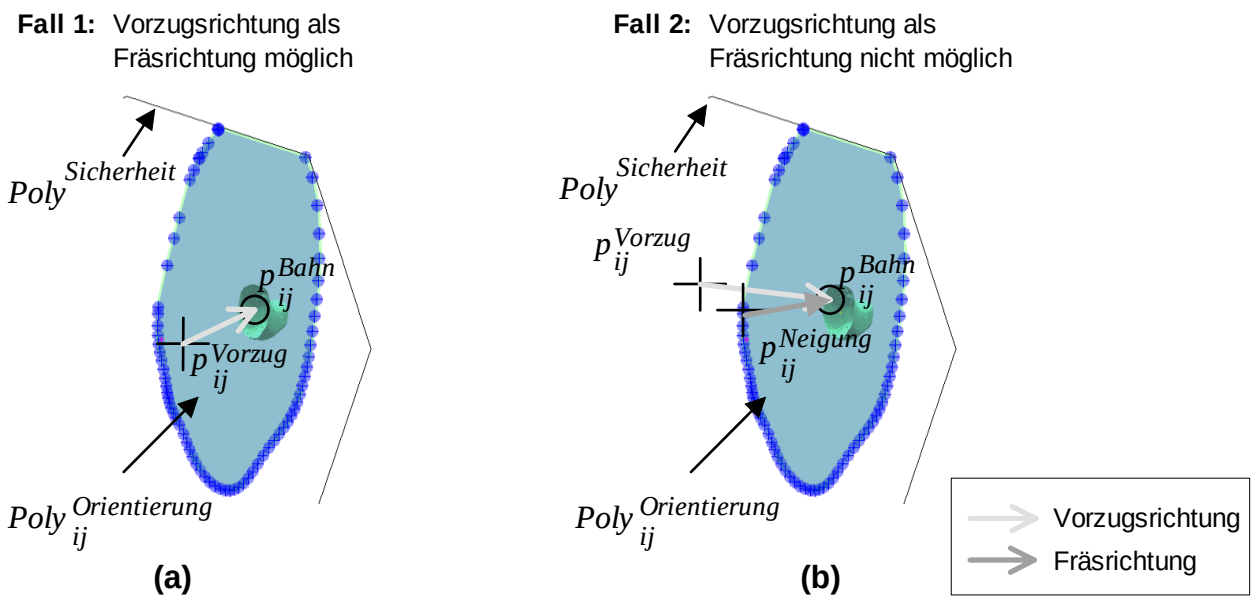


Bild 9.8: Bestimmung der einzunehmenden Fräsrichtung

$Poly_{ij}^{Bahn}$ ist wiederum ein Polygon $\underline{Poly_{ij}^{Bahn}}$. Anschließend wird das Polygon $Poly_{ij}^{(1)Orientierung}$ berechnet, das vollständig in allen projizierten Schnittpolygonen $\underline{Poly_{ij}^{Bahn}}$ und dem Sicherheitspolygon $Poly^{\mathrm{Si}cherheit}$ enthalten ist (Systemroutine `$CUT_POLYGONS`). Falls Polygon $Poly_{ij}^{(1)Orientierung}$ existiert, wird es um den Sicherheitsabstand $d_{\mathrm{Si}cherheit}$ geschrumpft (Systemroutine `$SHRINK_POLYGON`). Das resultierende Polygon $Poly_{ij}^{Orientierung}$ beschreibt alle Fräsrichtungen, die die ersten beiden Anforderungen erfüllen. Durch das Schrumpfen um $d_{\mathrm{Si}cherheit}$ wird sichergestellt, dass die Fräse den Abstand $d_{\mathrm{Si}cherheit}$ zum Zementmantel einhält. Wenn $Poly_{ij}^{Orientierung}$ nicht existiert, dann ist der zugehörige Stützpunkt p_{ij}^{Bahn} für die Fräse unter keinen Umständen erreichbar und damit kann der Roboter den Zementmantel mit dem vorhandenen Fräskopf nicht in die Spongiosa fräsen.

Um die dritte und vierte Anforderung in der zweiten Phase des Verfahrens zu erfüllen, müssen zwei Fälle in Abhängigkeit davon unterschieden werden, ob die Fräse die Vorzugsrichtung einnehmen kann, ohne dabei die ersten beiden Anforderungen zu verletzen (Bild 9.8).

Um festzustellen, um welchen Fall es sich handelt, wird eine Gerade durch den Stützpunkt p_{ij}^{Bahn} und die Vorzugsrichtung mit der Ebene des Sicherheitspolygon $Poly^{\mathrm{Si}cherheit}$ geschnitten (Systemroutine `$CUTS_LINE_PLANE`). Im ersten Fall liegt der resultierende Schnittpunkt (Vorzugspunkt p_{ij}^{Vorzug}) im Polygon $Poly_{ij}^{Orientierung}$ (Systemroutine `$POINT_INSIDE_POLYGON`). Tritt der erste Fall ein, so kann die Fräse als Fräsrichtung die Vorzugsrichtung einnehmen (Bild 9.8a). Im zweiten Fall liegt p_{ij}^{Vorzug} außerhalb von $Poly_{ij}^{Orientierung}$, dann muss eine andere Fräsrichtung als die Vorzugsrichtung bestimmt werden (Bild 9.8b). Dafür wird derjenige Punkt $p_{ij}^{Neigung}$ auf $Poly_{ij}^{Orientierung}$ bestimmt, der den geringsten Abstand zu p_{ij}^{Vorzug} besitzt (Systemroutine `$NEAREST_POINT_ON_POLYGON`), um die vierte Anforderung zu erfüllen. Es ergibt sich als Fräsrichtung für den zugehörigen Stützpunkt p_{ij}^{Bahn} die Richtung von $p_{ij}^{Neigung}$ zum Stützpunkt p_{ij}^{Bahn}.

Ergebnis

Der Nutzen, der sich für die Mediziner durch die Ausführung des für diese Applikation erstellten Generierungsplans ergibt, ist die Tatsache, dass sie unterschiedliche Fräsparameterwerte und Prothesenschäfte ausprobieren können, ohne tiefgreifende Kenntnisse über die Roboterprogrammierung besitzen zu müssen. Die Mediziner kommen durch die Programmgenerierung nicht in den Kontakt mit irgendeiner Programmiersprache.

Für das Fräsen des Zementmantels sind aufgrund der komplexen Geometrie der Prothesenschäfte und in Abhängigkeit der Wahl der Fräsparameterwerte zahlreiche Bahnpositionen (Tabelle 9.1) notwendig. Folglich kann selbst ein erfahrener Roboterprogrammierer nicht darauf verzichten, die Programmerstellung zu automatisieren.

Tabelle 9.1: Zahl der Bahnstützpunkte in Abhängigkeit von Zustellung d_z und Vorschub d_f bei Manteldicke $d_{Mantel} = 3mm$ und Abständen $d_{Anfang,Mitte} = 66mm$, $d_{Mitte,Ende} = 15mm$

	$d_z = \mathbf{1\,mm}$	$d_z = \mathbf{0.5\,mm}$	$d_z = \mathbf{0.25\,mm}$
$d_f = \mathbf{1\,mm}$	16048	32001	63895
$d_f = \mathbf{0.5\,mm}$	31148	62109	124031
$d_f = \mathbf{0.25\,mm}$	61268	122222	244082

9.2 Programmgenerierung zur Porzellankonturbearbeitung

Bei dieser Applikation wird eine Programmgenerierung zum Schleifen und Putzen von variantenreichen Porzellanprodukten vorgestellt.

Motivation

Die Programmierung von Putz- und Schleifvorgängen für Porzellanprodukte (Bild 9.9) wird bisher mittels Teach-in durchgeführt. Die Programmerstellung für eine Produktvariante dauert damit mehrere Stunden. Besonderes problematisch ist die hohe Zahl von Produkt- und Werkzeugvarianten (Bild 9.10). Das Ziel einer Programmgenerierung ist die Aufwandsreduktion für die Programmerstellung zur Bearbeitung einer Produktvariante mit verschiedenen Werkzeugen. Es werden folgende Anforderungen an die Programmgenerierung zur Porzellankonturbearbeitung gestellt:

1. Das System soll aus CAD-Daten der Produkte im Format STL Roboterprogramme erstellen.
2. Das System soll die Programme in der Zielsprache MELFA BASIC IV ausgeben.
3. Die generierten Programme sollen mit einem vorhandenen MELFA BASIC IV-Programm, das u. a. Anweisungen zur Kommunikation mit einer SPS enthält, Daten (Anzahl der Drehungen des Porzellanprodukts am Werkzeug, Geschwindigkeit für das Schleifen bzw. Putzen) austauschen.

Bild 9.9: Auswahl an Porzellanproduktvarianten

Bild 9.10: Auswahl an Werkzeugen für die Konturbearbeitung der Porzellanprodukte

Der Grund für die dritte Anforderung ist, dass ein Maschinenführer die Werte vor der Tellerbearbeitung über eine SPS eingibt. Folglich sind die Werte erst zur Laufzeit des generierten Roboterprogramms bekannt und nicht bereits zum Zeitpunkt der Programmgenerierung.

Realisierung

Unter Verwendung der Schablone zur Programmgenerierung für variantenreiche Produkte (Abschnitt 5.2.2, S. 64ff) wurde ein Generierungsplan erstellt, der die genannten Anforderungen erfüllt.

Eingangsparameter

Der Plan benötigt die CAD-Daten eines Tellers. Die CAD-Daten stellen den Teller als Fläche dar, deren äußerer Rand (Kontur), bearbeitet werden muss. Neben den CAD-Daten kann das Roboterprogramm über folgende einstellbare Parameter beeinflusst werden (Bild 9.11):

- Konturkorrektur
- Startwinkel
- Abstand und Winkel für An- und Abfahrbewegungen
- Winkelabweichung zur Stützpunktreduktion

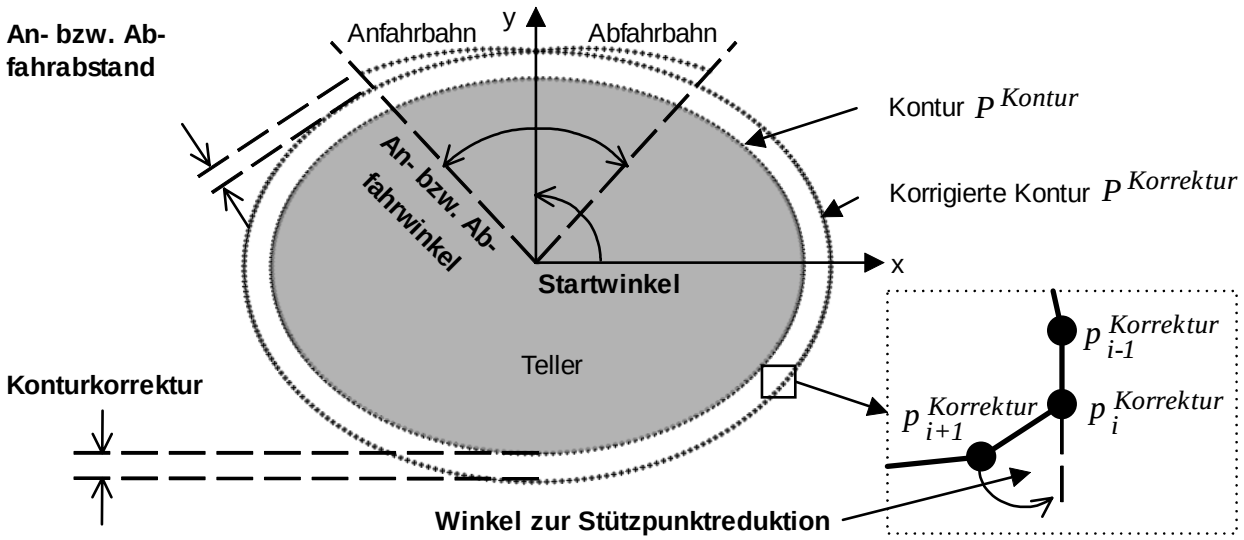


Bild 9.11: Einstellbare Parameter zur Spezifikation der Roboterbewegungen

Da die Produkte vor dem Schleif- und Putzvorgang gebrannt werden, kann über den Parameter ”Konturkorrektur” eine Schrumpfung oder Streckung der Kontur erzielt werden. Die Kontur ist eine Folge $P^{Kontur} = \{p_i^{Kontur}\}_{i=1}^{N^{Kontur}}$ von Konturpunkten p_i^{Kontur}. Die korrigierte Kontur wird analog als $P^{Korrektur} = \{p_i^{Korrektur}\}_{i=1}^{N^{Korrektur}}$ bezeichnet. Der Startwinkel gibt an, an welcher Stelle auf der Kontur die Bearbeitung beginnen soll. An dieser Stelle muss der Roboter An- bzw. Abfahrbewegungen durchführen, die über einen Abstand und einen Winkel zu der korrigierten Kontur $P^{Korrektur}$ spezifiziert werden. Um die Anzahl der Konturpunkte $N^{Korrektur}$, die aufgrund der Genauigkeit der CAD-Daten groß sein kann, einzuschränken, wird der Winkel zur Stützpunktreduktion verwendet. Ist der Winkel, den die Gerade, definiert durch die Punkte $p_{i-1}^{Korrektur}$ und $p_i^{Korrektur}$, und die Gerade, definiert durch die Punkte $p_i^{Korrektur}$ und $p_{i+1}^{Korrektur}$, bildet kleiner als die angegebene Winkelabweichung, so wird der Punkt $p_{i+1}^{Korrektur}$ nicht berücksichtigt und das Verfahren mit dem Punkt $p_{i+2}^{Korrektur}$ fortgesetzt. Auf diese Weise wird von Punkt zu Punkt vorgegangen und sukzessive Punkte aussortiert.

Generierungsplan

Vor der Planausführung müssen die CAD-Daten des Tellers in das COSIMIR®-Simulationsmodell importiert werden. Die CAD-Daten enthalten eine Polygonmenge (Bild 4.8, S. 48), die den Teller als Fläche beschreiben, deren äußere Kontur zu bearbeiten ist (Bild 9.12a). Sind mehrere Produkte im Simulationsmodell, so wird ein Programm für den selektierten Teller erzeugt.

Damit der Roboter die Position kennt, an der er den Teller greifen soll, kann man diesem manuell vor der Planausführung einen Greifpunkt angeben. Erfolgt diese Angabe nicht, so fügt der Plan dem Teller einen auf die Tellerfläche projizierten Schwerpunkt als Greifpunkt hinzu.

Im Folgenden werden die Schritte der Plananpassung dieser Applikation für die Schablone zur Programmgenerierung von variantenreichen Produkten (Bild 5.10, S. 67) erläutert. Die Planausführung wird in COSIMIR® dargestellt, um dem Anwender den Ablauf zu veranschaulichen (Bild 9.12).

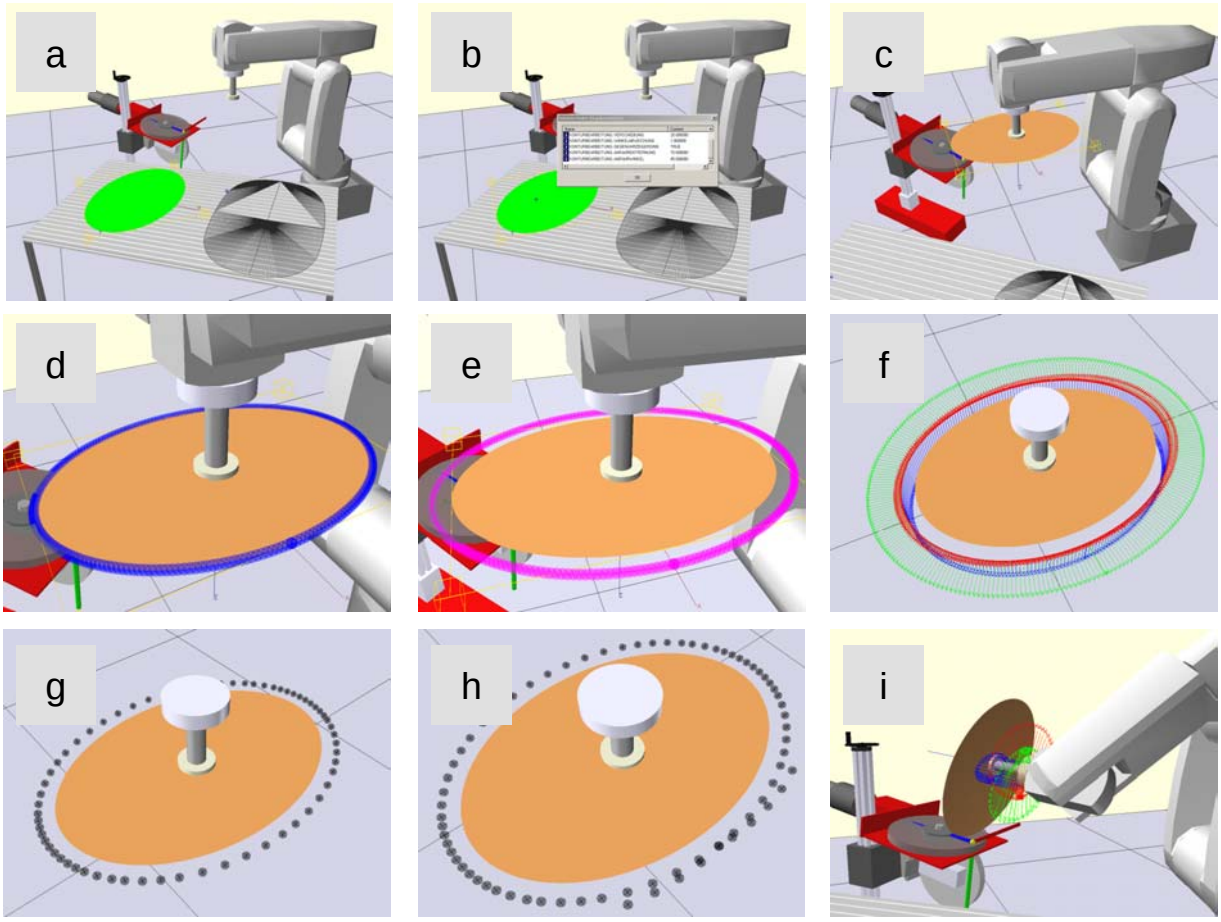


Bild 9.12: Schritte zur Programmgenerierung für die Bearbeitung der Porzellanprodukte

Im Schritt `INITIALISIERUNG` wird die Syntax der zu generierenden Programme über die Systemvariable `$LANGUAGE` festgelegt (Mitsubishi MELFA BASIC IV). Die `PARAMETERANFORDERUNG` enthält die Anforderung der im vorangegangenen Abschnitt erläuterten Parameter beim Eingabeassistenten mittels der Systemroutine `$PARAMETER` (Bild 9.12b).

In `ZUGRIFF_SIMULATIONSMODELL_LESEND` liest `LESEN_WERKSTÜCK_GEOMETRIE` die Tellerfläche, d.h., die Menge der Polygone, in die Planvariable `GEOMETRIE[][]` (Systemroutine `$GET_GEOMETRY`). Im Schritt `LESEN_WEITERE_INFORMATIONEN` wird ein eventuell vorhandener Greifpunkt (Systemroutine `$GET_GRIPPOINT`), an dem der Roboter den Teller greifen soll, aus dem Simulationsmodell gelesen. Der Schritt `ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND` weist dem Teller einen Greifpunkt im Schwerpunkt zu (Systemroutine `$SET_GRIPPOINT`), falls dieser nicht vorhanden ist, und verbindet den Teller (Systemroutine `$GRIP`) mit dem Greifer (Bild 9.12c).

Der Schritt `BERECHNUNG_STÜTZPUNKTE` trianguliert die Tellerfläche(`$TRIANGULIZE`), d.h., jedes einzelne Polygon wird in Dreiecke zerlegt. Aus dieser Dreieckszerlegung werden die Punkte p_i^{Kontur} für die äußere Kontur P^{Kontur} ermittelt (Systemroutine `$BOUNDARY`). Eine Kante eines Polygons gehört genau dann zur äußeren Kontur, wenn diese Kante einmal in allen Dreiecken vorkommt (Bild 9.12d). Der Schritt `SORTIERUNG_STÜTZPUNKTE` sortiert die Punkte p_i^{Kontur} in der Folge P^{Kontur} derart, dass zu jedem Punkt p_i^{Kontur} der Nachfolger p_{i+1}^{Kontur} ermittelt wird, der den geringsten Abstand

zu p_i^{Kontur} besitzt. Die Kontur P^{Kontur} wird dann durch Anwendung der Systemroutine `$SHRINK_POLYGONS` oder `$STRETCH_POLYGONS` (Bild 9.12e) um den Parameter "Konturkorrektur" in Abhängigkeit davon geschrumpft oder gestreckt, ob der Parameter "Konturkorrektur" positiv oder negativ ist. Die Punkte $p_i^{Korrektur}$ der sich ergebenden Folge $P^{Korrektur}$ bilden die initialen Bahnstützpunkte.

Der Schritt `ZUORDNUNG_ORIENTIERUNG` legt die Orientierung eines Punktes $p_i^{Korrektur}$ durch die Berechnung der X-, Y- und Z-Koordinatenachsen der Orientierung fest. Die X-Achse der Orientierung eines Punkts $p_i^{Korrektur}$ wird in Richtung dessen Nachfolgers $p_{i+1}^{Korrektur}$ gelegt. Die Z-Achse berechnet sich durch Schnitt von zwei Ebenen. Die erste Ebene wird durch die X-Achse der Orientierung als deren Normalenvektor festgelegt. Die zweite Ebene wird durch drei Punkte definiert. Dies sind der Punkt $p_i^{Korrektur}$, der Punkt $p_{i+1}^{Korrektur}$ und ein dritter Punkt, der oberhalb von $p_i^{Korrektur}$ liegt, d.h., in Richtung der Z-Achse des Tellerkoordinatensystems (Bild 9.12f). Die Y-Achse ergibt sich aus der X- und der Z-Achse (Rechtssystem). Der Schritt `REDUKTION_BAHNPOSITION` löscht Bahnpositionen gemäß dem Parameter "Winkel zur Stützpunktreduktion" (Bild 9.12g). Der Schritt `ZUORDNUNG_BEZUG` weist jedem Bahnstützpunkt als Bezugskoordinatensystem das Koordinatensystem des Produkts zu.

