Embedded Systems

I Tagungen und Berichte 1

Herausgegeben von Prof. Dr.-Ing. Birgit Vogel-Heuser,
Universität Kassel

Automation & Embedded Systems

Effizienzsteigerung im Engineering

Herausgegeben von
Prof. Dr.-Ing. Birgit Vogel-Heuser

kassel university press

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar

ISBN print: 978-3-89958-600-8
ISBN online: 978-3-89958-601-5
URN: urn:nbn:de:0002-6019

www.upress.uni-kassel.de

Druck und Verarbeitung: Unidruckerei der Universität Kassel
Printed in Germany

Inhaltsverzeichnis

1 Objektorientierung im Engineering der Automatisierungstechnik: Fluch oder Segen?

Prof. Dr.-Ing. Birgit Vogel-Heuser, Universität Kassel

Die Diskussion über Objektorientierung im Engineering der Automatisierungstechnik wird bereits seit 2001 geführt. Im Arbeitskreis Modellierung der FG Echtzeitsysteme der GI bzw. dem FA 5.12 der GMA wurden die Vorteile bzw. Nachteile der Objektorientierung bereits frühzeitig untersucht. Der Vergleich der verschiedenen objektorientierten Beschreibungsmittel (Tab. 1.2) anhand der VDI/VDE 3681 [1] Klassifizierung ist ein Ansatz um die Diskussion anhand von Kriterien zu bewerten. Die Auseinandersetzung lässt sich meines Erachtens auf drei Kernaspekte reduzieren:

1. Die Frage bei welchen Anforderungen aus dem Engineering bzw. Tätigkeiten im Engineering der Automatisierungstechnik wirkt sich die Objektorientierung aus und wirkt sie sich positiv oder negativ aus?
2. Welche Vor-/ bzw. Nachteile haben objektorientierter Programmiersprachen an sich?
3. Wie kommen Entwicklungs- und Applikationsingenieure der Automatisierungstechnik mit dem Paradigma der Objektorientierung klar? Die Killerthese hierzu lautet: „Applikationsingenieure der Automatisierungstechnik sind sowieso zu dumm um Objektorientierung zu verstehen und anzuwenden“.

Die erste Frage soll grundsätzlicher mit einer Detaillierung des morphologischen Kastens der Anforderungen an das Engineering im Maschinen- und Anlagenbau behandelt werden (Abb. 1.1). Die zweite Frage wird im Beitrag von Herrn Witsch detailliert behandelt. Die dritte Frage ist bereits im Rahmen von mehreren empirischen Evaluationen am Fachgebiet untersucht worden [2, 3, 4]. Ein wesentlicher Aspekt bei der Beurteilung ist die Werkzeugunterstützung und die Integration dieser Werkzeugunterstützung in die vorhandene Steuerungswelt [5]. Dieser Übersichtsbeitrag behandelt im Wesentlichen die erste Frage mit den Aspekten Prozess, Automatisierungsgeräte/Systemarchitektur bzw. Projekt.

Der morphologische Kasten (Abb. 1.1) unterscheidet in die Charakteristika des technischen Prozesses (diskret bzw. kontinuierliche). Der Batch-Prozess wird als Untergruppe des kontinuierlichen Prozesses mit einer unendlich langen Dauer eines Prozessschrittes betrachtet. Das zweite Kriterium unter der Rubrik Prozess ist die Domäne der Anwendung. Hierbei wird häufig nur zwischen Fertigungsautomatisierung und Prozessautomatisierung (Verfahrenstechnik) unterschieden. Die VDI/VDE 3687 [6] fügt die Gebäudeautomatisierung hinzu, ebenso wie die Ver- und Entsorgungstechnik und Umweltüberwachung, die für unsere Fragestellung wesentlich sind. Die Bezüge zum Kriterium Automatisierungsgeräte sind offensichtlich: in der Fertigungsautomatisierung

Abb. 1.1 Morphologischer Kasten „Anforderungen an Programmierumgebung“

Anforderungen an Programmierumgebung												
Prozess	Charakteristik des technisches Prozesses	diskret						kontinuierlich (Batch)				
	Domäne	Fertigungsautomatisierung (FT)		Prozessautomatisierung/ Verfahrenstechnik (VT)				Gebäudeautomatisierung (GT)		Ver- und Entsorgungstechnik, Umweltüberwachung		
	Sicherheitsanforderungen											
Automatisierungsgerät (heterogen)	Architektur der Steuerungstechnik	zentral		dezentral verteilt				dezentral intelligent				
	eingesetzte Steuerungstechnik (Systemtechnik)	FPGA										
		Mikrocontroller										
		IPC										
		CNC										
		SPS	Kleinsteuerung		Kleinsteuerung		mittlere Steuerung		große Steuerung			
		PLS										
SPS/PLS-Engineering-umgebung aus den Phasen des Engineering	Betrieb und Wartung (Multiuser)	Anforderungsermittlung (Grundlagenermittlung)		Nutzeranforderungen				Systemanforderungen				
		Basisplanung										
		Feinplanung/Konstruktion										
		Subsystemtest im Werk										
		Inbetriebsetzung		Methoden/Programmteile von Instanzen bilden								
		Abnahme		Sub-System Test				Systemtest				
		Monitoring der aktuellen Variablen										
		Programm-änderung online		Forcen von Variablen		„Brücken“		Sicherer Austausch von Programmteilen		Änderung von Variablenzuordnung im Programm		
	Re-Engineering: Erweiterung/Optimierung	Auslesen und Analysieren von bestehenden Programmen						Einspielen geänderter Programmteile				
Projekt	Art der Modularität/ Wieder-verwendung	Größe (Granularität) der Module		Basismodul				Applikationsmodul		Anlagenmodule		
		Art der Parametrier-barkeit der Module (Autonomie der Module)		Parametrierung der Variablen				Parametrierung der Verbindungen		Serviceorientierte Strukturen		
		Art der Wiederverwendung		Copy and Paste		Copy and Modify		Ableitung von Universalmodul (Modul enthält alle auftretenden Varianten); Varianten durch Parametrierung		Vererbung		
		Funktionalität der Module		FT: Module - modulare Maschine		VT: Module – Rezepte, Teilrezepte		FT/VT: Module - modularer Prozess				
	Personal-qualifikation	Engineering	Informatiker zur Modulerstellung	Hochsprachen								
			Regelungstechnische Aufgabe	Maschinenbauingenieur, Elektroingenieur, PLT - Ingenieur				CFC, ST, Hochsprache				
			Steuerungstechnische Aufgabe	Maschinenbauingenieur, Elektroingenieur, PLT - Ingenieur, Techniker				KOP, FUP, AWL, AS, ST				
		Betrieb und Wartung	FT/VT: Techniker, Facharbeiter	KOP, FUP, AWL, VT: CFC								
			GT: Hausmeister	vereinfachte Darstellung								

werden in der Regel als eingesetzte Steuerungstechnik Speicherprogrammierbare Steuerungen (SPS) und zunehmend auch Industrie Personal Computer (IPC), bzw. für Werkzeugmaschinen CNC eingesetzt, während in der Verfahrenstechnik Prozessleitsysteme (PLS) höhere Verbreitung finden. Darüber hinaus werden immer mehr eingebettete Systeme (FPGA, Mikrocontroller oder auch in Buskoppler integrierte IPCs) eingesetzt. Aber auch bei den PLS und SPS gibt es verschiedene Steuerungsfamilien, die für unterschiedliche anspruchsvolle Aufgaben (Anzahl der I/O, Schnelligkeit von Berechnungen etc.) eingesetzt werden. Am Beispiel der SPS soll dies weiter detailliert werden. Die angebotenen SPS'en reichen von Kleinststeuerungen, wie LOGO, die ihre Anwendung im Bereich des Handwerks finden, bis zu den großen Steuerungen der S7-400 Familie, um dies anhand des Marktführers in Siemens zu erläutern (Tab. 1.1). Bezogen auf unsere Frage der Anwendbarkeit der Objektorientierung in der Automatisierungstechnik stellt sich die Frage, ob diese für alle Steuerungsfamilien nutzbar sein soll oder nur für ausgewählte. Eine Analogiebetrachtung kann mittels der bisherigen fünf IEC 61131-3-Sprachen und des Continuous Function Chart (CFC) als zusätzlicher Sprache und deren Implementierung auf den verschiedenen Steuerungsfamilien durchgeführt werden. Sequentiell Function Chart (SFC oder auch Ablaufsprache genannt) und Strukturierter Text (ST) werden häufig als nicht geeignet für Techniker angesehen, insofern stellt sich die Frage welche Sprachen auf welcher Steuerungsfamilie verfügbar sind.
Die Übersicht (*Tab. 1.1*) zeigt zwei der Marktführer in den verschiedenen Kontinenten, Siemens und Rockwell, und anderseits zwei den Soft SPS'en zuzuordnenden Unternehmen (Phoenix Contact und Beckhoff), sowie Sabo, um auch Mikroprozessorbasierte Steuerungen einzubeziehen. Für die Kleinsteuerungen von Siemens (Logo), Rockwell (PicoControllers) sowie die Nanoline von Phoenix zeigt sich ein auf die Zielgruppe des Handwerks zugeschnittene (vereinfachte) Programmierumgebung, die ohne die klassischen IEC 61131-3 Sprachen bzw. nur mit einer erheblichen Einschränkung ausgeliefert wird. Demgegenüber stehen Phoenix Contact, Beckhoff sowie Sabo die den gesamten Umfang der IEC anbieten. Interessant ist auch die Einbeziehung der Hochsprache C (bzw. C^{++}), wenn die Frage nach den Programmierkenntnissen der Entwickler gestellt wird. C wird unter anderem von Phoenix Contact, aber auch von B&R für ausgewählte Steuerungen angeboten. In der Diskussion mit Herstellern wird als Grund häufig die Anforderung durch die Gruppe der Informatiker als Applikationsingenieure angegeben.