Der Schritt `BERECHNUNG_AN_UND_ABFAHRWEGE` berechnet die Anfahr- und Abfahrbewegungen gemäß den Parametern "An-/Abfahrabstand" und "An-/Abfahrwinkel" (Bild 9.12h). Der darauf folgende Schritt `UMRECHNUNG_EXTERNES_WERKZEUG` rechnet alle Bahnstützpunkte so um, dass der Roboter das Produkt entlang des Bearbeitungswerkzeugs (z. B. Schleifmaschine) führt (Bild 9.12i). Im Schritt Zuordnung `ZUORDNUNG_KONFIGURATION_UND_TURN` erfolgt die Zuordnung derselben Konfiguration für alle Stützpunkte (Right, Above, Flip). Die Turnangabe spielt in dieser Applikation keine Rolle.

Die Bahnstützpunkte für die Anfahrbewegung sind im Feld `BAHN[1][]` enthalten. Entsprechend enthält das Feld `BAHN[2][]` die Bahnstützpunkte für die Bewegung entlang der Kontur und `BAHN[3][]` die Bahnstützpunkte für die Abfahrbewegung.

Programmstrukturierung

Im Schritt `PROGRAMMSPEZIFIKATION` wird der Inhalt des zu generierenden Roboterprogramms festgelegt (Bild 9.13). Die Bedeutung der Plananweisungen der Programmspezifikation kann man aus Tabelle 7.2 (S. 104) entnehmen. Korrespondierende Plananweisungen und MELFA BASIC IV-Befehle, die aus der zugehörigen Syntaxdefinition stammen (S. 107ff), sind hervorgehoben. Aus Platzgründen fehlen die Implementierungen der Subroutinen `ANFAHREN` und `ABFAHREN`. Sie sind entsprechen der Implementierung der Subroutine `UMLAUF`. Die MELFA BASIC IV-Variable `POS1` (Zeile 2) enthält die Bahnpositionen der Anfahrbewegung aus der Planvariablen `BAHN[1][]`. Entsprechend enthält `POS2` (Zeile 3) die Bahnpositionen der Bewegung entlang der Kontur und `POS3` (Zeile 4) die Bahnpositionen der Abfahrbewegung. Die Variable `PRODUKT` ersetzt das in MELFA BASIC IV fehlende Konzept des Werkstückkoordinatensystems und enthält die Position des Porzellanprodukts, auf die sich die Werte der Variablen `POS1` bis `POS3` beziehen.

Programmspezifikation

```
1  $PROJECT("PORZELLAN");
2
3  $DECL_EXTERN_VAR(UMLAEUFE);
4  $DECL_EXTERN_VAR(VELOCITY);
5
6  $MAINPROGRAM();
7    $MAINROUTINE();
8      $CALL_PROGRAM("10070");
9      $PATH_SPEED(VELOCITY);
10     $PTP(BAHN[1][1]);
11     $CALL_SUBROUTINE("ANFAHREN");
12     $TIMES(UMLAEUFE)
13     $CALL_SUBROUTINE("UMLAUF");
14     $CALL_PROGRAM("PROG_JRC");
15     $ENDTIMES();
16     $CALL_SUBROUTINE("ABFAHREN");
17   $ENDMAINROUTINE();
18
19 ...
20   $SUBROUTINE("UMLAUF");
21     $PATH(BAHN[1][]);
22   $ENDSUBROUTINE();
23 ...
24
25 $ENDMAINPROGRAM();
26 $ENDPROJECT
```

Mitsubishi MELFA BASIC IV

```
  10 DEF INTE I,K
  20 DIM POS1(5)  ' BAHN[1][]
  30 DIM POS2(64) ' BAHN[2][]
  40 DIM POS3(5)  ' BAHN[3][]
  50 CALLP "10070"
  60 SPD M_VELOCITY
  70 MOV PRODUKT*POS1(1)
  80 GOSUB *ANFAHREN
  90 FOR K = 1 TO M_UMLAEUFE
 100 GOSUB *UMLAUF
 110 CALLP "PROG_JRC"
 120 NEXT K
 130 GOSUB *ABFAHREN
 140 END

 ...
 220 *UMLAUF
 230 CNT 1
 240 FOR I = 1 TO 64
 250  MVS PRODUKT*POS2(I)
 260 NEXT K
 270 CNT 0
 280 RETURN
 ...
```

Bild 9.13: Programmspezifikation und zugehöriges Programm zur Bearbeitung der Teller

Da das zu generierende Roboterprogramm Werte mit einem vorhandenen, manuell erstellten Roboterprogramm mit dem Namen `10070` austauschen muss, sind in der Programmspezifikation zwei externe Variablen (`UMLAEUFE`, `VELOCITY`) angeben (Zeile 3, 4). Das vorhandene Programm deklariert diese Variablen global und schreibt sie nach seinem Aufruf (Zeile 8) mit den Werten, die der Maschinenführer über die SPS vorgibt. In den Programmen tauchen diese Variablen als `M_UMLAEUFE` (Zeile 12) und `M_VELOCITY` (Zeile 9) auf. Dabei kennzeichnet das Präfix ”`M_`” eine globale, reellwertige Variable in MELFA BASIC IV. Die Variable `UMLAEUFE` enthält die Anzahl der Umdrehungen des Produkts an einer Bearbeitungsstation und die Variable `VELOCITY` enthält die dabei notwendige Geschwindigkeit.

Da der MELFA BASIC IV-Befehl `JRC` [107] verwendet werden muss, der nicht als Plananweisung in der Programmspezifikation zur Verfügung steht, wird dieser in einem eigenen MELFA BASIC IV-Programm mit dem Namen `PROG_JRC` gekapselt und dieses aufgerufen (Zeile 14).

Der Befehl `JRC` ist nötig, weil bei mehreren Drehungen des Tellers am Werkzeug die sechste Achse ihren Grenzwert erreichen kann. Der Befehl `JRC` setzt deshalb den Wert der sechsten Achse in der Robotersteuerung um 360 Grad zurück, ohne dabei die sechste Achse des Roboters zu drehen.

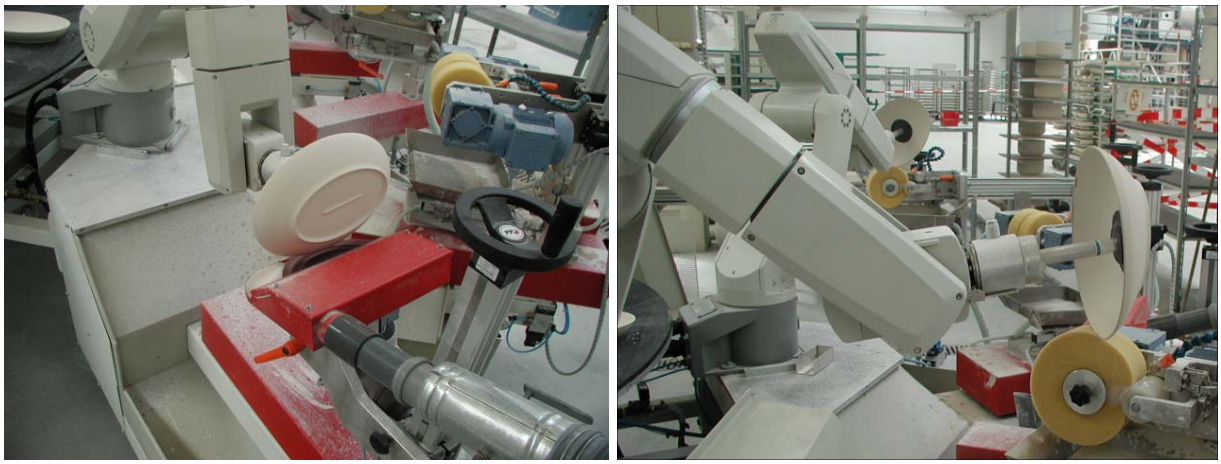


Bild 9.14: Schleifen und Putzen von Tellern durch ein generiertes Roboterprogramm

Tabelle 9.2: Programmieraufwands bei der Teach-In-Programmierung und der automatisierten Programmgenerierung (in % der Teach-In-Programmierzeit eines Produkts)

Anzahl Varianten	Teach-in [%]	Programm-generierung [%]	Reduktion [%]
0	0.0	70.0	-
1	100.0	71.4	28.6
2	200.0	72.8	63.6
10	1000.0	84.0	91.6

Ergebnis

Um den Programmieraufwand zwischen der bisher eingesetzten Teach-in-Programmierung und dem in dieser Arbeit entwickelten System zu vergleichen, wird der Aufwand zur Programmierung eines Tellers mit dem Teach-in-Verfahren zu 100 % gesetzt (Tabelle 9.2). Demzufolge beträgt der Aufwand zur Implementierung des Generierungsplans 70 % und der zusätzliche Aufwand für eine Variante (Import der CAD-Daten, Laden der Programme in die Robotersteuerung, Programmerzeugung für verschiedene Parameterwerte) zusammen etwa 1,4 %. Schon für eine Produktvariante ergibt sich eine Aufwandsreduktion beim Einsatz des in dieser Arbeit entwickelten Systems von etwa 28,6 %, für zehn Produktvarianten wird der Aufwand sogar um 91.6 % gesenkt.

Bei der Inbetriebnahmephase (Bild 9.14) erwies sich das Konzept des Generierungsplans als praktisch und flexibel, weil zusätzliche Kundenanforderungen, die dieser erst zu diesem späten Zeitpunkt formulierte, unmittelbar umsetzbar waren. Dazu zählt z. B. der Einsatz des MELFA BASIC IV-Befehls `JRC` und die Vermeidung einer Änderung des Tool Center Points im MELFA BASIC IV-Programm.

9.3 Planung einer exemplarischen Fertigungsstraße

Die bisherigen Applikationen haben gezeigt, dass durch das in dieser Arbeit entwickelte System der Programmieraufwand für Produktvarianten reduzierbar ist. In der nächsten Applikation wird die Generierung von Roboterprogrammen dazu eingesetzt, Fragen, die bei der Planung von Roboter-Fertigungszellen auftreten, zu beantworten.

Im Folgenden werden typische Probleme, die bei der Planung von Roboter-Fertigungszellen auftauchen betrachtet. Dazu dient eine exemplarische Fertigungsstraße zur Montage von Hydraulikventilen. Diese Fertigungsstraße besteht aus den folgenden verketteten Roboter-Fertigungszellen (Bild 9.15):

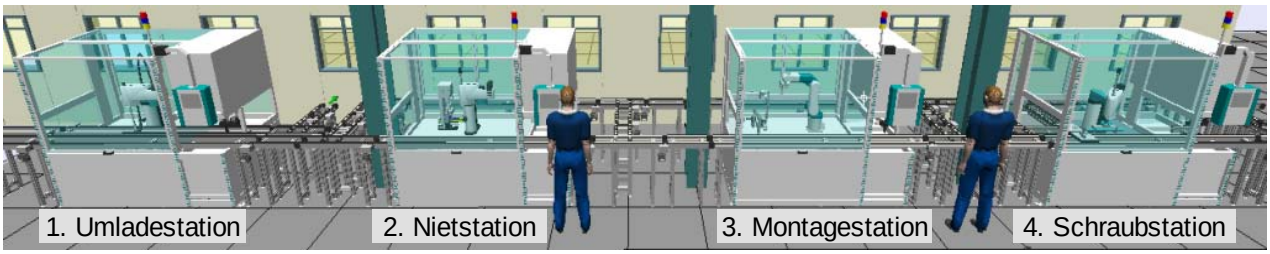


Bild 9.15: Exemplarische Fertigungstrasse in der Planung

1. Umladestation: In dieser Station muss ein Roboter die auf einem Werkstückträger befindlichen Teile Spule, Kolben und Polrohr auf einen anderen Werkstückträger umladen.

2. Nietstation: In dieser Station wird ein Typenschild auf das Gehäuse des Hydraulikventils genietet. Der Roboter muss dazu das Gehäuse und das Typenschild in der Zelle transportieren.

3. Montagestation: In dieser Station fügt ein Montageautomat den Kolben in das Gehäuse des Hydraulikventils. Der Roboter dient in dieser Zelle zum Einlegen des Polrohrs und des Gehäuses in den Montageautomaten und zur Entnahme des Gehäuses aus dem Montageautomaten.

4. Schraubstation: In dieser Station bringt der Roboter vier Schrauben an das Gehäuse an.

In diesen Roboter-Fertigungszellen treten folgende unterschiedliche Layoutprobleme auf, die mithilfe des in dieser Arbeit entwickelten Systems gelöst werden. Es handelt sich dabei um:

1. Umladestation: Bestimmung von Haltepositionen auf Transportbändern.

2. Nietstation: Anordnung der Zellenkomponenten

3. Montagestation: Auswahl und Standortsuche eines Roboters

4. Schraubstation: Roboterstandortsuche mit Reihenfolgeoptimierung des Schraubvorgangs

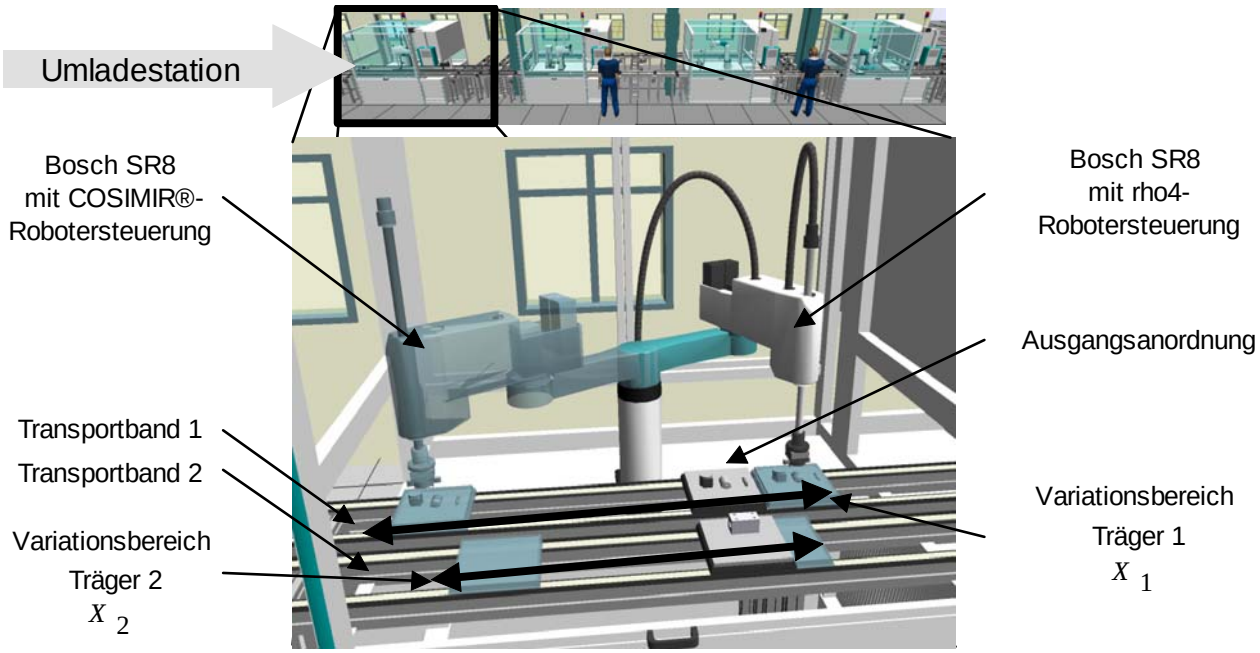


Bild 9.16: Bestimmung von Halteposition auf Transportbändern in der Umladestation

Umladestation

Bei dieser Applikation muss ein Roboter Bosch SR 8 die auf einem ersten Träger befindlichen Teile Spule, Kolben und Polrohr auf einen zweiten Träger legen (Bild 9.16). Die Träger liegen jeweils auf einem Transportband. Es sollen die optimalen Haltepositionen der Träger auf den Bändern ermittelt werden, damit der Roboter die drei Teile in möglichst kurzer Zeit von Träger 1 auf Träger 2 legen kann. Am Roboterflansch befindet sich ein Greifer, der immer nur ein Teil transportieren kann.

Es muss ein Optimierungsproblem gelöst werden, dessen Zielfunktion (Ausführungszeit des Roboterprogramms) die zwei Eingangsvariablen x_1 und x_2 besitzt, die jeweils die Variation eines Trägers auf dem zugehörigen Transportband relativ zu einer Ausgangsanordnung ($x_1 = 0$, $x_2 = 0$) angeben (Bild 9.16). Die Variablenwerte x_1^* und x_2^*, an denen die Zielfunktion ihr globales Minimum besitzt, legen die optimalen Haltepositionen der Träger fest. Die Berechnung der Zielfunktion erfordert die Ausführung von BAPS-Programmen, die die entsprechenden Bewegungen des Roboters und das Öffnen und Schließen des Greifers steuern. Ein unter Verwendung der Schablone zur Layoutoptimierung (Bild 5.14, S. 73) erstellter Generierungsplan erzeugt in Abhängigkeit von der Trägeranordnung die BAPS-Programme. Damit die im Simulationssystem berechneten Ausführungszeiten der BAPS-Programme möglichst mit den späteren Ausführungszeiten in der realen Roboter-Fertigungszelle übereinstimmen, werden die BAPS-Programme auf einem im Simulationsmodell befindlichen Roboter ausgeführt, der mit der realen Robotersteuerung Bosch rho4 gekoppelt ist. Diese Kopplung wurde von BAUER [17] (Möglichkeit 3, Bild 4.9, S. 49) über OPC (Open Process Control) realisiert.

Es besteht der Nachteil, dass bei unerreichbaren Positionen die Steuerung rho4 in einen Fehlerzustand wechselt und die weitere Ausführung von Roboterprogrammen einen manuellen Eingriff auf der rho4 erfordert. Um diesen Nachteil zu beseitigen, wird ein zweiter im Simulationsmodell befindlicher Roboter eingesetzt, der mit der Standardsteuerung von COSIMIR® verbunden ist. Dieser zweite Roboter

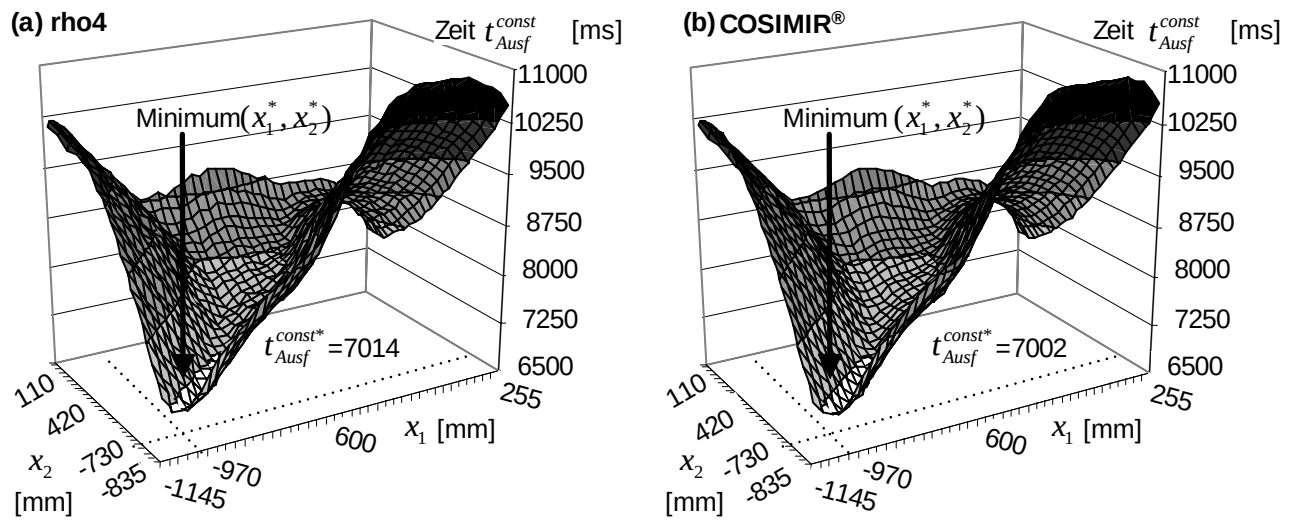


Bild 9.17: Ausführungszeiten für Steuerung rho4 und für Standardsteuerung in COSIMIR®

ist bis auf die gekoppelte Steuerung identisch zum ersten Roboter. Bevor ein BAPS-Programm auf der Steuerung rho4 ausgeführt wird, werden alle darin enthaltenen Positionen mithilfe des zweiten Roboters auf Erreichbarkeit und Kollisionsfreiheit geprüft.

Zur Lösung der Frage nach den optimalen Haltepositionen der beiden Träger werden diese auf den Bändern in äquidistanten Abständen (35 mm) variiert. Es wird für jede Kombination, die beiden Träger anzuordnen, ein BAPS-Programm erzeugt, auf der Steuerung rho4 ausgeführt und die Ausführungszeit ermittelt (Bild 9.17a). Für die Messung wird der Ansatz *Messen der Zeit* T_W eingesetzt (Abschnitt 8.3.1, S. 128), da dieser nicht die genannten Nachteile des Ansatzes *Peripherie ignorieren* besitzt. Aus der berechneten Zielfunktion lässt sich die optimale Halteposition der Träger auslesen.

Des Weiteren wird der gesamte Vorgang wiederholt, wobei statt der Steuerung rho4 die Standardsteuerung von COSIMIR® eingesetzt wird (Bild 9.17b). Da kein Compiler von BAPS nach IRDATA zur Verfügung steht, werden inhaltlich äquivalente Roboterprogramme in IRL generiert (Änderung der Variable `$LANGUAGE`). Zur Kalibrierung der Standardsteuerung von COSIMIR® wird der so genannte Steuerungstakt (vgl. KEIBEL [84]) so eingestellt (18 ms), dass die Ausführungszeit des IRL-Programms für die Ausgangsanordnung der Träger ($x_1 = 0$, $x_2 = 0$) auf der Standardsteuerung von COSIMIR® gleich ist mit der Ausführungszeit des entsprechenden BAPS-Programms auf der Steuerung rho4 (8.5 s). Die maximale Abweichung der Standardsteuerung von COSIMIR® zur Steuerung rho4 für alle 1080 ausgeführten Roboterprogramme betrug 90 ms. Dies entspricht in etwa 1 % bezogen auf die durchschnittliche Dauer (8,7 s) aller Roboterprogrammausführungen. Die zeitminimale Anordnung ist bei beiden Zielfunktionen gleich (Bild 9.18). Die Ausführungszeit der optimalen Anordnung (7,0 s) ist gegenüber der Ausgangsanordnung (8,5) um 17.6 % reduziert.

Die Zielfunktionen in Bild 9.17 wurden auf einem PC mit AMD™ Athlon™ 1200 MHz Prozessor, 256 MB Speicher und dem Betriebssystem Windows™ 2000 Professional berechnet. Die Berechnung dauerte für die Zielfunktion der Steuerung rho4 ca. 8 h 35 min und für die Zielfunktion der Standardsteuerung von COSIMIR® ca. 12 min.

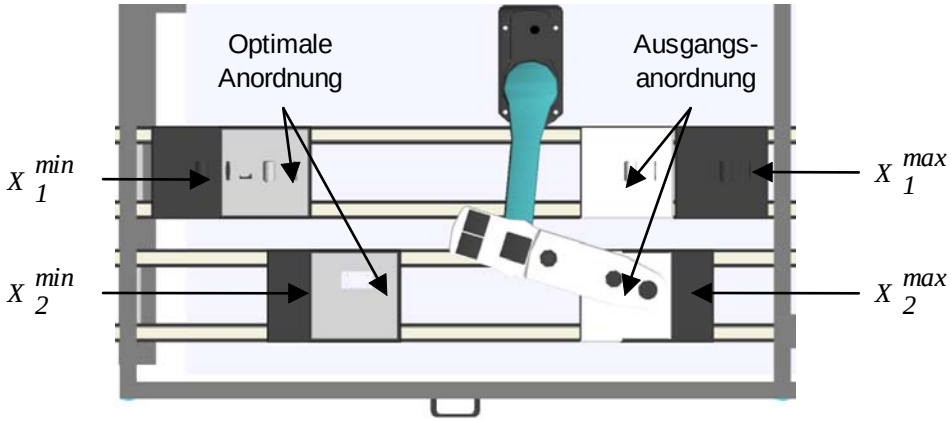


Bild 9.18: Zeitminimale Anordnung der beiden Träger in der Umladestation

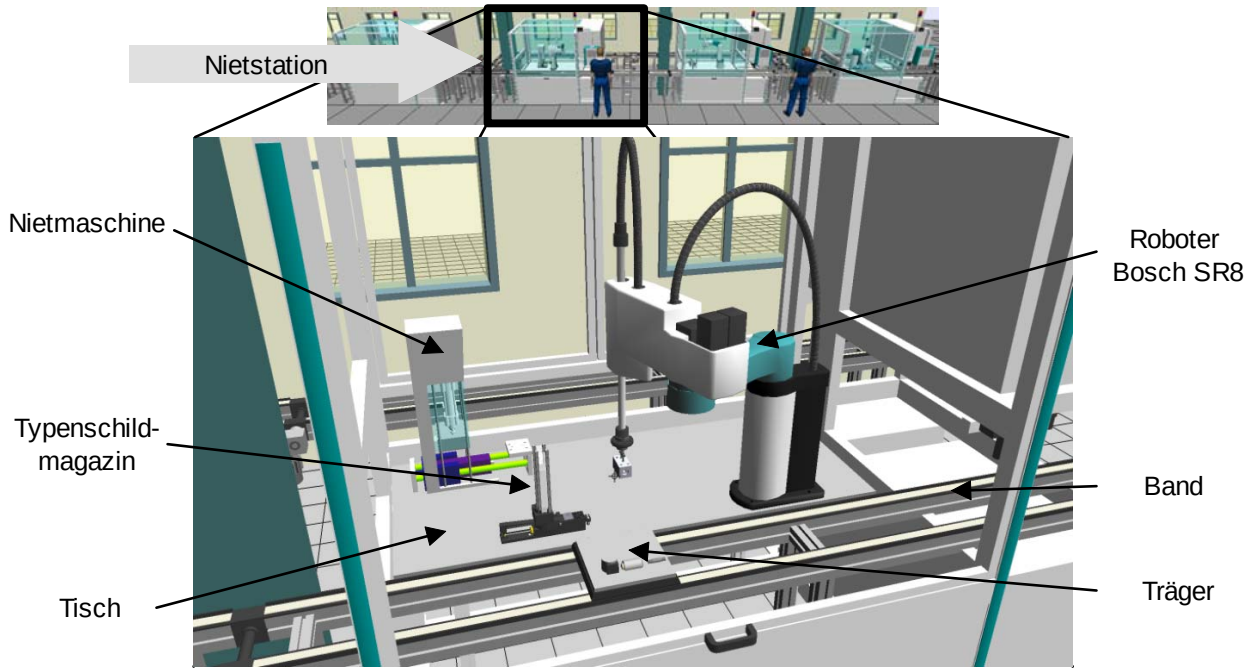


Bild 9.19: Aufbau der Nietstation

Nietstation

In dieser Station soll ein Layout ermittelt werden, sodass die Zeit für den Nietvorgang zur Anbringung des Typenschilds an das Gehäuse des Hydraulikventils minimal ist (Bild 9.19). Der Roboter Bosch SR 8 bringt zunächst das Gehäuse vom Träger in die Nietmaschine. Danach holt er ein Typenschild aus dem Magazin und legt es auf das Gehäuse. Nachdem die Nietmaschine das Typenschild am Gehäuse befestigt hat, legt der Roboter das Gehäuse zurück auf den Träger.

Zur Lösung wurde ein Verfahren aus zwei Phasen mit jeweils einem Generierungsplan realisiert:

1. *Globale Variationsphase*: Das Ziel dieser Phase ist die Suche eines günstigen Anfangslayouts. Dazu variiert der Plan die Standorte der Nietmaschine, des Typenschildmagazins und der Werkstückträger auf einem Raster. Für jedes Layout prüft der Plan, ob alle Roboterpositionen kollisi-

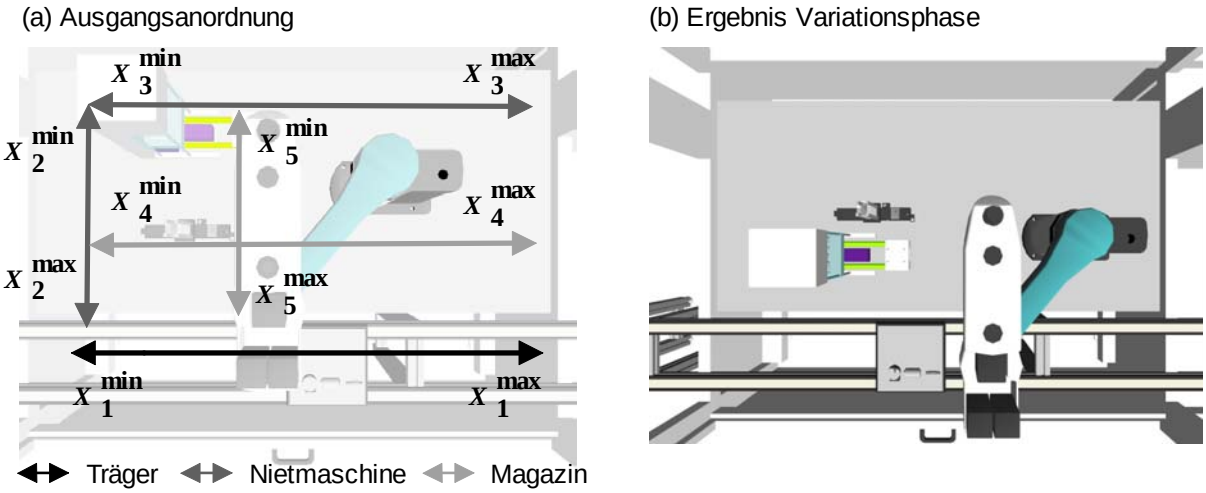


Bild 9.20: Globale Variationsphase

onsfrei erreichbar sind. Ist dies nicht der Fall, verwirft der Plan dieses Layout und untersucht das Nächste. Ansonsten generiert der Plan ein Roboterprogramm, führt es in einem Simulationslauf aus und protokolliert dessen Ausführungszeit. Die Variationsbereiche für die Nietmaschine und das Magazin sind zweidimensional (Verschiebung auf dem Tisch) und der Variationsbereich für den Träger ist eindimensional (Verschiebung entlang des Bandes). Der Plan berechnet somit die auftretende fünfdimensionale Zielfunktion der Ausführungszeit an äquidistanten Stellen.

2. *Lokale Optimierungsphase*: Das Ziel dieser Phase ist die Verbesserung des Anfangslayouts aus der vorangegangenen Phase. Dazu variiert der Plan neben den Verschiebungen der ersten Phase zusätzlich die Rotation der Nietmaschine und des Magazins sowie die Verschiebung und Rotation des Roboters. Damit ist die in dieser Phase betrachtete Zielfunktion zehndimensional. Ausgehend vom Anfangslayout der ersten Phase ändert der Plan jeweils eine Variable der Zielfunktion solange um eine kleine Schrittweite, bis er ein lokales Minimum findet.

Für beide Phasen werden Roboterprogramme auf der Steuerung von COSIMIR® und nicht auf der Steuerung rho4 ausgeführt, weil einerseits die Berechnungsdauer beider Phasen stark verkürzt wird. Andererseits ist der Vergleich der Ausführungszeiten zweier sich minimal unterscheidender Layouts auf der Steuerung von COSIMIR® einfacher, weil auf der Steuerung rho4 Schwankungen in der Ausführungszeit auftreten (Abschnitt 8.3.1, S. 126).

Bild 9.20a zeigt die Ausgangsanordnung und den Variationsbereich zu Beginn der ersten Phase. Insgesamt prüfte der Plan in dieser Phase 1875 Layoutvarianten. Von den 1875 Layoutvarianten wurden für 448 Layoutvarianten Simulationsläufe durchgeführt, von denen 72 Layoutvarianten erfolgreich waren. Bei den anderen Layoutvarianten traten Kollisionen oder nichterreichbare Roboterpositionen vor oder während eines Simulationslaufs auf. Die auftretenden Ausführungszeiten bei den erfolgreichen Simulationsläufen lagen zwischen 7.25 s und 9.4 s. Das Anfangslayout mit der kleinsten Ausführungszeit zeigt Bild 9.20b. Die globale Variationsphase dauerte auf einem PC mit AMD™ Athlon™ 1200 MHz Prozessor, 256 MB Speicher und dem Betriebssystem Windows™ 2000 Professional ca. 27 min.

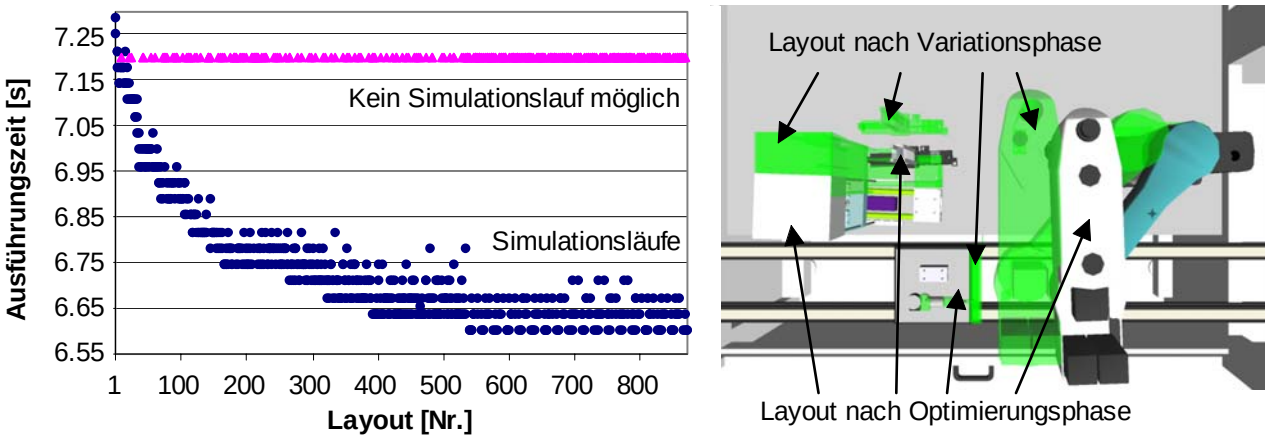


Bild 9.21: Lokale Optimierungsphase

Die lokale Optimierungsphase verbessert das Ergebnis der Variationsphase weiter. Der zugehörige Generierungsplan verschiebt und rotiert die Zellenkomponenten jeweils solange, bis keine weitere Verbesserung der Ausführungszeit auftritt. Die Zellenkomponenten werden dabei um jeweils 10 mm verschoben oder um jeweils zwei Grad rotiert. In dieser Phase prüfte der Plan 871 Layoutvarianten, von denen 533 erfolgreich waren. Diese Phase senkte die Ausführungszeit von 7.25 s auf 6.60 s. Dies entspricht einer Verbesserung von ca. 11 %. Den Verlauf und das Ergebnis der Optimierungsphase zeigt Bild 9.21. Die lokale Optimierungsphase dauerte auf einem PC mit AMD™ Athlon™ 1200 MHz Prozessor, 256 MB Speicher und dem Betriebssystem Windows™ 2000 Professional ca. 39 min.

Für das optimierte Layout wurden inhaltlich äquivalente Programme in BAPS generiert und auf der Steuerung rho4 ausgeführt. Die gemessenen Ausführungszeiten betragen 6.2 s. Dies entspricht einer Abweichung gegenüber der Ausführung auf der Standardsteuerung von COSIMIR® um 6.1 %. Die Ausführungszeit 8.1 s der Ausgangsanordnung (Bild 9.20a) wurde somit um 23.5 % reduziert.

Montagestation

Bei der Planung von Roboter-Fertigungszellen tritt häufig die Frage auf, welcher Roboter geeignet ist, eine Aufgabe durchzuführen. Nach einer Umfrage unter Fachleuten aus der Industrie ist die Beantwortung dieser Frage eine der drei größten Schwierigkeiten bei der Planung einer Roboter-Fertigungszelle [152], denn dazu muss der CAR-Anwender viele manuelle Schritte durchführen und zeitaufwendige Simulationsläufe beobachten. Mithilfe des in dieser Arbeit entwickelten Systems kann der CAR-Anwender Varianten ausprobieren, ohne dass er dabei am Simulationssystem anwesend sein muss.

In der Montagestation muss der Roboter nacheinander das Gehäuse und den Kolben vom Werkstückträger nehmen und in eine Montagevorrichtung legen. Nach Beendigung des Montagevorgangs legt der Roboter das montierte Gehäuse zurück auf den Träger. Für die Montagestation (Bild 9.22a) muss ein geeigneter Roboter aus einer Menge verfügbarer Roboter gefunden werden. Für die Entscheidungsunterstützung wurde ein Generierungsplan implementiert, der Roboter von unterschiedlichen Herstellern prüft. Der Plan tauscht nacheinander den Roboter im Simulationsmodell und verschiebt diesen auf einem zweidimensionalen Raster (Bild 9.22b). Die Höhe des Rasters gibt der

(a) Übersicht der Montagestation (b) Standortvariation

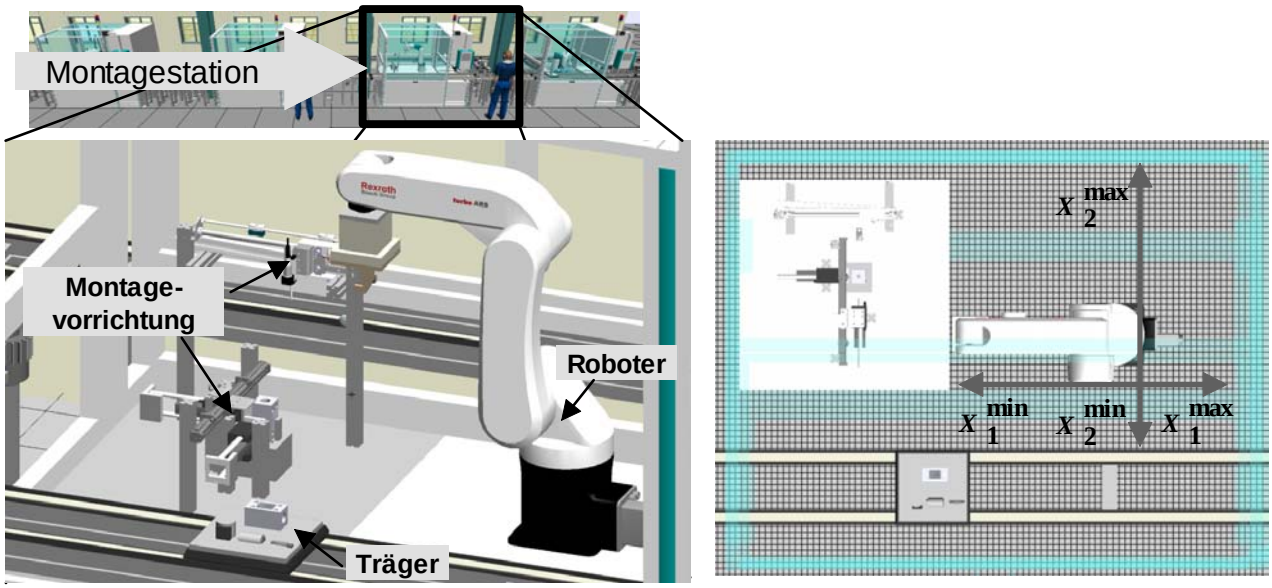


Bild 9.22: Montagestation

CAR-Anwender im Plan vor, sodass die Zahl der untersuchten Varianten und damit der Berechnungsaufwand eingeschränkt wird. Für jeden Roboter und für jeden Standort prüft der Plan, ob alle Roboterpositionen kollisionsfrei erreichbar sind. Ist dies der Fall generiert der Plan ein Roboterprogramm und führt dieses in einen Simulationslauf aus. Ist eine Roboterposition nicht kollisionsfrei erreichbar, probiert der Plan den nächsten Standort oder nächsten Roboter aus. Bei einem erfolgreichen Simulationslauf protokolliert der Plan die zugehörige Ausführungszeit.

Das Raster auf dem die Roboter verschoben werden hat eine Länge von $x_1 = 600$ mm und eine Breite von $x_2 = 400$ mm. Für die Rastereinteilung gilt $\Delta x_1 = \Delta x_2 = 25$ mm. Für jeden Roboter werden somit 425 Standortvarianten ausprobiert. Damit testet der Plan für alle 12 Roboter insgesamt 5100 Varianten. Die generierten Programme wurden alle auf Standardsteuerung von COSIMIR® ausgeführt. Das Ergebnis der Planausführung zeigt Tabelle 9.3.

Tabelle 9.3 stellt die Zahl der durchgeführten und der erfolgreichen Simulationsläufe dar. Bei Ersteren traten vorwiegend Kollisionen auf. Zusätzlich ist die minimale Ausführungszeit für jeden Roboter angegeben. Danach benötigt Roboter 8 die kürzeste Ausführungszeit aller 5100 untersuchten Varianten. Tabelle 9.3 zeigt außerdem die Berechnungsdauer für alle 425 Varianten auf einem PC mit AMD™ Athlon™ 1200 MHz Prozessor, 256 MB Speicher und dem Betriebssystem Windows™ 2000 Professional, angegeben. Auffällig ist die lange Planausführung für Roboter 3. Dies liegt an der großen Zahl der Polygone, mit denen die Roboteroberfläche modelliert wurde, weil dadurch die Kollisionserkennung einen hohen Berechnungsaufwand benötigt.

Verglichen mit der Ausgangsanordnung, bei der Roboter 2 eingesetzt wird, wurde die Ausführungszeit von 10.2 s auf 8.0 s verringert. Dies entspricht einer Verbesserung um 21.5 %.

Tabelle 9.3: Ergebnis der Planausführung für die Montagestation

Roboter	Zahl Simulationsläufe	Zahl erfolgreicher Simulationsläufe	Minimale Ausführungszeit	Dauer
1	33	33	8.5 s	10 min
2	110	110	9.0 s	30 min
3	174	174	9.8 s	9 min
4	25	25	9.6 s	1 h 10 min
5	100	94	8.3 s	1 min
6	71	69	8.5 s	30 min
7	11	11	9.2 s	3 min
8	85	85	8.0 s	1 h 30 min
9	210	207	8.9 s	4 min
10	61	57	8.6 s	1 h 20 min

Schraubstation

In der Schraubstation muss ein Roboter Bosch SR 8 vier Schrauben von einer Palette nehmen und diese am Gehäuse anbringen (Bild 9.23a). Der Roboter besitzt dazu ein Schraubwerkzeug, mit dem er die Schrauben von der Palette nehmen und diese in die Gewinde des Gehäuses schrauben kann. Dieser Vorgang soll in möglichst kurzer Zeit durchgeführt werden. Da sich auf der Palette 24 Schrauben befinden und diese für sechs Gehäuse verwendbar sind, stellen sich folgende Fragen:

1. In welcher Reihenfolge sollen die Schrauben in die Gewinde geschraubt werden?
2. In welches Gewinde muss eine Schraube?
3. Welche vier Schrauben von der Palette soll der Roboter für jedes der sechs Gehäuse verwenden?
3. Wo muss der Roboter auf dem Tisch stehen, sodass der gesamte Schraubvorgang minimal ist?

In industriellen Applikationen findet man häufig Layoutprobleme, die mit Reihenfolgeproblemen kombiniert sind, insbesondere mit palettierten Werkstücken. Um die nachfolgenden Verfahren besser beschreiben zu können, werden zunächst Bezeichner für die einzelnen Schrauben und Gewinde vergeben (Bild 9.23b).Die Schrauben sind in Spalten A bis F und Zeilen 1 bis 4 angeordnet. Die vier Gewinde sind nummeriert von 1 bis 4. Demzufolge ist der Bezeichner S_{F1} derjenige für die Schraube in Spalte F und Zeile 1 und der Bezeichner G_2 derjenige für das zweite Gewinde.