Zusammenfassend zeigt diese Zusammenstellung

- erstens, dass es offensichtlich nicht richtig ist anzunehmen, dass eine auf leistungsfähigeren Steuerungen angebotene Programmiersprache auch auf allen kleineren Steuerungen bereit gestellt werden muss,
- zweitens, dass bereits heute Hochsprachen (sogar objektorientierte) als Alternative auf vielen Steuerungen angeboten werden.
- drittens, dass es somit nicht richtig wäre eine objektorientierte Erweiterung der IEC 61131-3 nicht anzubieten bzw. zu implementieren, weil diese auf bestimmten Steuerungen für bestimmte Zielgruppe nicht angemessen erscheint

*Tab. 1.1: Von unterschiedlichen Steuerungsfamilien unterstützte IEC - Sprachen plus CFC (die Indizes verweisen auf die [7], * spezieller KOP)*

Siemens	Steuerungsfamilie	Logo![1]	S7 200[2]	S7 300[3]	S7 400[4]			
	Programmiersoftware	Logo! Soft Comfort V6.0	Step7 - Micro/WIN	Step7	Step7 / PCS7			
	Programmiersprachen IEC und höhere Programmiersprachen	LAD*, FBD	KOP, FUP, AWL	KOP, FUP, AWL	KOP, FUP, AWL			
				ab S7 31x PN/DP	SCL, CFC, GRAPH, HiGraph			
				SCL, CFC, GRAPH, HiGraph	AS (SFC), CFC			
Phoenix Contact[6]	Steuerungsfamilie	Nanoline[5]	ILC 150	ILC 250	ILC 350	IPC	PCI-Steuerungskarten	
	Programmiersoftware	NanoNavigator	PC WORX	PC WORX	PC WORX	PC WORX		
	Programmiersprachen IEC und höhere Programmiersprachen	Ablaufdiagramme	KOP, FUP, AWL	KOP, FUP, AWL	KOP, FUP, AWL	C, C++, VB	KOP, FUP, AWL	
			ST, AS	ST, AS	ST, AS		ST, AS	
			Maschinen Ablaufsprache	Maschinen Ablaufsprache	Maschinen Ablaufsprache		Maschinen Ablaufsprache	
			Fixed-Format-Ladder	Fixed-Format-Ladder	Fixed-Format-Ladder		Fixed-Format-Ladder	
			Steeplechase VLC	Steeplechase VLC	Steeplechase VLC		Steeplechase VLC	
							C, C++, VB	
B&R Automation[7]	Steuerungsfamilie	X20 Kompakt CPU	X20	System 2003	System 2005			
	Programmiersoftware	B&R Automation Studio	B&R Automation Studio	B&R Automation Studio	B&R Automation Studio			
	Programmiersprachen IEC und höhere Programmiersprachen	KOP, FUP, AWL, ST, AS	KOP, FUP, AWL, ST, AS		KOP, FUP, AWL, ST, AS			
		Unterstützung von C	Unterstützung von C		Unterstützung von C			
Beckhoff[8]	Steuerungsfamilie	BCxx0	BCxx50	BCxx20	BXxx00	CX		
	Programmiersoftware	TwinCAT	TwinCAT	TwinCAT	TwinCAT	TwinCAT		
	Programmiersprachen IEC und höhere Programmiersprachen	IEC 61131-3 + CFC	IEC 61131-3 + CFC	IEC 61131-3 + CFC	IEC 61131-3 + CFC	IEC 61131-3 + CFC		
						Anwendertasks in Hochsprachen unter Windows OE		
Rockwell	Steuerungsfamilie	Pico Controlers[10]	MicroLogix 1x00 System[11]	CompactLogix System[12]	ControlLogix System[12]	FlexLogic System[12]	SoftLogix Controller[12]	
	Programmiersoftware	PicoSoft 6	RS Logix 5000	RS Logix 5000	RS Logix 5000	RS Logix 5000	RS Logix 5000	
			RS MicroLogix					
	Programmiersprachen IEC und höhere Programmiersprachen	KOP	KOP, FUP, ST, SFC	KOP, FUP, ST, SFC	KOP, FUP, ST, SFC	KOP, FUP, ST, SFC	KOP, FUP, ST, SFC	
Sabo Elektronik	Steuerungsfamilie	Mikroprozessormodul[13]	Basismodul[14]	Modulare Systemfamilie[15]	Master-Terminal-Baugruppe[16]	Master-Terminal-Baugruppe (9" Touchpanel) 17		
	Programmiersoftware	MPM.730.20 D1	PLM230	PLM 500	MTB.711.10 D1	MTB 747.13 D[1]		
	Programmiersprachen IEC und höhere Programmiersprachen	CoDeSys (2.8)						