Zur Beantwortung der oben gestellten Fragen wurde ein Generierungsplan implementiert, der folgendes Verfahren realisiert: Der Plan bewegt den Roboter auf einem Raster (Bild 9.24). Für jeden der dadurch entstandenen Roboterstandorte wird die bestmögliche Ausführungszeit berechnet. Diese Berechnung teilt sich in folgende drei Phasen:

1. *Reihenfolgephase*: In dieser Phase wird eine feste Zuordnung zwischen Schrauben und Gewinden vorausgesetzt. Für jeweils vier Schrauben und ein Gehäuse wird ein Rundreiseproblem

(a) Schraubstation (b) Bezeichner der Gewinde und Schrauben

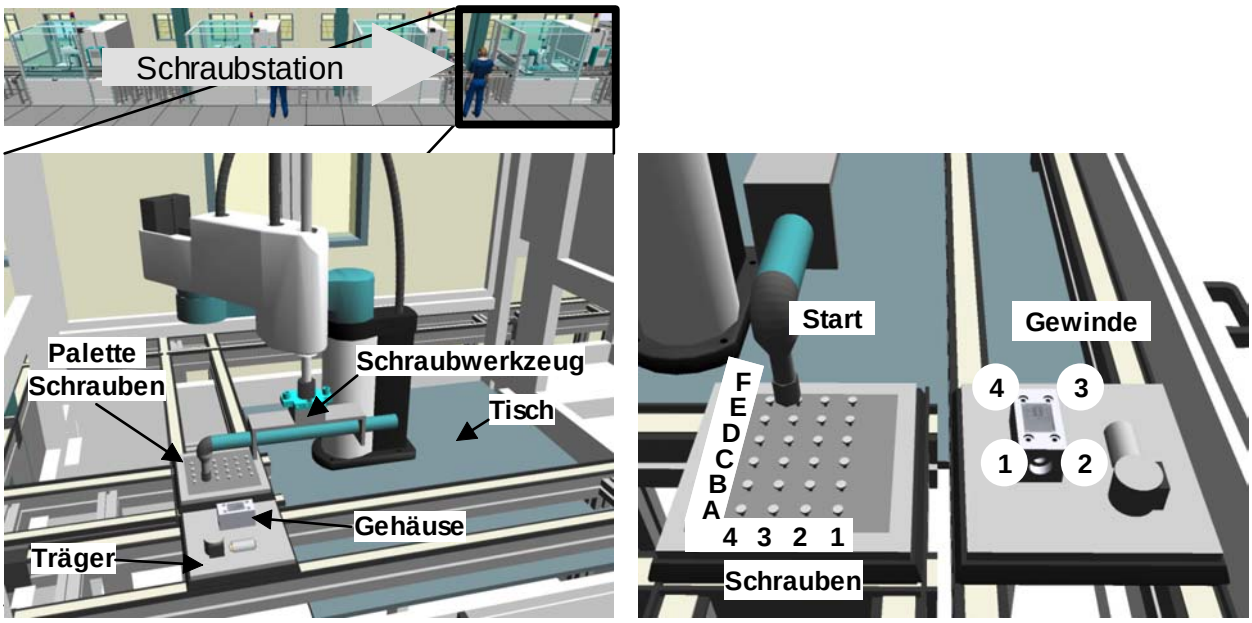


Bild 9.23: Aufbau der Schraubstation und die Bezeichnung der Schrauben und Gewinde

gelöst, das eine optimale Reihenfolge ermittelt. Das Ergebnis dieser Phase sind sechs Rundreisen, sodass der Roboter eine Palette und sechs Gehäuse vollständig bearbeitet hat.

2. *Verbesserungsphase*: In dieser Phase wird versucht, einzelne Schrauben zwischen den Rundreisen zu vertauschen, falls die gesamte Ausführungszeit für alle sechs Rundreisen verkleinert wird (Savingsverfahren [113]). Diese Vertauschung wird solange durchgeführt, bis keine weitere Verbesserung erzielt werden kann.

3. *Ausführungsphase*: In dieser Phase wird ein Roboterprogramm in BAPS generiert und auf der Steuerung rho4 ausgeführt. Dieses Roboterprogramm enthält die Bewegung der sechs Rundreisen aus der vorangegangenen Phase. Die Ausführungszeit dieses Roboterprogramms stellt die bestmögliche Ausführungszeit für einen Roboterstandort dar. Für die Messung der Ausführungszeit in dieser Phase enthalten die BAPS-Programme Befehle gemäß Bild 8.8 (S. 128).

Aufgrund der Komplexität dieses Problems wurde mit diesem Verfahren eine Heuristik implementiert. Dies ist sinnvoll, da für die Zuordnung zwischen Schraube und Gewinde $\binom{\text{Anzahl Schrauben}}{\text{Anzahl Gewinde}} = \binom{24}{4} = \frac{24!}{20!4!} = 10626$ Möglichkeiten existieren, für die wiederum jeweils ein NP-vollständiges Rundreiseproblem [30] gelöst werden muss.

Zur Berechnung der initialen sechs Rundreisen in der Reihenfolgephase wird die Gesamtbewegung des Roboters in vier Einzelbewegungen unterteilt (Bild 9.25).

(a) Die Einzelbewegung mit der Zeit ${}^{Start}t_{S_i}$ ist die Bewegung von der Startposition zur Position für die Aufnahme der Schraube S_i (Bild 9.25a).

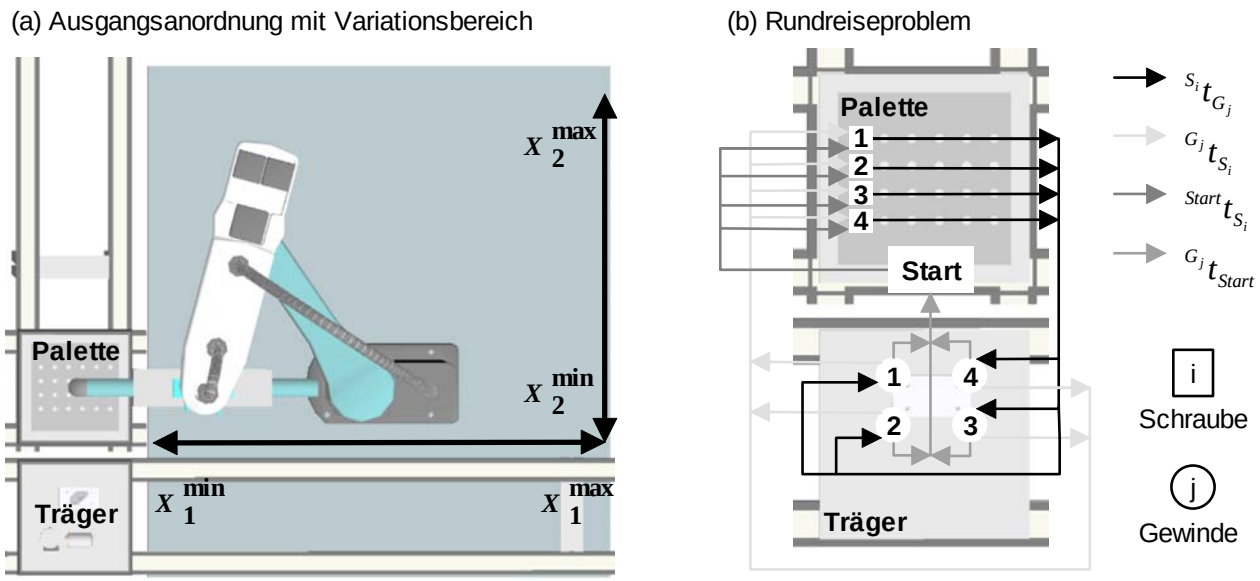


Bild 9.24: Verfahren zur Lösung des mit einem Reihenfolgeproblem kombinierten Layoutproblems

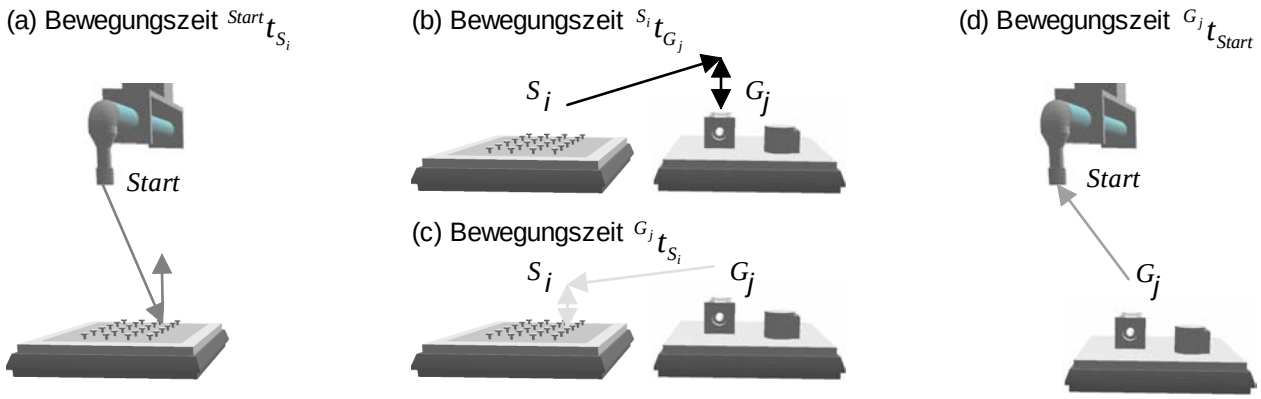


Bild 9.25: Zeiten zur Ausführung der Einzelbewegungen

(b) Die Einzelbewegung mit der Zeit ${}^{S_i}t_{G_j}$ ist die Bewegung von der Position nach der Aufnahme zur Position für den Schraubvorgang der Schraube S_i in das Gewinde G_j (Bild 9.25b).

(c) Die Einzelbewegung mit der Zeit ${}^{Gj}t_{S_i}$ ist die Bewegung nach einem Schraubvorgang an Gewinde G_j zur Position für die Aufnahme der nächsten Schraube S_i (Bild 9.25c).

(d) Die Einzelbewegung mit der Zeit ${}^{G_j}t_{Start}$ ist die Bewegung nach einem Schraubvorgang an Gewinde G_j zur Startposition (Bild 9.25d).

Zur Bestimmung der Zeiten ${}^{Start}t_{S_i}$, ${}^{G_j}t_{Start}$, ${}^{S_i}t_{G_j}$ und ${}^{Gj}t_{S_i}$ erzeugt der Plan jeweils ein Roboterprogramm und führt es auf der Standardsteuerung von COSIMIR® aus. Die vorausgesetzte, feste Zuordnung der Reihenfolgephase wird so festgelegt, dass die Schrauben einer Spalte (Spalte A-F, Bild 9.23b) jeweils demselben Gehäuse zugeordnet sind. Für jede Spalte ist ein Rundreiseproblem zu lösen. Das Rundreiseproblem der Schrauben der Spalte A ist in Bild 9.24b dargestellt. Aus Platzgründen liegen darin gemeinsame Kanten übereinander. Der Roboter muss ausgehend von einer Startposition

solange abwechselnd eine Schraube und anschließend ein Gewinde besuchen, bis er schließlich alle Schrauben im Gehäuse angebracht hat und zum Startknoten zurückkehren kann. Die Matrix zur Definition des Rundreiseproblems für die Schrauben S_{A1} bis S_{A2} der Spalte A lautet:

$$\begin{bmatrix}
\infty & \infty & \infty & \infty & {}^{G_1}t_{S_{A1}} & {}^{G_1}t_{S_{A2}} & {}^{G_1}t_{S_{A3}} & {}^{G_1}t_{S_{A4}} & {}^{G_1}t_{Start} \\
\infty & \infty & \infty & \infty & {}^{G_2}t_{S_{A1}} & {}^{G_2}t_{S_{A2}} & {}^{G_2}t_{S_{A3}} & {}^{G_2}t_{S_{A4}} & {}^{G_2}t_{Start} \\
\infty & \infty & \infty & \infty & {}^{G_3}t_{S_{A1}} & {}^{G_3}t_{S_{A2}} & {}^{G_3}t_{S_{A3}} & {}^{G_3}t_{S_{A4}} & {}^{G_3}t_{Start} \\
\infty & \infty & \infty & \infty & {}^{G_4}t_{S_{A1}} & {}^{G_4}t_{S_{A2}} & {}^{G_4}t_{S_{A3}} & {}^{G_4}t_{S_{A4}} & {}^{G_4}t_{Start} \\
{}^{S_{A1}}t_{G_1} & {}^{S_{A1}}t_{G_2} & {}^{S_{A1}}t_{G_3} & {}^{S_{A1}}t_{G_4} & \infty & \infty & \infty & \infty & \infty \\
{}^{S_{A2}}t_{G_1} & {}^{S_{A2}}t_{G_2} & {}^{S_{A2}}t_{G_3} & {}^{S_{A2}}t_{G_4} & \infty & \infty & \infty & \infty & \infty \\
{}^{S_{A3}}t_{G_1} & {}^{S_{A3}}t_{G_2} & {}^{S_{A3}}t_{G_3} & {}^{S_{A3}}t_{G_4} & \infty & \infty & \infty & \infty & \infty \\
{}^{S_{A4}}t_{G_1} & {}^{S_{A4}}t_{G_2} & {}^{S_{A4}}t_{G_3} & {}^{S_{A4}}t_{G_4} & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & {}^{Start}t_{S_{A1}} & {}^{Start}t_{S_{A2}} & {}^{Start}t_{S_{A3}} & {}^{Start}t_{S_{A4}} & \infty
\end{bmatrix}$$

Es handelt sich hier um ein asymmetrisches Rundreiseproblem: Der Roboter darf vom Startknoten ausschließlich zu einer Schraube fahren, nicht aber von einer Schraube zum Startknoten. Es gilt demnach:

$${}^{Start}t_{S_i} = const < \infty \text{ und } {}^{S_i}t_{Start} = \infty \Rightarrow {}^{Start}t_{S_i} \neq {}^{S_i}t_{Start} \tag{9.4}$$

Ebenso darf der Roboter von einem Gewinde zum Startknoten fahren, umgekehrt aber nicht. Es gilt:

$${}^{G_j}t_{Start} = const < \infty \text{ und } {}^{Start}t_{G_j} = \infty \Rightarrow {}^{G_j}t_{Start} \neq {}^{Start}t_{G_j} \tag{9.5}$$

Ebenso unterscheiden sich die Bewegungen zwischen Palette und Gehäuse. Es gilt:

$${}^{S_i}t_{G_j} \neq {}^{Gj}t_{S_i} \tag{9.6}$$

Für die Lösung des asymmetrischen Rundreiseproblems kommt ein Branch-and-Boundverfahren mit exponentiellem Aufwand zur Anwendung [113]. Das Verfahren findet eine optimale Lösung, falls eine existiert. Dieses Verfahren ist aufgrund der geringen Größe, d. h. 9 Knoten (4 Schraubenpositionen, 4 Gewindepositionen, 1 Startposition), des vorliegenden Rundreiseproblems möglich. Es ist über die Systemroutine `$TSP` anwendbar (Anhang B, S. 170).

Nach den in der Reihenfolgephase berechneten sechs Rundreisen versucht der Plan in der Verbesserungsphase diese bezüglich ihrer Gesamtausführungsdauer weiter zu optimieren. In dieser Phase wird ein so genanntes Savingsverfahren [113] angewendet. In diesem Verfahren wird eine Reduktion (Savings) berechnet, die angibt, inwieweit sich die Gesamtausführungsdauer aller sechs Rundreisen verringert, wenn man zwei Schrauben S_i und S_j innerhalb der Rundreisen vertauscht. Die Vertauschung wird als $S_i \leftrightarrow S_j$ bezeichnet. Dies hat zur Folge, dass die feste Zuordnung der ersten Phase zwischen Schrauben und Gehäusen aufgehoben wird.

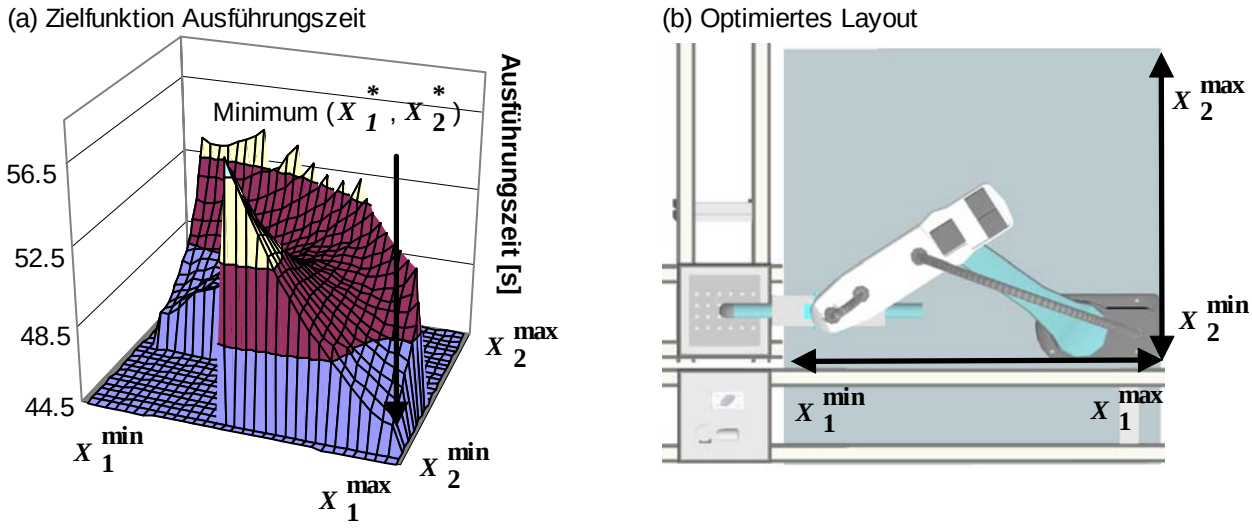


Bild 9.26: Zielfunktion und optimierte Anordnung für die Montagestation

Der Plan berechnet für alle Vertauschungen $S_i \leftrightarrow S_j$ die zugehörige Reduktion und vertauscht die beiden Schrauben, die die größte positive Reduktion der Gesamtausführungsdauer liefern. Für die aufgrund der Vertauschung $S_i \leftrightarrow S_j$ neu entstandenen sechs Rundreisen wird erneut die maximale Reduktion aller Vertauschungen berechnet und die zugehörigen Schrauben vertauscht, sofern die Reduktion positiv war. Die Verbesserungsphase endet, wenn keine positive Reduktion mehr existiert.

In der Ausführungsphase erzeugt der Plan ein BAPS-Programm, das die Roboterbewegung und die Ansteuerung des Schraubwerkzeugs für die sechs Rundreisen aus der Verbesserungsphase enthält. Das Ergebnis der Ausführungsphase zeigt Bild 9.26. In Bild 9.26a ist die sich ergebende Zielfunktion mit ihrem Minimum dargestellt. Bild 9.26b zeigt das zugehörige Layout. Tabelle 9.4 zeigt die bestmöglichen Reihenfolgen. Die gefundene minimale Ausführungszeit des auf der Steuerung rho4 ausgeführten BAPS-Programms beträgt 45.7 s. Verglichen mit der errechneten Ausführungsdauer (Tabelle 9.4, 45.0 s) aus der Verbesserungsphase, die aufgrund der Ausführung auf der Standardsteuerung von COSIMIR® beruht, ist die Abweichung 1.5 %. Damit ist die durchschnittliche Ausführungszeit für ein Gehäuse ca. 7.6 s. Die Ausführungszeit der Ausgangsanordnung (Bild 9.24a) mit der folgenden Anfangsreihenfolge beträgt 55.2 s. Bei der Anfangsreihenfolgen bearbeitet die Roboter nacheinander die Spalten A-F (Bild 9.23). In einer Spalte k lautet die Reihenfolge: S_{k1}, G_2, S_{k2}, $G_{3,}$ S_{k3}, G_3, S_{k4}, G_4. Damit benötigt die Bearbeitung eines Trägers durchschnittlich 9.2 s. Die Ausführungszeit beim optimierten Layout mit der bestmöglichen Reihenfolge ist somit um 17.4 % kleiner.

Insgesamt untersucht der Plan 725 Roboterstandorte, von den 448 Standorte erfolgreich sind, d. h., die sechs Rundreisen sind kollisionsfrei ausführbar. An jedem erfolgreichen Standort führt der Plan 220 Simulationsläufe aus, die sich aus 24 Einzelbewegungen des Typs A, $24 \cdot 4$ Einzelbewegungen des Typs B, $24 \cdot 4$ Einzelbewegungen des Typs C und 4 Einzelbewegungen des Typs C (Bild 9.25), zusammensetzen. Damit löst der Plan in der Reihenfolgephase insgesamt $448 \cdot 6 = 2688$ Rundreiseprobleme und führt $448 \cdot 220 = 98560$ Simulationsläufe aus. Dazu benötigt ein PC mit AMD™ Athlon™ 1200 MHz Prozessor, 256 MB Speicher und dem Betriebssystem Windows™ 2000, ca. 23 h. Die Gesamtzeit für die Verbesserungsphase beträgt ca. 15 min. Für die Generierung, den Download

Tabelle 9.4: Rundreisen beim optimierten Layout mit Ausführungszeiten der COSIMIR®-Steuerung

Reisen	Reihenfolge										t_{reise}
1	$Start$	S_{D3}	G_1	S_{A1}	G_2	S_{F4}	G_3	S_{B1}	G_4	$Start$	7.32 s
2	$Start$	S_{D2}	G_1	S_{B3}	G_3	S_{E3}	G_2	S_{B2}	G_4	$Start$	7.44 s
3	$Start$	S_{C4}	G_1	S_{A3}	G_3	S_{E2}	G_2	S_{C2}	G_4	$Start$	7.42 s
4	$Start$	S_{D4}	G_2	S_{A2}	G_1	S_{F2}	G_3	S_{B4}	G_4	$Start$	7.5 s
5	$Start$	S_{D1}	G_3	S_{E1}	G_2	S_{E4}	G_1	S_{C1}	G_4	$Start$	7.64 s
6	$Start$	S_{C3}	G_1	S_{A4}	G_3	S_{F1}	G_2	S_{F3}	G_4	$Start$	7.68 s
											$\sum$ 45 s

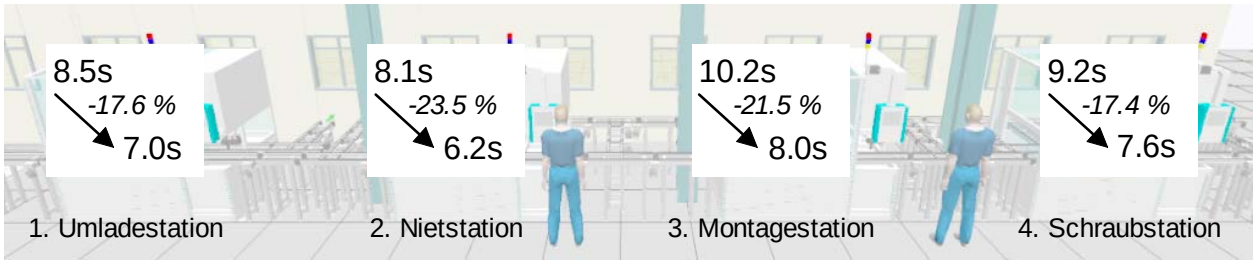


Bild 9.27: Erzielte Ausführungszeitreduktion der einzelnen Stationen

und die Ausführung der 448 BAPS-Programme in der letzten Phase benötigt der Plan ca. 12 h.

Bewertung der Versuchsergebnisse

In Bild 9.27 sind die mithilfe dieses Systems erzielten Ausführungszeiten und Reduktionen bezüglich ihrer jeweiligen Anfangsanordnungen aufgeführt. Danach liegt die mit dieser Anlage erreichbare Taktzeit, als Maximum der Taktzeiten der Einzelstationen, bei 10.2 s, weil die Montagestation den Flaschenhals darstellt.

Bemerkenswert ist die Tatsache, dass eine Reduktion der gesamten Taktzeit der Anlage, die Reduktion aller Einzelstationen erfordert. Die alleinige Optimierung der Montagestation verlagert den Flaschenhals in Richtung Schraubstation und reduziert die Anlagentaktzeit auf 9.2 s. Die zusätzliche Optimierung der Schraubstation reduziert die Anlagentaktzeit auf 8.5 s, weil damit die Umladestation den Flaschenhals bildet. Bei einer weiteren Optimierung der Umladestation wird die Nietstation zum Flaschenhals und die Anlagentaktzeit sinkt auf 8.1 s. Erst wenn alle Stationen optimiert werden, wird eine Anlagentaktzeit von 8.0 s erreicht.

Insgesamt zeigt dieses Beispiel die Leistungsfähigkeit und Flexibilität des in dieser Arbeit entwickelten Systems und den dadurch erbrachten Beitrag zur Erschließung möglicher Produktivitätssteigerungen im Umfeld von Roboter-Fertigungszellen.

10 Zusammenfassung

Sowohl die Roboterprogrammierung für variantenreiche Produkte als auch die Planung von Roboter-Fertigungszellen erfordern unter Verwendung heutiger CAR-Systeme (Computer Aided Robotics) einen hohen manuellen Aufwand. Die Ursache hierfür liegt in der Notwendigkeit der Erstellung, Ausführung und Bewertung vieler Roboterprogrammvarianten.

Um diesem Problem zu begegnen, wurde in dieser Arbeit ein System konzipiert und realisiert, mit dem die Erstellung, Ausführung und Bewertung von steuerungsspezifischen Roboterprogrammvarianten automatisiert wird.

Zentraler Bestandteil des Systemkonzepts ist der *Generierungsplan*, der festlegt, wie und welche Roboterprogrammvarianten das System erstellen soll. Damit wird es erstmals möglich, dass der CAR-Anwender den Ablauf der Programmgenerierung selbstständig formuliert. Nachdem er einen Generierungsplan für ein festgelegtes Produktspektrum erstellt hat, können auch Personen, die die umfangreichen Kenntnisse eines CAR-Anwenders nicht besitzen, neue Roboterprogramme für dieses Produktspektrum erzeugen. Das Konzept des Generierungsplans erweist sich als äußerst flexibel, weil die Pläne applikationsspezifische Anforderungen bei der Generierung berücksichtigen können, auch wenn diese erst spät, z. B. in der Inbetriebnahmephase, auftauchen.

Die Plansprache wurde bewusst so entworfen, dass sie nur prozedurale Programmierkonzepte heutiger Robotersprachen enthält, damit sie schnell erlernbar ist. Des Weiteren wurden beim Sprachentwurf darauf geachtet, dass der CAR-Anwender für die Generierung steuerungsspezifischer Programme keine Kenntnisse einer speziellen Steuerungssyntax besitzen muss. Der Ansatz der steuerungsneutralen Programmspezifikation erlaubt erstmalig die automatisierte Beurteilung des Verhaltens von Robotern verschiedener Hersteller für eine spezifische Aufgabe auf der Basis von realen Roboterprogrammen.

In der Praxis zeigte sich, das für ähnliche Applikationen Generierungspläne benötigt werden, die nur an einzelnen Stellen variieren. Um den Umgang mit derartigen Plänen zu verbessern, wurde das Konzept der *Planaddition* entwickelt. Damit ist es möglich, statt einen ähnlichen Plan neu zu formulieren, lediglich die Änderungen eines Plan anzugeben und diesen über eine Planaddition mit dem bestehenden Plan zu verknüpfen.

Um die Planerstellung für verschiedene Problemklasse zu abstrahieren, wurde das Konzept der *Planschablonen* entwickelt. Diese stellen ein Gerüst dar, mit dem die Erstellung neuer Pläne unter Einsatz der Planaddition für ähnliche Probleme vereinfacht und beschleunigt wird.

Beim Entwurf der Systemarchitektur standen die Aspekte Erweiterbarkeit und Modularität im Vordergrund. Es gelang, einen anwendungsneutralen Kern zu schaffen, der zur Planausführung *Funktionserweiterungen* verwendet, die dem CAR-Anwender häufig benötigte Funktionen bei der Planerstellung zur Verfügung stellen. Über eine leicht erlernbare Erweiterungsschnittstelle ist es möglich, neue Funktionserweiterungen zu entwickeln und an das System zu binden, ohne dass dazu weitere Systemkenntnisse erforderlich sind. Die Schnittstellenbeschreibung für eine Funktionserweiterung erfolgt in der Plansyntax um sowohl deren Inhalt zu dokumentieren als auch die Schnittstelle für den Aufruf einer einzelnen Funktion festzulegen.

Aufgrund einer Analyse von heutigen Industrierobotersteuerungen wurde die steuerungsunabhängige Syntax zur Spezifikation generierbarer Roboterprogramme entwickelt. Diese Roboterprogramme lassen sich somit auf die analysierten und auf weitere Robotersprachen abbilden. Aufgrund des darin entwickelten Variablenkonzepts können die generierten Roboterprogramme Angaben berücksichtigen, die erst zur Laufzeit und nicht bereits zum Zeitpunkt ihrer Erstellung zur Verfügung stehen.

Eine Besonderheit des Systems ist die Kapselung der Information über eine konkrete Steuerungssyntax in *Syntaxdefinitionen*. Es gelang für alle analysierten Sprachen, den Aufbau und die Syntax in einer sprachneutralen Struktur abzubilden. Durch die Darstellung dieser Struktur in Form einer Datei ist die jeweilige Steuerungssyntax leicht änderbar und nicht im System verankert. Diese Vorgehensweise ermöglicht einerseits versionsbedingte Anpassungen der Steuerungssyntax ohne weitere Eingriffe in das Gesamtsystem. Anderseits wird der Entwicklungsaufwand zur Unterstützung einer neuen Steuerungssyntax minimiert.

Im Gegensatz zu bestehenden Systemen prüft dieses System umfassend und herstellerübergreifend die fehlerfreie Ausführbarkeit der generierten, steuerungsspezifischen Roboterprogramme durch deren Ausführung in einem Simulationsmodell der Roboter-Fertigungszelle. Auch das Zusammenspiel von generierten und manuell erstellten Programmen wird erstmals berücksichtigt. Somit sind die generierten Programme ohne weitere Konvertierungen in der realen Roboter-Fertigungszelle lauffähig. Neben kinematischen Restriktionen und Kollisionen kann das System auch das Verhalten der Roboterprogramme bei Eintritt von Störungen untersuchen.

Für die Ausführung der steuerungsspezifischen Roboterprogramme kamen sowohl eine reale Robotersteuerung als auch die Standardsteuerung von COSIMIR® zum Einsatz. Letztere benötigt dazu einen steuerungsspezifischen Compiler. Um die Entwicklung solcher Compiler zu beschleunigen, gelang es, ein Framework aufzustellen, das bis auf den Parser aus einem Standardcompiler besteht. Unter Verwendung dieses Frameworks kann sich ein Compilerentwickler ausschließlich auf die besonderen Eigenschaften "seiner Sprache" konzentrieren. Der Programmier- und Wartungsaufwand für mehrere Compiler konnte auf diese Weise stark reduziert werden.

Zum ersten Mal enthält ein derartiges System eine Rückkopplung, um die Bewertung der Ausführung eines steuerungsspezifischen Roboterprogramms in einen automatisierten Ablauf einzubinden. Erst diese Vorgehensweise ermöglicht die Verbesserung einer Planungsvariante oder eines Roboterprogramms und eröffnet dem System dadurch viele Einsatzgebiete. Dadurch kann das System Optimierungspotenzial bei der Programmierung und Planung von Roboter-Fertigungszellen erschließen, das sonst ungenutzt bleibt.

Der Nachweis der Funktionstüchtigkeit und der Leistungsfähigkeit des in dieser Arbeit entworfenen und realisierten Systems wurde anhand von Applikationen aus Industrie und Forschung erbracht. Dabei wurden die Notwendigkeit und der Nutzen eines solchen Systems unter Beweis gestellt. Somit stellt das in dieser Arbeit entwickelte System ein leistungsfähiges Werkzeug dar, das den zukünftigen Anforderungen der ”digitalen Fabrik” genügt.

A Grammatik der Generierungsplansprache

Zur Beschreibung der Generierungsplansprache wird im Folgenden die *Extended Backus-Naur Form* (EBNF) verwendet. Abweichend von den Festlegungen für die EBNF in der Norm ISO/IEC 14977 [3] werden hier zur Reduktion des Umfanges der Grammatikbeschreibung an verschiedenen Stellen reguläre Ausdrücke verwendet. Die regulären Ausdrücke sind *kursiv* und die Schlüsselwörter bzw. Terminalzeichen der Sprache sind **fett** dargestellt.

GENERATION-PLAN =
    (EXTENSION-INTERFACE | USER-PLAN | ADAPTION-PLAN)

EXTENSION-INTERFACE =
    "**SYSTEM**" "**PLUGIN**" ID { SYSTEM-DECLARATIONS } { SYSTEM-ROUTINE } "**ENDPLUGIN**"

SYSTEM-DECLARATIONS =
    SYSTEM-DECLARATION "**;**" { "**;**" }

SYSTEM-DECLARATION =
    (SYSTEM-DECLARATION-VARIABLE | SYSTEM-DECLARATION-CONSTANT |
    SYSTEM-LIBRARY-STEP | SYSTEM-LIBRARY-FUNCTION | DECLARATION-STRUCTURE)

SYSTEM-DECLARATION-VARIABLE =
    SYSTEM-DATATYPE SYSTEM-IDLIST [INITIALIZATION]

SYSTEM-DECLARATION-CONSTANT =
    "**CONST**" SYSTEM-DATATYPE SYSTEM-ID INITIALIZATION

SYSTEM-LIBRARY-STEP =
    SYSTEM-ID "**(**" SYSTEM-FORMAL-PARAMETERS "**)**"

SYSTEM-LIBRARY-FUNCTION =
    SYSTEM-DATATYPE SYSTEM-ID "**(**" SYSTEM-FORMAL-PARAMETERS "**)**"

SYSTEM-ROUTINE =
    ["**PUBLIC**"] "**STEP**" ID "**(**" SYSTEM-FORMAL-PARAMETERS "**)**" BLOCK "**ENDSTEP**"

SYSTEM-DATATYPE =
    (DATATYPE | "**VOID**")

SYSTEM-FORMAL-PARAMETERS =
    [INPUT-SYSTEM-FORMAL-PARAMETERS] [OUTPUT-SYSTEM-FORMAL-PARAMETERS]

INPUT-SYSTEM-FORMAL-PARAMETERS =
     INPUT-SYSTEM-FORMAL-PARAMETER { "**,**" INPUT-SYSTEM-FORMAL-PARAMETER }

INPUT-SYSTEM-FORMAL-PARAMETER =
    ["**CONST**"] SYSTEM-FORMAL-PARAMETER

OUTPUT-SYSTEM-FORMAL-PARAMETERS =
    "**=>**" OUTPUT-SYSTEM-FORMAL-PARAMETER { "**,**" OUTPUT-SYSTEM-FORMAL-PARAMETER }

OUTPUT-SYSTEM-FORMAL-PARAMETER =
    SYSTEM-FORMAL-PARAMETER

SYSTEM-FORMAL-PARAMETER =
    SYSTEM-DATATYPE ID [INITIALIZATION]

```
ADAPTION-PLAN =
    "ADAPTION" "PLAN" ID { ADAPTION-DECLARATION } { ADAPTION-ROUTINE } "ENDPLAN"

ADAPTION-DECLARATION =
    ( ADAPTION-DECLARATION-VARIABLE | ADAPTION-DECLARATION-CONSTANT ) ";" { ";" }

ADAPTION-DECLARATION-VARIABLE =
    DATATYPE ID "::" ID-LIST [ INITIALIZATION ]

ADAPTION-DECLARATION-CONSTANT =
     "CONST" DATATYPE ID "::" ID INITIALIZATION

ADAPTION-ROUTINE =
    [ "PUBLIC" ] "STEP" ID "::" ID "(" FORMAL-PARAMETERS ")" BLOCK "ENDSTEP"

USER-PLAN =
    "GENERATION" "PLAN" ID { DECLARATION } { ROUTINE } "ENDPLAN"

DECLARATION =
    ( DECLARATION-VARIABLE | DECLARATION-CONSTANT | DECLARATION-STRUCTURE ) ";" { ";" }

DECLARATION-VARIABLE =
    DATATYPE IDLIST [ INITIALIZATION ]

DECLARATION-CONSTANT =
     "CONST" DATATYPE ID INITIALIZATION

DECLARATION-STRUCTURE =
    "TYPE" ID COMPONENT { ";" COMPONENT } "ENDTYPE"

COMPONENT =
    DATATYPE ID

ROUTINE =
    [ "PUBLIC" ] "STEP" ID "(" FORMAL-PARAMETERS ")" BLOCK "ENDSTEP"

FORMAL-PARAMETERS =
    [ INPUT-FORMAL-PARAMETERS ]  [ OUTPUT-FORMAL-PARAMETERS ]

INPUT-FORMAL-PARAMETERS  =
    INPUT-FORMAL-PARAMETER { ","  INPUT-FORMAL-PARAMETER}

INPUT-FORMAL-PARAMETER =
    [ "CONST" ] FORMAL-PARAMETER

OUTPUT-FORMAL-PARAMETERS  =
    "=>" OUTPUT-FORMAL-PARAMETER
    {  ","  OUTPUT-FORMAL-PARAMETER}

OUTPUT-FORMAL-PARAMETER  =
    FORMAL-PARAMETER

FORMAL-PARAMETER =
    DATATYPE ID [ INITIALIZATION ]

BLOCK =
    STATEMENT ";" { ";" } { STATEMENT ";" { ";" } }

STATEMENT =
```

```
    ( DECLARATION | ASSIGN | CONTROL |  EVENT | PERSISTENT )

CONTROL =
    ( CALL | LABEL | GOTO | IF | FOR | REPEAT | WHILE | SWITCH |
    "RETURN" | "BREAK" | "CONTINUE" | "EXIT"  | "STOP" )

ASSIGN =
    DESIG "=" EXPR

EVENT =
    ( ON-EVENT-CALL | BLOCK-EVENT | UNBLOCK-EVENT )

PERSISTENT =
    ( ARCHIVE | RETRIEVE )

CALL =
    PUBLIC-ID "(" [ INPUT-ACTUAL-PARAMS ] [ OUTPUT-ACTUAL-PARAMS ] ")"

INPUT-ACTUAL-PARAMS =
    ACTUAL-PARAMS

OUTPUT-ACTUAL-PARAMS =
    "=>" ACTUAL-PARAMS

ACTUAL-PARAMS =
    ACTUAL-PARAM { "," ACTUAL-PARAM}

ACTUAL-PARAM =
    [ ID "=" ] EXPR

LABEL =
    "=>" ID ":"

GOTO =
    "GOTO" ID

IF =
    "IF" EXPR BLOCK [ ELSE ] "ENDIF"

ELSE =
    ( ELSE" BLOCK | { "ELSEIF" EXPR BLOCK } )

FOR =
    "FOR" ID  "=" EXPR "TO" EXPR [ "STEP" EXPR ] LOOP

REPEAT =
    LOOP "UNTIL" EXPR

WHILE =
    "WHILE" EXPR LOOP

LOOP =
    "LOOP" BLOCK "ENDLOOP"

SWITCH =
    "SELECT" EXPR CASES [DEFAULT] "ENDSELECT"

CASES =
    CASE { CASE }
```

```
CASE =
    "CASE" EXPR-LIST ":" BLOCK

DEFAULT =
    "DEFAULT" ":" BLOCK

ON-EVENT-CALL  =
     "ON" DESIG "CALL" CALL [ WITH "PRIO" EXPR ]

BLOCK-EVENT =
     "BLOCK" " DESIG  { ","  DESIG}

UN-BLOCK-EVENT =
     "UNBLOCK" " DESIG  { ","  DESIG}

ARCHIVE =
     "PERSISTENT" "(" DESIG ")"

RETRIEVE =
     "PERSISTENT" "(" "=>" DESIG ")"

EXPR =
    EXPR-PRIO-7

EXPR-PRIO-7 =
    [ EXPR-PRIO-7 ( "==" | "<>" | "!=" | "<" | "<=" | ">" | ">=" ) ] EXPR-PRIO-6

EXPR-PRIO-6 =
    [ EXPR-PRIO-6 ( "OR" | "BOR" ) ] EXPR-PRIO-5

EXPR-PRIO-5 =
    [ EXPR-PRIO-5 ( "XOR" | "BXOR" ) ] EXPR-PRIO-4

EXPR-PRIO-4 =
    [ EXPR-PRIO-4 ( "AND" | "BAND" ) ] EXPR-PRIO-3

EXPR-PRIO-3 =
    [ EXPR-PRIO-3 ( "+" | "-"| "NOT"| "BNOT" ) ] EXPR-PRIO-2

EXPR-PRIO-2 =
    [ EXPR-PRIO-2 ( "#" | "*" | "/" | "DIV" | "MOD" ) ] EXPR-PRIO-1

EXPR-PRIO-1 =
    [ ( "+" | "-" ) ] FAKTOR

FAKTOR =
    ( CALL-EXPR | DESIG | CONST | "(" EXPR ")" )

CALL-EXPR=
    PUBLIC-ID "("  [ INPUT-ACTUAL-PARAMS ] [ OUTPUT-ACTUAL-PARAMS ]   ")"

DESIG =
    ( ID | DESIG "." ID | DESIG "[" [ EXPR ] "]" )

CONST =
    ( CONST-NUMBER | CONST-BOOL | CONST-TEXT | CONST-OBJECT | CONST-SECTION )

DATATYPE =
    ( BUILT-IN-TYPE  { "[" "]" } | USER-TYPE  { "[" "]" } )
```

USER-TYPE =
"**TYPE**" "**::**" ID

BUILT–IN-TYPE =
(STANDARD-TYPE | MODEL-TYPE | ROBOT-TYPE | GEOMETRY-TYPE | "**$EVENT**")

STANDARD-TYPE =
("**$NUMBER**" | "**$TEXT**" | "**$BOOL**")

MODEL-TYPE =
("**$OBJECT**" | "**$SECTION**")

ROBOT-TYPE =
("**$POSITION**" | "**$FRAME**" | "**$AXIS**" |"**$ORIENT**")

GEOMETRY-TYPE =
("**$PLANE**" | "**$SPHERE**" | "**$CYLINDER**" | "**$LINE**" | "**$BOX**")

INITIALIZATION =
"**=**" CONST

EXPR-LIST =
EXPR { "**,**" EXPR}

PUBLIC-IDLIST =
PUBLIC-ID { "**,**" PUBLIC-ID }

SYSTEM-IDLIST =
SYSTEM-ID { "**,**" SYSTEM-ID }

PUBLIC-ID =
(SYSTEM-ID | SYSTEM-ID "**::**" ID)

SYSTEM-ID =
(PURE-SYSTEM-ID | ID)

IDLIST =
ID { "**,**" ID }

CONST-OBJECT =
"<" ID ">"

CONST-SECTION =
"<" ID "**.**" ID ">"

PURE-SYSTEM-ID =
$[A-Za-z_][A-Za-z0-9_]

ID =
[A-Za-z_][A-Za-z0-9_]

CONST-TEXT =
"([^"])"*

CONST-BOOL =
("**TRUE**" | "**FALSE**")

CONST-NUMBER=
([0-9]* | *[0-9]+"."[0-9]+(E[-+]?[0-9]+)?* | *"."[0-9]+(E[-+]?[0-9]+)?*
| *[0-9]+(E[-+]?[0-9]+)?* | *[0-9]+"."(E[-+]?[0-9]+)?*)

B Auszug vordefinierter Plananweisungen

Das in dieser Arbeit entwickelte System stellt Plananweisungen (Basisfunktionen) zur Verfügung, die den CAR-Anwender bei der Planerstellung entlasten sollen. Hier sind die in dieser Arbeit verwendeten Plananweisungen alphabetisch aufgeführt und kurz erläutert.

Basis

`$ADD_TAIL(VOID A[], $VOID E => VOID[] B)`
Feld `B[]` enthält Elemente von `A[]` und als letztes Element `E`

`$VECTOR $CREATE_VECTOR($NUMBER X, $NUMBER Y, $NUMBER Z);`
Erzeugung eines Vektors aus X-, Y- oder Z-Koordinatenwerten

`$VECTOR $CREATE_VECTOR($NUMBER X, ...,$NUMBER YAW);`
Erzeugung eines Frames aus X-, Y-, Z-, Roll-, Pitch-, Yaw-Koordinatenwerten

`$FRAME $INVERSE($FRAME F)`
Liefert das inverse Frame zu dem Frame `F`

`$NUMBER $MAX($NUMBER[] N => $NUMBER INDEX)`
Liefert Maximum mit dessen `INDEX` aus Zahlenarray `N`

`$NUMBER $MIN($NUMBER[] N= > $NUMBER INDEX)`
Liefert Minimum mit dessen `INDEX` aus Zahlenarray `N`

`$REMOVE_MULTIPLE_ELEMENTS(VOID A[] => VOID[] B)`
Feld `B[]` enthält alle Elemente von Feld `A[]`, aber jedes nur einmal.

`$REMOVE_TAIL(VOID A[], $NUMBER I => VOID[] B)`
Feld `B[]` enthält Elemente von 1 bis `I` aus `A[]`

`$REVERSE(VOID[] AB => VOID[] BA)`
Umdrehen der Reihenfolge eines Feld `AB`

`$NUMBER $SIZE(VOID[] A)`
Liefert Anzahl der Elemente eines Felds `A`

`$NUMBER $SORT_DISTANCE(POSITION[] A, POSITION P => POSITION[] B)`
Sortiert Feld `A[]`, beginned bei Position `P`, in das Feld `B[]` bezüglich ihrer Distanz

`$NUMBER $X($VECTOR[] A)`
`$NUMBER $Y($VECTOR[] A)`
`$NUMBER $Z($VECTOR[] A)`
Liefert jeweils ein Feld, das die X-, Y- oder Z-Koordinatenwerte eines Punktefelds `A` enthält.

Parameter

`$GET_PARAMETER(=> VOID P)`
Abfrage eines Parameters `P` beim Eingabeassistenten

`$PARAMETER(=> VOID P)`
Registrierung von Parameter `P` zur Abfrage beim Eingabeassistenten

`$GET_ALL_PARAMETER()`
Abfrage der registrierten Parameter beim Eingabeassistenten

Optimierung

`$INIT_VARIATION($NUMBER[] MAX,$NUMBER[] MIN,$NUMBER[] NI=>$NUMBER N)`
Initialisierung `$VARIATION` : $x_i^{\min}$ (`MAX`), $x_i^{\max}$ (`MIN`), n_i^x (`NI`) gemäß Formel 5.3 (S. 72), liefert Anzahl der Variationen `N`

`$NUMBER $MAX_ITERATION`
Variable zur Begrenzung der Zahl der Zielfunktionsberechnungen im Schritt `$OPTIMIZATION`

`$OPTIMIZATION($TEXT F, $NUMBER P[] => $NUMBER MIN)`
Berechnung des Minimums `MIN` einer Funktion `F` (anwenderdefinierter Schritt) beginned bei `P`

`$TEXT $STRATEGY`
Variable zur Defintion des Optimierungsverfahrens für `$OPTIMIZATION`

`$TSP($NUMBER[][] M => $NUMBER P, $NUMBER K)`
Lösung des Rundreiseproblems: Kostenmatrix `M`, optimaler Pfad `P` mit Kosten `K`

`$OPTIMIZATION($TEXT ZF, $NUMBER X[] => $NUMBER LM, $NUMBER[] MS)`
Minimiere der Zielfunktion `ZF` mit Startvektor `X[]`, lokalem Minimum `LM` an Stelle `MS`

`$BOOL $VARIATION(=> X[])`
Bei jedem Aufruf Berechnung einer Variation gemäß Formel 5.3 (S. 72)

Datei

`$OPEN($TEXT N => $NUMBER F)`
Öffnen einer Datei `F` mit Namen `N`

`$CLOSE($NUMBER F)`
Schließen einer Datei `F`

`$READ_LINE($NUMBER F => $TEXT S)`
Einlesen einer Zeile `S` aus Datei `F`

`$TOKENIZE($TEXT S, $TEXT D =>T[])`
Teilen eines Textes `S` in Token `T[]` durch die Begrenzungzeichen aus `D`

Geometrie

`$BALANCE_POINT($VECTOR[] P => $VECTOR S)`
Berechnung des Schwerpunkts `S` aus einer Punktmenge `P`

`$BOUNDARY($VECTOR[][] G => $VECTOR[] R)`
Berechnung des äußeren Randes `R` von einer Menge planarer Polygone

`$BOUNDING_BOX($VECTOR[][] G => $BOX B)`
Berechnung des kleinsten, die Geometrie `G` umhüllenden Quaders `B`

`$BOOL $CUTS_LINE_PLANE(L,P => S)`
Erzeugung eines Schnittpunkts `S` zwischen einer Geraden `L` und einer Ebene `P`

`$CENTRAL_PROJECTION($VECTOR P, $VECTOR C, $PLANE E => $VECTOR S)`
Berechnung des auf die Ebene `E` projizierten Punkt `S` eines Punkts `P` bezüglich dem Zentrum `C`

`$VECTOR $CROSS($VECTOR A, $VECTOR B)`
Liefert das Kreuzprodukt zu den Vektoren `A` und `B`

`$CUT_LINE_POLYGONS($LINE L, $VECTOR[][] G => $VECTOR[] S)`
Erzeugung von Schnittpunkten `S` durch Schnitt einer Geraden `L` mit der Geometrie `G`

`$CUT_PLANE_POLYGONS($PLANE P, $VECTOR[][] G => $VECTOR[] S)`
Erzeugung von Schnittpunkten `S` durch Schnitt einer Ebene `P` mit der Geometrie `G`

`$CUT_POLYGONS($VECTOR[][] P => $VECTOR[] S)`
Erzeugung des kleinsten Polygons `S`, das alle (planaren) Polygone aus `P` enthalten

`$LINES_2D($VECTOR P1, $VECTOR P2, $NUMBER N=> $LINE[] L)`
Erzeugung von `N` äquidistante Geraden in `L`, die alle zwschen `P1` und `P2` liegen

`$VECTOR $NORM($VECTOR V)`
Liefert den Normalenvektor zum Vektor `V`

`$NEAREST_POINT($VECTOR[] P, $VECTOR R => $VECTOR PN, $NUMBER I)`
Liefert den zu `R` nächstgelegenen Punkt `PN` mit seinem Index `I` in `P`

`$NEAREST_POINT_ON_POLYGON($VECTOR[] POLY, $VECTOR P => $VECTOR PN)`
Berechnung des zu einem Punkt `P` nächstgelegenen Punktes `PN` auf einem Polygon `POLY`

`$BOOL $POINT_INSIDE_POLYGON($VECTOR[] POLY, $VECTOR P)`
Liegt Punkt `P` in Polygon `POLY`?

`$SCAN_POLYGON($VECTOR[] P, $NUMBER A, $VECTOR C => $VECTOR[] S)`
Abtasten eines Polygons `P` in äquidistanten Winkelabständen `A` um Punkt `C`

`$SHRINK_POLYGON($VECTOR[] P, $NUMBER D => $VECTOR[] S)`
Schrumpfen eines Polygons `P` um den Betrag `D`

`$STRETCH_POLYGON($VECTOR[] P, $NUMBER D => $VECTOR[] S)`
Dehnen eines Polygons `P` um den Betrag `D`

`$TRIANGULIZE($VECTOR[] P => $VECTOR[][] T)`
Berechnung von traingulierten Polygonen `T` aus einem Polygon `P`

Synthese

Für Programmbefehle siehe Tabelle 7.2 (S. 104).

`$GENERATE()`
Ausgabe der steuerungsspezifischen Roboterprogramms

`$TEXT $LANGUAGE`
Einstellen der Sprache zur Programmgenerierung

`$LOAD_SYNTAX()`
Laden der Syntaxdefinition für die in `$LANGUAGE` angegene Sprache

`$RESET()`
Abbau der internen Repräsentation der Roboterprogramme im Modul Programmsynthese

Simulation

`$EVENT $COLLISION`
Ereignis bei Einritt einer unerwünschten Kollision

`$COLLISION_EXCEPTION($SECTION G, $OBJECT O)`
Angabe einer erwünschten Kollision zwischen einer Gruppe `G` und einem Objekt `O`

`$LOAD()`
Laden des Roboterprogramms in die virtuelle Robotersteuerung

`$EVENT $MALFUNCTION`
Ereignis bei Einritt einer Störung

`$EVENT $PROGRAM_ERROR`
Ereignis bei Einritt eines Programmfehlers

`$RUNTIME($OBJECT R => $NUMBER T)`
Auslesen der Ausführungszeit des Roboters im letzten Simualtionslauf

`$BOOL $SIMULATION_OK`
Ergebnis, ob Simualtionslauf fehlerfrei

`$SIMULATION()`
Starten eines Simualtionslaufs

`$STOP()`
Stoppen eines Simualtionslaufs

`$STORE_TO_TIDY()`
Speichern des Simulationsmodell, um dieses nach Änderung mit `$TIDY` wiederherzustellen

`$TIDY()`
Wiederhestellen des Simulationsmodell das mit `$STORE_TO_TIDY` gespeichert wurde

`$EVENT $VRC`
Ereignis bei Einritt eines Fehlers während der Programmausführung

Modell

`$OBJECT $CURRENT_ROBOT`
Name des aktuellen Roboters im Simulationsmodell

`$POSITION $CURRENT_ROBOT_POSITION`
Kartesische Position des aktuellen Roboters

`$FRAME $CURRENT_ROBOT_PLACE`
Standort des aktuellen Roboters im Simulationsmodell

`$EXCHANGE($OBJECT OLD = <CurrentRobot>, $OBJECT NEW)`
Austausch einer Komponenten (Standard: aktueller Roboter) durch eine neue Komponente

`$GET_DO($OBJECT O, $TEXT N => $NUMBER NR)`
Lesen der Nummer `Nr` des digitalen Ausgangs `N` von Objekt `O` aus dem Simualtionsmodell
Falls `N` nicht angegeben, wird standardmäßig der Namen der Variablen `NR` verwendet

`$GET_GEOMETRY($OBJECT O => $VECTOR[][] G)`
Lesen der Geometrie `G` eines Objektes `O` aus dem Simualtionsmodell

`$GET_GRIPPOINT($OBJECT O, $TEXT N = "1" => $OBJECT G)`
Lesen des (ersten) Greifpunkts `G` mit Namen `N` von Objekt `O` aus dem Simualtionsmodell

`$GET_POS($TEXT N => $POSITION P)`
Lesen einer vordefinierten Position `P` mit Namen `N`

`$GET_POSITIONLIST($TEXT N => $POSITION[] P)`
Lesen von vordefinierten Positionen `P[]` aus einer Liste mit Namen `N`

`$BOOL $IS_REACHABLE($POSITION P)`
Erreicht der aktuelle Roboter die Position `P`?

`$BOOL $IS_MODEL_COLLISIONFREE()`
Treten im Simulationsmodell unerwünschte Kollisionen auf?

`$MOVE_ROBOT($POSITION P)`
Setzen des aktuellen Roboters auf Position `P`

`$NUMBER $NUMBER_OF_ROBOTS`
Variable, die die Anzahl der Roboter im Simultionsmodell enthält

`$PLACE($OBJECT O, $FRAME F)`
Setzen des Standorts von Objekt `O` auf Ort `F`

`$ROTATE($OBJECT O, $NUMBER ROLL, $NUMBER PITCH, $NUMBER YAW)`
Drehen von Objekt `O` um die Winkel `ROLL`, `PITCH`, `YAW`

`$SET_GRIPPOINT($OBJECT O, $OBJECT GP)`
Hinzufügen eines Greifpunkts `GP` mit Namen `N` zu Objekt `O` in das Simualtionsmodell

Visualisierung

`$COLOR($TEXT N)`
Festlegung der Farbe für visalisierte Elemente

`$VISUALIZE(VOID E, $FRAME F)`
Geometrisches Element (Fläche, Quader, Gerade, ...) bezgl Koordinatensystem `F` darstellen

`$HIDE(VOID E)`
Ausblenden eines eingeblendeten Elements

C Planimplementierungen

Der Anhang enthält die beiden Implementierungen der in Kapitel 5 vorgestellten Schablonen und die für die einleitenden Polierapplikation (Bild 1.1, S. 2) benötigten Plananpassungen.

C.1 Programmgenerierung für variantenreiche Produkte

Dieser Abschnitt zeigt zunächst die Implementierung der Schablone zur Erstellung von Plänen, die das Systen zur Programmgenerierung für variantenreiche Produkte benötigt. Die Schritte, die keine Anweisungen enthalten (z. B. der Schritt `VARIATION`) wurden aus Platzgründen weggelassen. Anschließend ist die Implementierung einer Plananpassung zur Programmgenerierung für die Produktvarianten aus dem einleitenden Beispiel (Bild 1.1, S. 2) angegeben.

C.1.1 Schablone

```
GENERATION PLAN PRODUKT
   $VECTOR[][]   GEOMETRIE;
   $POSITION[][] BAHN;
   $BOOL         OK=TRUE;

   STEP START()                        --- Beginn der Planausführung
     INITIALISIERUNG();
     LOOP
     PARAMETERANFORDERUNG();
     ZUGRIFF_SIMULATIONSMODELL();
     RECHENVORSCHRIFT();
     PRÜFUNG_VOR_SIMULATION(=>OK);
     IF NOT(OK)
       VARIATION();
       CONTINUE;
     ENDIF;
     PROGRAMMSPEZIFIKATION();
     GENERIERUNG();
     INITIALISIERUNG_SIMULATION();
     SIMULATION();
     BEWERTUNG(=>OK);
     IF NOT(OK)
       VARIATION();
     ENDIF;
     ENDLOOP UNTIL OK;
   ENDSTEP

   STEP INITIALISIERUNG()
     ON $COLLISION CALL SIMULATION_ERROR_HANDLER();
     ON $VRC CALL SIMULATION_ERROR_HANDLER();
   ENDSTEP

   STEP PARAMETERANFORDERUNG()
     $GET_ALL_PARAMETER();
   ENDSTEP

   STEP ZUGRIFF_SIMULATIONSMODELL()
     ZUGRIFF_SIMULATIONSMODELL_LESEND();
     ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND();
   ENDSTEP
```

```
  STEP RECHENVORSCHRIFT()
    BERECHNUNG_STÜTZPUNKTE();
    ZUORDNUNG_ORIENTIERUNG();
    REDUKTION_BAHNPOSITIONEN();
    ZUORDNUNG_BEZUG();
    BERECHNUNG_AN_UND_ABFAHRWEGE();
    UMRECHNUNG_EXTERNES_WERKZEUG();
    ZUORDNUNG_KONFIGURATION_UND_TURN();
  ENDSTEP

  STEP BERECHNUNG_STÜTZPUNKTE()
    BERECHNUNG_SCHNITTGEOMETRIE();
    SCHNITTBERECHNUNG();
    SORTIERUNG_STÜTZPUNKTE();
    NACHBEARBEITUNG();
  ENDSTEP

  STEP PRÜFUNG_VOR_SIMULATION(=>$BOOL OK=FALSE)
    $NUMBER I, J=1;
    FOR I = 1 TO $SIZE(BAHN[][]) LOOP
      FOR J = 1 TO $SIZE(BAHN[I][]) LOOP
        IF $IS_REACHABLE(BAHN[I][J])
          $MOVE_ROBOT(BAHN[I][J]);
          IF $IS_MODEL_COLLISIONFREE()
            CONTINUE;
          ENDIF;
        ENDIF;
        RETURN;
      ENDLOOP;
    ENDLOOP;
    OK = TRUE;
  ENDSTEP

  STEP GENERIERUNG()
    $LOAD_SYNTAX();
    $GENERATE();
    $RESET();
    $LOAD();
  ENDSTEP

  STEP INITIALISIERUNG_SIMULATION()
    $MOVE_ROBOT(BAHN[1][1]);
  ENDSTEP

  STEP SIMULATION()
    UNBLOCK $COLLISION, $VRC;
    $SIMULATION_OK = TRUE;
    $SIMULATION();
    BLOCK $COLLISION, $VRC;
  ENDSTEP

  STEP SIMULATION_ERROR_HANDLER()
    BLOCK $COLLISION, $VRC;
    $STOP();
    $SIMULATION_OK = FALSE;
    UNBLOCK $COLLISION, $VRC;
  ENDSTEP

  STEP BEWERTUNG(=> $BOOL SIMULATION_OK)
    SIMULATION_OK=$SIMULATION_OK;
  ENDSTEP

  STEP ZUGRIFF_SIMULATIONSMODELL_LESEND()
    LESEN_WERKSTÜCK_GEOMETRIE();
    LESEN_ROBOTER_INFORMATIONEN();
    LESEN_WEITERE_INFORMATIONEN();
  ENDSTEP
ENDPLAN
```

C.1.2 Anpassung für einleitende Polierapplikation

```
ADAPTION PLAN PRODUKT

   $OBJECT PRODUKT::VARIANTE = <Produkt C>;       -- Name im Modell
   $NUMBER PRODUKT::WINKEL = 10.0;                -- Anstellwinkel (Grad)
   $NUMBER PRODUKT::GESCHWINDIGKEIT = 50.0;       -- Bahn (mm/s)
   $NUMBER PRODUKT::D= 3;                         -- Geradenabstand (mm)
   $NUMBER PRODUKT::ANFAHRDISTANZ= 20;            -- Anfahrdistanz
   $OBJECT PRODUKT::MASCHINE = <Poliermaschine>;  -- Name im Modell
   $NUMBER PRODUKT::AN;                           -- Ausgang zur MASCHINE
   $FRAME  PRODUKT::GREIFPUNKT;                   -- Produktgreifpunkt
   $LINE[] PRODUKT::LINE_X, LINE_Y;               -- Schnittgeraden
   $NUMBER PRODUKT::I, J;                         -- Zählvariablen


   STEP PRODUKT::INITIALISIERUNG()
     => PREVIOUS STEP;
     $LANGUAGE = "MBA4";           -- Programmiersprache
   ENDSTEP


   STEP PRODUKT::PARAMETERANFORDERUNG()
     $PARAMETER(=>VARIANTE);             -- Eingabe der Variante
     $PARAMETER(=>WINKEL);               -- Eingabe des Winkels
     $PARAMETER(=>GESCHWINDIGKEIT);      -- Eingabe der Geschwindigkeit
     => PREVIOUS STEP;                   -- PARAMETERANFORDERUNG S.175
   ENDSTEP


   STEP PRODUKT::LESEN_WERKSTÜCK_GEOMETRIE()
     $GET_GEOMETRY(VARIANTE=>GEOMETRIE[][]);  -- Geometriedaten lesen
   ENDSTEP


   STEP PRODUKT::LESEN_ROBOTER_INFORMATIONEN()
     $GET_DO(MASCHINE=>AN);                      -- Ausgang lesen
   ENDSTEP


   STEP PRODUKT::LESEN_WEITERE_INFORMATIONEN()
     $GET_GRIPPOINT(VARIANTE=>GREIFPUNKT);     -- Greifpunkt lesen
   ENDSTEP


   STEP PRODUKT::ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND()
     $ROTATE(<Kontaktpunkt>, YAW=WINKEL); -- Kontaktpunkt ändern
     $GRIP(<Magnetgreifer>, VARIANTE);    -- Greifer greift Variante
   ENDSTEP


   STEP PRODUKT::BERECHNUNG_SCHNITTGEOMETRIE()
     $NUMBER X_MIN, Y_MIN, Z_MIN, X_MAX, Y_MAX, Z_MAX;
     $BOUNDING_BOX(GEOMETRIE[][]=>X_MIN,Y_MIN,Z_MIN,X_MAX,Y_MAX,Z_MAX);
     $NUMBER Z = (Z_MAX-Z_MIN)/2;
     $NUMBER NX = $INT((X_MAX-X_MIN)/D);
     $NUMBER NY = $INT((Y_MAX-Y_MIN)/D);
     $VECTOR P0= $CREATE_VECTOR(X_MIN, Y_MIN, Z);
     $VECTOR P1= $CREATE_VECTOR(X_MAX, Y_MIN, Z);
     $VECTOR P2= $CREATE_VECTOR(X_MIN, Y_MAX, Z);
     $LINES_2D(P0, P1, NX => LINE_X[]);
     $LINES_2D(P0, P2, NY => LINE_Y[]);
   ENDSTEP
```

```
STEP PRODUKT::SCHNITTBERECHNUNG()
  $NUMBER      I_MIN, I_MAX;
  $VECTOR[]    S;
  $POSITION    P_MAX, P_MIN;
  FOR I = 1 TO $SIZE(LINE_Y[]) LOOP
    $CUT_LINE_POLYGONS(LINE_Y[I], GEOMETRIE[][]=>S[]);
    $NUMBER MIN      = $MIN($X(S[])=>I_MIN);
    $NUMBER MAX      = $MAX($X(S[])=>I_MAX);
    P_MIN.FRAME.VEC  = S[I_MIN];
    P_MAX.FRAME.VEC  = S[I_MAX];
    $ADD_TAIL(BAHN[1][], P_MIN=>BAHN[1][]);
    $ADD_TAIL(BAHN[1][], P_MAX=>BAHN[1][]);
  ENDLOOP
  FOR I = 1 TO $SIZE(LINE_X[]) LOOP
    $CUT_LINE_POLYGONS(LINE_X[I], GEOMETRIE[][] => S[]);
    $NUMBER MIN     = $MIN($Y(S[]) => I_MIN);
    $NUMBER MAX     = $MAX($Y(S[]) => I_MAX);
    P_MIN.FRAME.VEC = S[I_MIN];
    P_MAX.FRAME.VEC = S[I_MAX];
    $ADD_TAIL(BAHN[1][], P_MIN => BAHN[1][]);
    $ADD_TAIL(BAHN[1][], P_MAX => BAHN[1][]);
  ENDLOOP
  $REMOVE_MULTIPLE_ELEMENTS(BAHN[1][] => BAHN[1][]);
ENDSTEP


STEP PRODUKT::SORTIERUNG_STÜTZPUNKTE()
  $SORT_DISTANCE(BAHN[1][1], BAHN[1][] => BAHN[1][]);
ENDSTEP


STEP PRODUKT::ZUORDNUNG_ORIENTIERUNG()
  $NUMBER   N = $SIZE(BAHN[1][]);
  $ORIENT   O;
  FOR J = 1 TO N-1 LOOP
    O.U = $NORM(BAHN[1][J+1].FRAME.VEC-BAHN[1][J].FRAME.VEC);
    O.W = $CREATE_VECTOR(0, 0, 1);
    O.V = $CROSS(O.W, O.U);
    BAHN[1][I].FRAME.ORI = O;
  ENDLOOP;
  O.U = $NORM(BAHN[1][1].FRAME.VEC-BAHN[1][N].FRAME.VEC);
  O.W = $CREATE_VECTOR(0, 0, 1);
  O.V = $CROSS(O.W, O.U);
  BAHN[1][N].FRAME.ORI = O;
ENDSTEP


STEP PRODUKT::ZUORDNUNG_BEZUG()
  FOR I = 1 TO $SIZE(BAHN[1][]) LOOP
    BAHN[1][I].RELATIVETO=VARIANTE;
  ENDLOOP;
ENDSTEP


STEP PRODUKT::BERECHNUNG_AN_UND_ABFAHRWEGE()
  $POSITION ANFAHRPOS = BAHN[1][1];
  ANFAHRPOS.FRAME = ANFAHRPOS.FRAME*$CREATE_FRAME(Y=ANFAHRDISTANZ);
  $ADD_HEAD(BAHN[1][], ANFAHRPOS => BAHN[1][]);
  $ADD_TAIL(BAHN[1][], ANFAHRPOS => BAHN[1][]);
ENDSTEP


STEP PRODUKT::UMRECHNUNG_EXTERNES_WERKZEUG()
  FOR I = 1 TO $SIZE(BAHN[1][]) LOOP
    BAHN[1][I].FRAME=$INVERSE(BAHN[1][I].FRAME)*GREIFPUNKT;
    BAHN[1][I].RELATIVETO=<Kontaktpunkt>;
  ENDLOOP;
ENDSTEP
```

```
   STEP PRODUKT::ZUORDNUNG_KONFIGURATION_UND_TURN()
     FOR I = 1 TO $SIZE(BAHN[1][]) LOOP
       BAHN[1][I].RIGHTLEFT  = $RIGHT;
       BAHN[1][I].ABOVEBELOW = $ABOVE;
       BAHN[1][I].FLIPNOFLIP = $NOFLIP;
     ENDLOOP;
   ENDSTEP

   STEP PRODUKT::PROGRAMMSPEZIFIKATION()
     $PROJECT("POLIEREN");
     $MAINPROGRAM();
       $MAINROUTINE();
         $SET_DIG_OUTPUT(AN);             -- Poliermaschine einschalten
         $PATH_SPEED(GESCHWINDIGKEIT);    -- Poliergeschwindigkeit
         $PATH(BAHN[1][]);                -- Polierbahn abfahren
         $RESET_DIG_OUTPUT(AN);           -- Poliermaschine ausschalten
       $ENDMAINROUTINE();
     $ENDMAINPROGRAM();
   ENDSTEP
ENDPLAN
```

C.2 Layoutoptimierung

Dieser Abschnitt zeigt zunächst die Implementierung der Schablone zur Erstellung von Plänen, die das System zur Layoutoptimierung benötigt. Anschließend wird die Implementierung einer Plananpassung zur Layoutoptimierung für das einleitende Beispiel (Bild 1.1, S. 2) vorgestellt.

C.2.1 Schablone

```
GENERATION PLAN LAYOUT
   $POSITION[][]  BAHN;
   $OBJECT[]      OBJEKT;
   $NUMBER[]      MIN, MAX, AUFLÖSUNG, LOKALES_MINIMUM;
   $FRAME[]       INITIALER_OBJEKT_STANDORT;
   $NUMBER[][]    ÄNDERUNG;
   $NUMBER        I, J, K = 1;
   $BOOL          OK;

   STEP START()
     INITIALISIERUNG();
     ZUGRIFF_SIMULATIONSMODELL_LESEND();
     WHILE $VARIATION(=> ÄNDERUNG[K][]) LOOP
       ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND(ÄNDERUNG[K][]);
       PRÜFUNG_VOR_SIMULATION(=> OK);
       IF OK
         $OPTIMIZATION("ZIELFUNKTION", ÄNDERUNG[K][]
                   => LOKALES_MINIMUM[K], ÄNDERUNG[K][]);
       ENDIF;
       K = K + 1;
     ENDLOOP;
     ERGEBNIS();
   ENDSTEP
```

```
STEP ZIELFUNKTION($NUMBER[] PARAM => $NUMBER FOPT, $BOOL LIMIT=TRUE)
  ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND(PARAM[]);
  PRÜFUNG_VOR_SIMULATION(=> OK);
  IF OK
    PROGRAMMSPEZIFIKATION();
    GENERIERUNG();
    INITIALISIERUNG_SIMULATION();
    SIMULATION();
    BEWERTUNG(=> LIMIT, FOPT);
  ENDIF;
AUFRÄUMEN();
ENDSTEP

STEP INITIALISIERUNG()
  ON $COLLISION CALL SIMULATION_ERROR_HANDLER();
  ON $VRC CALL SIMULATION_ERROR_HANDLER();
  $INIT_VARIATION(MIN[], MAX[], AUFLÖSUNG[]);
  FOR I = 1 TO $SIZE(OBJEKT[]) LOOP
    INITIALER_OBJEKT_STANDORT[I] = $GET_PLACE(OBJEKT[I]);
  ENDLOOP;
  $STORE_TO_TIDY();
ENDSTEP

STEP ZUGRIFF_SIMULATIONSMODELL_LESEND()
  LESEN_ROBOTER_POSITIONEN();
  LESEN_WEITERE_INFORMATIONEN();
ENDSTEP

STEP ZUGRIFF_SIMULATIONSMODELL_SCHREIBEND($NUMBER[] ÄNDERUNG)
  $FRAME[] OBJEKT_ÄNDERUNG;
  VARIATION_ZU_OBJEKTEN(ÄNDERUNG[] => OBJEKT_ÄNDERUNG[]);
  OBJEKTE_ANORDNEN(OBJEKT_ÄNDERUNG[]);
ENDSTEP

STEP ERGEBNIS()
  $NUMBER I_BEST;
  $FRAME  BESTES_LAYOUT[];
  $NUMBER BESTE_BEWERTUNG = $MIN(LOKALES_MINIMUM[] => I_BEST);
  VARIATION_ZU_OBJEKTEN(ÄNDERUNG[I_BEST][] =>BESTES_LAYOUT[]);
  OBJEKTE_ANORDNEN(BESTES_LAYOUT[]);
  PROGRAMMSPEZIFIKATION();
  GENERIERUNG();
ENDSTEP

STEP OBJEKTE_ANORDNEN($FRAME[] ÄNDERUNG)
  FOR I = 1 TO $SIZE(OBJEKT[]) LOOP
    $FRAME STANDORT = INITIALER_OBJEKT_STANDORT[I]*ÄNDERUNG[I];
    $PLACE(OBJEKT[I], STANDORT);
  ENDLOOP;
ENDSTEP

STEP GENERIERUNG()
... -- Schablone "Programmgenerierung für variantenreiche Produkte"
ENDSTEP

STEP PRÜFUNG_VOR_SIMULATION(=>$BOOL OK=FALSE)
... -- Schablone "Programmgenerierung für variantenreiche Produkte"
ENDSTEP

STEP INITIALISIERUNG_SIMULATION()
  $MOVE_ROBOT(BAHN[1][1]);
ENDSTEP

STEP SIMULATION()
... -- Schablone "Programmgenerierung für variantenreiche Produkte"
ENDSTEP

STEP SIMULATION_ERROR_HANDLER()
... -- Schablone "Programmgenerierung für variantenreiche Produkte"
ENDSTEP
```

```
   STEP BEWERTUNG(=>$BOOL LIMIT, $NUMBER EVAL)
     LIMIT = NOT($SIMULATION_OK);
     $RUNTIME($CURRENT_ROBOT =>EVAL);
   ENDSTEP

   STEP AUFRÄUMEN()
     $TIDY();
   ENDSTEP
 ENDPLAN
```

C.2.2 Anpassung für einleitende Polierapplikation

```
ADAPTION PLAN LAYOUT
   $OBJECT LAYOUT::PRODUKT                = <VarianteC>;
   $OBJECT LAYOUT::MASCHINE               = <Poliermaschine>;
   $NUMBER LAYOUT::AN, AUF;
   $NUMBER LAYOUT::GESCHWINDIGKEIT=50.0;

   STEP LAYOUT::INITIALISIERUNG()
     $LANGUAGE = "MBA4";
     $STRATEGY = "HOOKE-JEEVES";
     OBJEKT[1] = MASCHINE; --- Variation der Poliermaschine
     MIN[1] = -250; MAX[1] = 750; AUFLÖSUNG[1] = 200; -- Welt X
     MIN[2] = -100; MAX[2] = 600; AUFLÖSUNG[2] = 100; -- Welt Y
     $COLLISION_EXCEPTION(<Poliermaschine.Scheibe>, PRODUKT);
     => PREVIOUS STEP;
   ENDSTEP

   STEP LAYOUT::LESEN_ROBOTER_POSITIONEN()
     $GET_POS("ÜberAufnahme" => BAHN[1][1]);
     $GET_POS("Aufnahme" => BAHN[2][1]);
     $GET_POSITIONLIST(LISTNAME = "Polierbahn" => BAHN[3][]);
   ENDSTEP

   STEP LAYOUT::LESEN_WEITERE_INFORMATIONEN()
     $GET_DO(MASCHINE => AN);
     $GET_DO(<Magnetgreifer> => AUF);
   ENDSTEP

   STEP LAYOUT::VARIATION_ZU_OBJEKTEN($NUMBER[] ÄNDERUNG
                              => $FRAME[] OBJEKT_ÄNDERUNG)
     OBJEKT_ÄNDERUNG[1].VEC.X = ÄNDERUNG[1];
     OBJEKT_ÄNDERUNG[1].VEC.Y = ÄNDERUNG[2];
   ENDSTEP

   STEP LAYOUT::PROGRAMMSPEZIFIKATION()
     $PROJECT("POLIEREN");
       $MAINPROGRAM();
         $MAINROUTINE();
           $PTP(BAHN[1][1]);              -- Roboter über Aufnahme
           $LINEAR(BAHN[2][1]);           -- Roboter bei Aufnahme
           $SET_DIG_OUTPUT(AUF);          -- Platte aufnehmen
           $LINEAR(BAHN[1][1]);           -- Roboter über Aufnahme
           $PTP(BAHN[3][1]);              -- Start Polierbahn
           $SET_DIG_OUTPUT(AN);           -- Poliermaschine einschalten
           $PATH_SPEED(GESCHWINDIGKEIT);  -- Poliergeschwindigkeit
           $PATH(BAHN[3][]);              -- Polierbahn abfahren
           $RESET_DIG_OUTPUT(AN);         -- Poliermaschine ausschalten
           $PTP(BAHN[1][1]);              -- Roboter über Aufnahme
           $LINEAR(BAHN[2][1]);           -- Roboter bei Aufnahme
           $RESET_DIG_OUTPUT(AUF);        -- Platte ablegen
         $ENDMAINROUTINE();
       $ENDMAINPROGRAM();
     ENDSTEP
   ENDPLAN
```

Literatur

[1] ISO/TR 10562: Manipulating Industrial Robots - Intermediate Code for Robots (ICR), Beuth-Verlag, 1995.

[2] DIN 66312, Teil 1: Industrieroboter - Programmiersprache - Industrial Robot Language (IRL), 1996.

[3] ISO/IEC 14977: EBNF Notation - Final Draft Version - SC22/N2249, 1996.

[4] DIN 66314: IRDATA - Schnittstelle zwischen Programmiersystem und Robotersteuerung. Allgemeiner Aufbau, Satztypen und Übertragung, Beuth-Verlag, 1997.

[5] CATALOG: Der Leitfaden rund um die digitale Produktentwicklung, IWT Magazin Verlags-GmbH, Vaterstetten, 2002.

[6] ABB: RAPID Referenzhandbuch, Ausgabe 3HAB 7548 - 1, ABB Robotics, Väseteras, Schweden, 1997.

[7] ABB: RobotStudio™: True Off-line Programming, Produktinformation, ABB Flexible Automation GmbH, 2001.

[8] ADEPT: V+ Language - Reference Guide Version 12.1, Adept Technology Inc., San Jose, Kalifornien, USA, September 1997.

[9] AHLBEHRENDT, N.; PAUL, L.; BÜCHNER, K.; NEUMANN, A.: 3D-MPS: Modellierungs-, Programmier- und Simulationssystem zum Glasurauftrag in der keramischen Industrie, CAD 2000 - Kommunikation, Kooperation, Koordination, Berlin, März 2000.

[10] AHO, A. V.; SETHI, R.; ULLMANN, J. D.: Compilerbau, Addison-Wesley, 1999.

[11] AHRENS, G.: Software Companies Respond to Needed Improvements in Simulation, Offline Programming, Robotics World, Bd. 19, Nr. 2, S. 26–35, März 2001.

[12] AUTOCAM: MOSES Roboter-Offline-Programmierung, AUTOCAM Informationstechnik GmbH, Dortmund, Produktinformation, 2001.

[13] BARRAL, D.; PERRION, J.-P.; DOMBRE, E.; LIEGEOIS, A.: Development of Optimisation Tools in the Context of an Industrial Robotic CAD Software Product, International Journal of Advanced Manufacturing Technology, Bd. 15, S. 822–831, 1999.

[14] BARRAL, D.; PERRION, J.-P.; DOMBRE, E.; LIEGEOIS, A.: Simulated Annealing Combined with a Constructive Algorithm for Optimising Assembly Workcell Layout, International Journal of Advanced Manufacturing Technology, Bd. 17, S. 593–602, 2001.

[15] BASTERT, R.: Entgraten mit Industrierobotern, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, April 1990.

[16] BAUER, B.; HÖLLERER, R.: Übersetzung objektorientierter Programmiersprachen, Springer-Verlag, 1998.

[17] BAUER, R.: Öffnung einer Arbeitszellensimulation zur Anbindung externer Steuerungen über industrielle Kommunikationssysteme, Dissertation, Institut für Roboterforschung, Universität Dortmund, April 2003.

[18] BERNHARDT, R.; SCHRECK, G.; WILLNOW, C.: Virtual Robot Controller (VRC) Interface, Robotik 2000, S. 115–120, VDI-Verlag, Berlin, April 2000.

[19] BERNHARDT, R.; SCHRECK, G.; ZANDER, H.: Off-line Programming of Robots for Arc Welding in Shipyards, Robotics '94 - Flexible Production - Flexible Automation, Proceedings of the 25th International Symposium on Industrial Robots, S. 477–484, Hannover, April 1994.

[20] BÖHM, R.: Praktikumsbericht, Technische Universität Chemnitz, FLEXIVA automation & Anlagenbau, September 2002.

[21] BICKENDORF, J.: Full-Automatic Off-Line Programming of Complex Cutting Paths - A Contribution to the Economic Production of 'Lotsize 1', Robotics '94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 499–506, Hannover, April 1994.

[22] BICKENDORF, J.: Featurebasierte CAD/CAM-Kopplung für das Profilschneiden mit Robotern, VDI-Z, Bd. 142, Nr. 5, S. 51–53, Mai 2000.

[23] BICKENDORF, J.: Neue Anwendungsgebiete für Industrieroboter durch CAD-basierte Offline-Programmierung, Robotik 2000, S. 121–126, VDI-Verlag, Berlin, April 2000.

[24] BICKENDORF, J.: Integration der Offline-Programmierung in die Prozesskette, Robotik 2002, VDI-Berichte Nr. 1679, S. 237–242, VDI-Verlag, Düsseldorf, Ludwigsburg, Juni 2002.

[25] BLUME, C.; JAKOB, W.: Programming Languages for Industrial Robots, Springer-Verlag, Heidelberg, Berlin, New York, 1986.

[26] BOLEY, D.: Sensorgestütztes Programmierverfahren für das Entgraten mit Industrierobotern, Dissertation, IPA-IAO Forschung und Praxis Nr. 127, Fraunhofer-Institut für Produktionstechnik und Automatisierung (IPA), Stuttgart, Mai 1988.

[27] BOSCH: BOSCH rho4 BAPS32 Programmieranleitung, Software-Handbuch, Robert Bosch GmbH, Erbach, Ausgabe 102, 1999.

[28] BYG SYSTEMS: Grasp 2000, BYG Systems LTD, Produktinformation, April 2001.

[29] CARAT: FAMOS® Offline-Programmierung und Prozeßoptimierung für Industrieroboter, Produktinformation, carat robotic innovation GmbH, Dortmund, 2001.

[30] CLAUS, V.; SCHWILL, A.: Duden Informatik - Ein Sachlexikon für Studium und Praxis, Bibliografisches Institut, Mannheim, 2001.

[31] CLOCKSIN, W. F.; MELLISH, C. S.: Programming in Prolog, Springer-Verlag, 2000.

[32] COMPUCRAFT: RobotWorks, Compucraft Ltd., Produktinformation, April 2002.

[33] DAI, W.: PC-basierte, interaktive Off-line-Programmierung von Schweissrobotern mit verbesserter Nutzerunterstützung, Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, Mai 2000.

[34] DAMMERTZ, R.: Ein Programmiersystem zur graphisch strukturierten Erstellung von Roboterprogrammen und Programmieroberflächen, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, August 1996.

[35] DAMSBO, M.; RUHOFF, P.T.: An Evolutionary Algorithm for Welding Task Sequence Ordering, CALMET, J.; PLAZA, J.A. (Hrsg.), Artificial Intelligence and Symbolic Computation, Bd. 1476, S. 120–131, Plattsburgh, New York, USA, September 1998.

[36] DELMIA: IGRIP®, Delmia, Produktinformation, April 2002.

[37] DILLMANN, R.; HUCK, M.: Informationsverarbeitung in der Robotik, Springer-Verlag, Heidelberg, Berlin, New York, 1991.

[38] DOU, S.: Off-line Programmierung von Industrierobotern für die konturgebende Fertigung, VDI Fortschritt-Berichte Nr. 646, Dissertation, VDI-Verlag, Düsseldorf, Universität Hannover, Februar 1997.

[39] DUBROWSKY, S.; BLUBAUGH, T.D.: Planning time-optimal robotic manipulator motions and work places for point-to-point tasks, IEEE Transactions on Robotics and Automation, Bd. 5, S. 377–381, Juni 1989.

[40] EASY-ROB: EASY-ROB 3D Robot Simulation Tool, Produktinformation, EASY-ROB™, Frankfurt/Main, 2001.

[41] ERICSSON, A.; GRÖNDAHL, P.; ONORI, M.: An Application with the MARK III Assembly System, Robotics '94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 127–132, Hannover, April 1994.

[42] FEDROWITZ, C.H.: Is there a future for simulation and offline programming in the scope of small and medium sized companies, Proceedings of the 29th International Symposium on Robotics (ISR), Birmingham, Grossbritannien, April 1998.

[43] FLETCHER, R.: Practical Methods of Optimization, John Wiley & Sons, New York, USA, 1987.

[44] FLOW SOFTWARE TECHNOLOGIES: WORKSPACE: Eine neue Generation von PC Software für die Roboterprogrammierung, Flow Software Technologies Inc., Produktinformation, 2001.

[45] FREUND, E.; HECK, H.; KREFT, K.; MAUVE, CH.: OSIRIS - Ein objektorientiertes System zur impliziten Roboterprogrammierung, Robotersysteme, Bd. 6, S. 185–192, Juni 1990.

[46] FREUND, E.; HEINZE, F.; HYPKI, A.: Real-time Coupling of the 3D Workcell Simulation System COSIMIR®, Proceedings of the First IEEE International Conference on Information Technology and Applications (ICITA2002), Bathurst, Australien, Nov. 2002.

[47] FREUND, E.; HYPKI, A.; KEIBEL, A.: Merging Simulation and Control of Industrial Robot Workcells, Proceeding of the International NAISO Congress on Information Science Innovations (ISI), S. 457–452, Dubai, Vereinigte Arabische Emirate, März 2001.

[48] FREUND, E.; LÜDEMANN-RAVIT, B.: A System to Automate the Generation of Program Variants for Industrial Robot Applications, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002), Bd. 2, S. 1856–1861, Lausanne, Schweiz, Oktober 2002.

[49] FREUND, E.; LÜDEMANN-RAVIT, B.; PENSKY, D.: Process-Oriented Modeling for Automatic Generation of Robot Programs, Proceedings of the 17th International Conference on CAD/CAM, Robotics and Factories of the Future (CARS and FOF 2001), S. 9–16, ISPE/IEE, Durban, Südafrika, Juli 2001.

[50] FREUND, E.; LÜDEMANN-RAVIT, B.; PENSKY, D.: Automatisierte Layout- und Programmgenerierung für Roboter in industriellen Anwendungen, Robotik 2002, Ludwigsburg, VDI-Berichte Nr. 1679, S. 219–224, VDI-Verlag, Düsseldorf, Juni 2002.

[51] FREUND, E.; LÜDEMANN-RAVIT, B.; STERN, O.; KOCH, T.: Creating the Architecture of a Translator Framework for Robot Programming Languages, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2001), S. 187–192, IEEE, Seoul, Südkorea, Mai 2001.

[52] FREUND, E.; PENSKY, D.; LÜDEMANN-RAVIT, B.: Automatic Generation of Models and Programs for Robot-Based Manufacturing Processes, Proceedings of the International Symposium on Manufacturing Technology (ISMT 2001) on the 3rd International ICSC-NAISO World Manufacturing Congress (WMC 2001), NAISO, Rochester, New York, USA, April 2002.

[53] FREUND, E.; STERN, O.; LÜDEMANN-RAVIT, B.: A System Architecture for the Translation and Automatic Testing of Industrial Robot Programs, Proceedings of the 32nd International Symposium on Robotics (ISR 2001), Bd. 1, S. 216–221, Seoul, Südkorea, April 2001.

[54] FREUND, E; UTHOFF, J.; HYPKI, A.; VAN DER VALK, U.: COSIMIR und PCROB: Integration von Zellensimulation und Robotersteuerung auf PCs, VDI-Berichte 1094, intelligente Steuerung und Regelung von Robotern, VDI-Verlag, Düsseldorf, November 1993.

[55] FRIEDRICH, H.; HOLLE, J.; DILLMANN, R.: Interactive Generation of Flexible Robot Programs, IEEE International Conference on Robotics and Automation, Leuven, Belgien, Mai 1998.

[56] FROMMHERZ, B.: Ein Roboteraktionsplanungssystem, Dissertation, Institut für Prozessrechentechnik (IPR), Universität Karlsruhe, September 1990.

[57] FU, K.S.; GONZALEZ, R.C.; LEE, C.S.G.: Robotics - Control, Sensing, Vision, and Intelligence, McGraw-Hill, Inc., 1987.

[58] FUCHSBERGER, A.: Untersuchung der spanenden Berarbeitung von Knochen, Dissertation, iwb-Forschungsberichte, Band 2, Technische Universität München, 1988.

[59] FUNDER, J.: Automatische Bahngenerierung für thermische Schneidaufgaben mit komplexen Industrierobotersystemen, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, September 1996.

[60] GLAVINA, B.: Planung kollisionsfreier Bewegungen für Manipulatoren durch Kombination von zielgerichteter Suche und zufallsgesteuerter Zwischenzielerzeugung, Dissertation, Technische Universität München, 1991.

[61] GRUBE, G.: Schleifen mit Industrierobotern, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, März 1991.

[62] GRUENEIS, C.; RICHTER, R.; HENNING, F.: Clinical Introduction of the CASPAR System: Problems and Initial Results, 4th International Symposium on Computer Assisted Orthopaedic Surgery, S. 19, Davos, Schweiz, März 1999.

[63] GUNNARSSON, K.: New Concepts and Ideas for Robotic Simulation and Off-line Teaching Systems, Robotics Online, Robotic Industries Association, März 2000.

[64] HAMMES, F.: Ein Beitrag zur Verbesserung der Programmierung von Schweissrobotern unter Verwendung von verfahrensangepassten Modulen, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, 1988.

[65] HAMURA, M.; MIZUNO, T.: Robot Deburring System, Robotics ’94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 443–446, Hannover, April 1994.

[66] HARTFUSS, C.: Wissensbasierte Programmierung von Industrierobotern zum Schutzgasschweissen im Stahlhochbau, Dissertation, IPA-IAO Forschung und Praxis Nr. 225, Fraunhofer Institut für Produktionstechnik und Automatisierung (IPA), Stuttgart, Sept. 1995.

[67] HEIM, A.: Modellierung, Simulation und optimale Bahnplanung bei Industrierobotern, Dissertation, Technische Universität München, 1999.

[68] HENRICH, D.; WÖRN, H.; WURLL, C.: Automatic Off-line Programming and Motion Planning for Industrial Robots, Proceedings of the 29th International Symposium on Robotics (ISR 1998), Advanced Robotics: Beyond 2000, Birmingham, Großbritannien, 1998.

[69] HEROLD, H.: lex und yacc: Lexikalische und syntaktische Analyse, Addison-Wesley (Deutschland), 1995.

[70] HESSE, S.: Industrieroboterpraxis, Vieweg-Verlag, Braunschweig, Wiesbaden, Juni 1998.

[71] HOLLENBERG, F.: CAD-basierte Off-line Programmierung von Lichtbogenschweissrobotern, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, August 1995.

[72] HOOKE, R.; JEEVES, T.A.: Direct Search Solution of Numerical and Statistical Problems, Journal of the Association for Computing Machinery, Bd. 8, Nr. 2, S. 212–229, 1961.

[73] HÖRATH, I.: Das Band läuft - virtuell, Computer Zeitung, Nr. 14, März 2003.

[74] HÖRMANN, K.: Kollisionsfreie Bahnen für Industrieroboter, Dissertation, Informatik- Fachbericht Nr. 166, Universität Karlsruhe, 1987.

[75] HÖRMANN, K.; WERLING, V.: Ein Verfahren zur Planung von Feinbewegungen für Montageoperationen, Robotersysteme, Bd. 6, Nr. 2, S. 119–125, 1990.

[76] HSU-CHANG, L.; LÜTH, T.; NNAJI, O.; PRINZ, M.: From CAD-based Kinematic Modeling to Automated Robot Programming, Robotics and Computer-Integrated Manufacturing, Bd. 12, Nr. 1, S. 99–109, März 1996.

[77] HUCK, M.: Produktorientierte Montageablauf- und Layoutplanung, Dissertation, VDI-Fortschritt-Berichte Nr. 279, Universität Karlsruhe, 1990.

[78] HUMBURGER, R.: Konzeption eines Systems zur aufgabenorientierten Roboterprogrammierung, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, April 1998.

[79] IMAM, M.: Geometrische Methoden zur aufgabenorientierten Lackierplanung, Dissertation, Reihe Produktionstechnik Berlin - Band 118, Technische Universität Berlin, Hanser-Verlag, München, Wien, 1993.

[80] INVISION: INVISION Virtual Process Control, intro Industrieautomation, Rücker AG, Ravensburg, Produktionformation, März 2001.

[81] JUNG, C.: Beitrag zum flexibel automatisierten Entgraten mit Industrierobotern, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, 1996.

[82] KAMPKER, M.: New Ways of User-Oriented Robot Programming, IEEE International Conference on Robotics and Automation, S. 1936–1940, Mai 1998.

[83] KAMPKER, M.: Werkzeuge für die verbesserte, nutzerorientierte Werkstattprogrammierung von Schweissrobotern, Dissertation, Technisch-wissenschaftlicher Bericht Nr. 32, Rheinisch-Westfälische Technische Hochschule Aachen, Juli 2000.

[84] KEIBEL, A.: Konzeption und Realisierung eines integrierten Moduls zur Simulation und Steuerung von Kinematiksystemen, Dissertation, Institut für Roboterforschung, Universität Dortmund, April 2003.

[85] KERNEBECK, U.: Automatische Handlungsplanung für Mehrrobotersysteme, Dissertation, Institut für Roboterforschung, Universität Dortmund, 1995.

[86] KIPPELS, D.: Nur gut geschulte Mitarbeiter nutzen CAE-Software voll, VDI Nachrichten, April 2003.

[87] KLINGSTAM, P.; GULLANDER, P.: Overview of Simulation Tools for Computer-Aided Production Engineering, Computers in Industry, Bd. 38, Nr. 3, S. 173–186, 1999.

[88] KÄMPFER, S.: Techno-Push und Market-Pull halten die Robotik auf Trab, VDI Nachrichten, S. 14, Nr. 8, Februar 2002.

[89] KÄMPFER, S.: Grenzen setzt dem Roboter nur die Phantasie, VDI Nachrichten, S. 11, Nr. 6, Februar 2003.

[90] KOBAYASHI, M.; OGAWA, S.; KOIKE, M.: Off-line teachung system od a robor cell for steel pipe processing, Advanced Robotics, Bd. 15, Nr. 3, S. 327–331, Juni 2001.

[91] KUGELMANN, D.: Aufgabenorientierte Offline-Programmierung von Industrierobotern, Dissertation, Technische Universität München, Juli 1999.

[92] KUKA: KRL (KUKA Robot Language), Reference Guide Release 1.0.0, KUKA Roboter GmbH, Augsburg, Oktober 1996.

[93] KUKA: Offline-Programmierung: KRC1 Office PC-Simulation: KR SIM, KUKA Roboter GmbH, Augsburg, Produktinformation, 2000.

[94] KUKA: KUKA Sim, KUKA Roboter GmbH, Augsburg, Produktinformation, April 2002.

[95] LAHMER, A.; WIESEL, U.; BÖRNER, M.: Experiences in Using the ROBODOC System without Pins, 4th International Symposium on Computer Assisted Orthopaedic Surgery, S. 20, International Society for Computer Assisted Orthopaedic Surgery, Davos, Schweiz, März 1999.

[96] LAUGIER, C.: Planning robot motions in the Sharp System, RAVANI, B. (Hrsg.), CAD Based Programming for Sensory Robots, Bd. F50 von *NATO ASI Series*, S. 151–187, Springer-Verlag, Heidelberg, Berlin, New York, 1988.

[97] LEVI, P.: Planen für autonome Montageroboter, Bd. 191, Springer-Verlag, Heidelberg, Berlin, New York, 1988.

[98] LIEBENOW, D.: Ein Beitrag zur Makroprogrammierung in der automatischen schweisstechnischen Fertigung, Dissertation, Lehr- und Forschungsgebiet Prozesssteuerung in der Schweisstechnik der RWTH Aachen, Aachen, Dezember 1991.

[99] LIU, M.H.: Wissensbasiertes, Fuzzy-Logik-gestütztes Entgraten von Werkstücken mit kraft- und positionsgeregelten Industrierobotern, Dissertation, Technische Hochschule Darmstadt, 1995.

[100] LOZANO-PEREZ, T.: A Simple Motion-Planning Algorithm for General Robot Manipulators, IEEE Journal of Robotics and Automation, Bd. 3, S. 224–238, 1987.

[101] LÜTH, T.: Automatisierte Layoutplanung von Roboter-Fertigungszellen, Dissertation, Institut für Prozessrechentechnik, Karlsruhe, Februar 1993.

[102] LUSZEK, G.: Entwicklung einer Systematik zur Schweissablaufplanung als Teil der Off-line-Programmierung von Bahnschweissrobotern, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, 1990.

[103] MACFARLANE, S.: Whats wrong with this picture? Robot simulation is so obvious, but the industry wonders why sales still aren't taking off, IT for Industry, Online Magazine, Dez. 2000.

[104] MEHLHORN, K.; NÄHER, S.: LEDA - A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[105] MEISSNER, A.: Aufgabenorientierte Programmierung von Bewegungsbahnen für Grossroboter mit redundanter Gelenkarmkinematik, Dissertation, IPA-IAO Forschung und Praxis Nr. 321, Universität Stuttgart, 2000.

[106] MITSUBISHI: Mitsubishi Industrial Robot CR2/CR4/CR7 Controller - Instruction Manual - Detailed Explanation of Functions and Operations, Mitsubishi Electric Europe B.V.- Factory Automation, Ratingen, 2001.

[107] MITSUBISHI: Movemaster Industrieroboter Bedienungs- und Programmieranleitung CR1/CR2, Mitsubishi Electric Europe B.V.- Factory Automation, Ratingen, August 2001.

[108] MOSEMANN, H.: Beiträge zur Planung, Dekomposition und Ausführung von automatisch generierten Roboteraufgaben, Dissertation, Fortschritte in der Robotik Band 6, Technische Universität Braunschweig, März 2000.

[109] MYERS, D: An Approach to Automated Programming of Industrial Robots, Proceedings of the IEEE International Conference on Robotics and Automation, S. 3109–3114, Mai 1999.

[110] MYERS, D.; PRITCHARD, M.; BROWN, M.: Automated Programming of an Industrial Robot through Teach-By Showing, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2001), S. 4078–4083, Seoul, Korea, Mai 2001.

[111] NAGAO, Y.; URABE, H.; HONDA, F.; KAWABATA, J.; KAWAMURA, T.; MIYAMOTO, N.: Development od a teachingless robot system for welding a large-sized box-type construction, Advanced Robotics, Bd. 15, Nr. 3, S. 287–291, Juni 2001.

[112] NARDMANN, M.: eM-OLP - Schulungsunterlagen zur OLP-Schulung, Fachhochschule Osnabrück, Oktober 2001.

[113] NEUMANN, K.; MORLOCK, M.: Operations Reseach, Hanser-Verlag, München, 1993.

[114] NIS: Roboplan: Das wirtschaftliche Offline-Programmiersystem für Schweissroboter, NIS - Norddeutsche Informations-Systeme GmbH, Raisdorf, Produktinformation, 1997.

[115] N.N.: Und es rentiert sich doch - Offline-Programmierung: Wirtschaftlich bei kleinen Serien, Roboter, S. 58–59, April 1994.

[116] N.N.: COSIMIR® Manufacturing - Systeme für die digitale Fabrik, EF-Robotertechnik, Schwerte, Produktinformation, April 2001.

[117] N.N.: PROARC - CAD-Based Programming System for Arc Welding Robots in One- Off Production Runs, Copernicus Projekt 7831, Final Report, Januar 2001.

[118] N.N.: Erfolgsgeschichte: 30 Jahre Roboterkompetenz bei der Bosch Rexroth AG, drive & control, Januar 2003.

[119] NNAJI, O.: Theory of Automatic Robot Assembly and Programming, Chapmann & Hall, 1993.

[120] OLSCHEWSKI, U.: Roboter-Montagesysteme, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, 1990.

[121] ONORI, M.: The Robot Motion Module: A Task-Oriented Robot Programming System for FAA Cells, Dissertation, The Royal Institute of Technology, Stockholm, Schweden, 1996.

[122] ONORI, M.; ERICSSON, A.; ARNSTRÖM, A.: Practical Implementation of F.A.C.E: Flexible Assembly Control Environment, Robotics ’94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 485–490, Hannover, April 1994.

[123] ORTO MAQUET: CASPAR Computer Assisted Surgical Planning and Robotics, Orto Maquet, Rastatt, Produktinformation, 1999.

[124] OSCARSSON, J.: Towards Realistic Computer Aided Robotics with Simulation of Process Variations, Proceedings of Robotikdagarna, Linköping, Schweden, Mai 1997.

[125] OSCARSSON, J.: Production System Design using Advanced Computer Aided Robotics, Proceedings of the 32st CIRP International Seminar on Manufacturing Systems; Networked Manufacturing; Integrated Design, Prototyping and Rapid Fabrication, S. 252–257, Berkley, Kalifornien, USA, Mai 1998.

[126] PARK, H.-P.: Rechnerbasierte Montageplanung in der Mittelserienfertigung, Dissertation, Institut für Fertigungstechnik und Spanende Werkzeugmaschinen (IFW), Hannover, Juni 1992.

[127] PASHKEVICH, A.: Genetic Algorithms in Computer-Aided Design of Robotic Manufacturing Cells, 3rd Oregon Symposium on Logic, Design and Learning, Portland, Oregon, USA, Mai 2000.

[128] PEPER, S.: Schweissstruktur-orientierte Offline-Programmierung von Lichtbogenschweissrobotern, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, April 1998.

[129] PERSOONS, W.: Model-based Off-line Programming of Robots, Dissertation, Katholieke Universiteit, Leuven, Belgien, Juni 1997.

[130] PETRY, M.: Systematik zur Entwicklung eines modularen Programmbaukastens für robotergeführte Klebeprozesse, Dissertation, iwb-Forschungsberichte, Band 44, Technische Universität München, 1992.

[131] QUARTIER, F.: Ein Beitrag zur nutzerorientierten Bedienung und rechnergestützten Werkstattprogrammierung von Schweissrobotern, Dissertation, Technisch-Wissenschaftlicher Bericht Nr. 26 der Prozesssteuerung in der Schweisstechnik, Rheinisch-Westfälische Technische Hochschule Aachen, Januar 1996.

[132] REINISCH, H.: Planungs- und Steuerungswerkzeuge zur impliziten Geräteprogrammierung in Roboterzellen, Dissertation, Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik, Erlangen, 1992.

[133] RÖHRDANZ, F.: Modellbasierte automatisierte Greifplanung, Dissertation, Fortschritte in der Robotik Band 3, Technische Universität Braunschweig, 1998.

[134] ROBOTICS, F.: PalletTool® PC and PalletTool®, Produktinformation, April 2001.

[135] ROKOSSA, D.: Prozessorientierte Offline-Programmierung von Industrierobotern, Dissertation, Institut für Roboterforschung, Dortmund, Dezember 1999.

[136] ROSELL, J.; GRATACOS, J.; BASANEZ, L.: An Automatic Programming Tool For Robotic Polishing Tasks, IEEE International Symposium on Assembly and Task Planning, S. 250–255, Porto, Portugal, Juli 1999.

[137] ROSSMANN, J.: Echtzeitfähige, kollisionsvermeidende Bahnplanung für Mehrrobotersysteme, Dissertation, Institut für Roboterforschung, Dortmund, Februar 1993.

[138] ROTHERT, B.: Entwurf und Implementierung eines Compilers für die höhere Roboterprogrammiersprache IRL, Diplomarbeit, Universität Dortmund, Institut für Roboterforschung, 1992.

[139] RUBINOVITZ, J.; WYSK, R.A.: Task Level Offline-Programming for Robotic Arc Welding - An Overview, Journal of Manufacturing Systems, Bd. 7, Nr. 4, S. 293–306, 1988.

[140] RUDLOFF, D.: Ein Beitrag zu Automatisierung der Programmierung von Industrierobotern, Dissertation, Lehrstuhl für Maschinenelemente, -gestaltung und Handhabungstechnik, Universität Dortmund, 1995.

[141] SCHMID, P.: Arbeitsprogrammgenerierung zum Schutzgasschweissen mit Industrierobotersystemen im Schiffbau, Dissertation, IPA-IAO Forschung und Praxis Nr. 214, Universität Stuttgart, 1995.

[142] SCHRAFT, R. D.; HARTFUSS, C.: Knowledge-based Programming of Industrial Robots for Arc Welding of Small Lot Sizes, Robotics '94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 427–434, Hannover, April 1994.

[143] SCHRÖDER, C.: Integration von Sensorik in die visuelle Roboterprogrammierung, Dissertation, Fortschritt-Berichte pak, Robotik, Band 6, Universität Kaiserslautern, 2002.

[144] SCHRÖER, F.W.: The GENTLE Compiler Construction System, R. Oldenbourg Verlag, München, Wien, 1997.

[145] SCHWINN, W.: Verfahren zur Optimierung des Robotereinsatzes in Automatisierungssystemen, Dissertation, Lehrstuhl für Systemtheorie der Elektrotechnik, Saarbrücken, Febr. 1992.

[146] SHENG, W.; XI, N.; SONG, M.; CHEN, Y.: Optimization in Automated Surface Inspection of Stamped Automotive Parts, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002), Bd. 2, S. 1850–1855, Lausanne, Schweiz, Okt. 2002.

[147] SHIMOGA, K. B.: Robot Grasp Analysis, International Journal of Robotics Research, Bd. 15, Nr. 3, S. 230–266, Juni 1996.

[148] SIEGERT, H.-J.; BOCIONEK, S.: Robotik: Programmierung intelligenter Roboter, Springer-Verlag, Heidelberg, Berlin, New York, 1996.

[149] SILMA: CimStation™ Robotics, SILMA - A Divison of Adept Technology Inc., Produktinformation, 1998.

[150] SILMA: CimStation™ Robotics - User's Manual, Version 2.6, SILMA/Adept Technology Inc., 1998.

[151] SL AUTOMATISIERUNGSTECHNIK GMBH: Mechatronisches Systemkonzept MSK 2100, Produktionformation, Iserlohn, April 2003.

[152] SPENCER, R.: 2002 Robotics World Industry Survey, Robotics World, Bd. 20, Nr. 7, S. 19–26, September 2002.

[153] SPUR, G.; ZANDER, H.: Off-line Planungsverfahren für Bahnschweissaufgaben am Beispiel Schiffbau, PRITSCHOW, G.; SPUR, G.; WECK, M. (Hrsg.), Roboteranwendung für die flexible Fertigung, S. 167–183, Hanser-Verlag, 1994.

[154] SPUR, G.; ZANDER, H.: Optimale Bewegungsplanung für Industrieroboter nach dem Kriterium der minimalen Verfahrzeit, PRITSCHOW, G.; SPUR, G.; WECK, M. (Hrsg.), Roboteranwendung für die flexible Fertigung, S. 63–82, Hanser-Verlag, 1994.

[155] STETTER, R.: Rechnergestützte Simulationswerkzeuge zur Effizienzsteigerung des Industrierobotereinsatzes, Dissertation, iwb-Forschungsberichte, Band 62, Technische Universität München, Heidelberg, Berlin, New York, 1993.

[156] STÄUBLI: VCAT Computer Aided Trajectories: Software zur computergestützten Generierung von Bahnen, Stäubli Unimation Inc., Produktinformation, April 2001.

[157] TEBIS: Tebis ist CAD plus CAM, Tebis Technische Informationssystem AG, Gräfelfing/München, Produktinformation, 2002.

[158] TECNOMATIX: eM-Workplace™, Produktinformation, Tecnomatix Technologies Ltd., 2001.

[159] THOMSEN, M.; ALDINGER, P.; GÖRTZ, W.; LUKOSCHEK, M.; LAHMER, A.; HONL., M.; BIRKE, A.; NÄGERL, H.; EWERBECK, V.: Die Bedeutung der Fräsbahngenerierung für die roboterassistierte Implantation von Hüftendoprothesenschäften, Unfallchirurg, Bd. 104, S. 692–699, 2001.

[160] UNITED NATIONS: The Profitability of Industrial Robots: Analysis of Case Studies, World Robotics: Statistics, Market Analysis, Forecasts, Case Studies and Profitability of Robot Investments, S. 265–267, International Federation of Robotics, New York, USA, 2002.

[161] UTHOFF, J.: Offenes, modulares System zur zellenorientierten Robotersimulation, Dissertation, VDI-Fortschritt-Berichte Nr. 279, Institut für Roboterforschung, Universiät Dortmund, Juli 1998.

[162] VAN DER VALK, U.: Konzeption und Realisierung einer PC-basierten Robotersteuerung, Dissertation, Institut für Roboterforschung, Dortmund, Mai 1995.

[163] VETORAZZI, C.N. JR.; TELLES, G.N.: Robot program generation for PCB (printed circuit board) component insertion via CAD system, IEEE International Symposium on Industrial Electronics, S. 339 –344, Santiago, Chile, Mai 1994.

[164] WAGNER, R.: Robotersysteme zum kraftgeführten Entgraten grobtolerierter Leichtmetallwerkstücke mit Fräswerkzeugen, Dissertation, Bericht aus dem Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen, Band 125, Universität Stuttgart, Oktober 1998.

[165] WALDEMAR LINK GMBH & CO: Operatives Vorgehen, SP II Modell Lubinus - Anatomisch angepaßtes Hüftprothesensystem, S. 16-20, Hamburg, Juli 1996.

[166] WALTHER, T.: Modellierung und Simulation der Lage von Gewebestrukturen, Diplomarbeit, Institut für Roboterforschung, Universität Dortmund, September 2001.

[167] WECK, M.; PEPER, S.: Werkstattorientierte Roboterprogrammierung für die Montage modularer Spannvorrichtungen, PRITSCHOW, G.; SPUR, G.; WECK, M. (Hrsg.), Roboteranwendung für die flexible Fertigung, S. 150–166, Hanser-Verlag, 1994.

[168] WEEKS, J.: Entwicklung eines aufgabenorientierten Greif- und Bahnplanungssystems für die automatisierte Montage mit SCARA-Robotern, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, 1996.

[169] WHITTAKER, S.: Robotic Off-Line Progrmming for Rapid Prototyping, International Symposium on Robotics, Birmingham, Grossbritannien, April 1998.

[170] WOENCKHAUS, C.: Rechnergestütztes System zur automatisierten Layoutoptimierung, Dissertation, iwb-Forschungsberichte, Band 65, Technische Universität München, 1994.

[171] WOENCKKAUS, C.; KUGELMANN, D.: USIS - An Integrated 3D-Tool for Planning Production Cells, IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, München, September 1994.

[172] WÖRN, H.: Tendenzen in der Fabrikautomation und Robotik, atp - Automatiserungstechnische Praxis, Bd. 45, Nr. 7, S. 48–53, Juli 2003.

[173] WURLL, C.: "Pick To Pallet" - Automated Order Picking with Industrial Robots, Robotik 2002, Ludwigsburg, VDI-Berichte Nr. 1679, S. 167–172, 2002.

[174] ZABEL, A.: Werkstattorientierte Programmierung von Industrierobotern für automatisiertes Lichtbogenschweissen, Dissertation, Laboratorium für Werkzeugmaschinen und Betriebslehre (WZL), Rheinisch-Westfälische Technische Hochschule Aachen, 1993.

[175] ZHA, X. F.; DU, H.: Generation and Simulation of Robot Trajectories in a Virtual CAD-Based Off-Line Programming Environment, International Journal of Advanced Manufacturing Technology, Bd. 17, S. 610–624, 2001.

[176] ZHANG, Y.; MÜNCH, H.: Optimal Motion Programming for Industrial Robots, Robotics ’94 - Flexible Production - Flexible Automation. Proceedings of the 25th International Symposium on Industrial Robots, S. 701–707, Hannover, April 1994.

Lebenslauf

24. Februar 1970	geboren in Böblingen verheiratet, zwei Kinder
1976-1980	Grundschule Darmsheim
1977	Ravenscroft School, Raleigh, USA
1980-1989	Stiftsgymnasium Sindelfingen Abschluss: Abitur
1989-1990	Wehrdienst in Landsberg am Lech
1990-1995	Studium der Informatik an der Universität Karlsruhe (TH)
1994	Studienarbeit an der ENSAIS in Straßburg, Frankreich
1995	Diplomarbeit am Daimler-Benz-Forschungszentrum in Ulm Abschluss: Diplom-Informatiker
1995-1996	Mitarbeiter bei der Mercedes-Benz AG in Sindelfingen
1997-2004	Wissenschaftlicher Mitarbeiter am Institut für Roboterforschung in Dortmund
seit Oktober 2004	Mitarbeiter bei der DaimlerChrysler AG in Sindelfingen