Als zweites Kriterium bezüglich der Automatisierungsgeräte ist der Grad der Verteilung der Anwendung aufgeführt, dies ist für die Modellierung und für das Engineering eine weitere anspruchsvolle Anforderung, insbesondere bezüglich der Variablenzuordnung über Steuerungen hinweg.

Das dritte Kriterium mit dem Titel Projekt im morphologischen Kasten beinhaltet drei Kriteriengruppen: den Lebenszyklus des Engineering, die Art der Modularität und Wiederverwendung sowie die Personalqualifikation. Diese Kriteriengruppen haben durchaus unterschiedliche Ausprägungen für die verschiedenen Domänen (Kategorie Prozess), beispielsweise bezüglich der zulässigen Änderungen zur Laufzeit, der Funktionalität zur Laufzeit und des Personals für Engineering, Betrieb und Wartung. Zunächst soll im Folgenden der Lebenszyklus (SPS/PLS-Engineering-Umgebung aus den Phasen des Engineering) diskutiert werden. Die Engineeringphasen lehnen sich dabei an das NAMUR - Vorgehensmodell [8] an, sind aber um für die Softwareentwicklung wesentliche Elemente detailliert, um die wesentlichen Forderungen an die Änderungsmöglichkeiten während der Inbetriebnahme, Betrieb und Wartung sowie Re-Engineering einfügen zu können.

Im Bereich des Maschinen- und Anlagenbaus gewinnt der Subsystemtest im Werk immer mehr an Bedeutung, um Inbetriebnahmezeiten auf der Baustelle zu reduzieren. Die Inbetriebsetzung besteht aus Ein-/Ausgangsprüfung sowie der eigentlichen Inbetriebnahme der Subsysteme bis zum Gesamtsystemtest, der durch die Abnahme abgeschlossen wird. Als wesentliche Funktionalität während der Inbetriebsetzung wird das Bilden von Instanzen von Methoden bzw. Programmteilen aus Sicht des Maschinenbaus angestrebt (bei objektorientierter Programmierung), wenn nicht der Konstrukteur selber die Inbetriebnahme durchführt, weil es sich um einen Prototypen einer Anlage oder Teilanlage handelt. Für die Betriebs- bzw. Wartungsphase sind folgende Fälle zu berücksichtigen: das Monitoring von aktuellen Variablen und die Onlinemanipulation des Programms, wie das Forcen von Variablen, das Brücken oder auch komplexere Änderungen, wie der sichere Austausch von Programmteilen bzw. das sichere Ändern von Variablenzuordnungen im Programm. Wesentlich ist dabei auch der Multiuser-Modus, d.h. mehrere Anwendungsentwickler /Inbetriebnehmer können gleichzeitig ihre Programmteile auf ein und derselben Steuerung bearbeiten. An der Erfüllung dieser Kriterien ist ein Objektorientierter Engineeringansatz genauso zu messen wie ein Komponenten- oder Funktionsblock-orientierter Ansatz. Die speziellen Anforderungen der Prozessleittechnik werden hier nicht weiter betrachtet. Für das Re-Engineering bzw. die Optimierung der Anlage und damit auch der Software sind insbesondere das Auslesen und das Analysieren von bestehenden Programmen wesentlich. Dies wird bisher nur unzureichend unterstützt, da SPS-Programme häufig unübersichtlich sind. Häufig müssen beim Re-Engineering geänderte Programmteile auch während des Betriebs eingespielt werden. Dabei sind geeignete Umschaltpunkte zu finden bzw. geeignete Übergangsstrategien zu finden, wie für den Zeitpunkt des Austausches die Ausgänge stabil auf dem alten Wert zu halten.

Gerade aus den Fähigkeiten der Online-Beobachtung und der Online-Änderungen bezieht die SPS ihr Alleinstellungsmerkmal im Vergleich zu Prozessrechner oder IPCs mit Hochsprachenprogrammen. Durch Objektorientierte Ansätze darf dieser Vorteil nicht zerstört werden.

Eine zweite Kriteriengruppe, die Modularität und Wiederverwendung, ist der Ursprungspunkt für die Diskussion über Objektorientierung und deren Einführung in der Automatisierungstechnik. Um die Effizienz im Engineering der Automatisierungstechnik für den Maschinen –und Anlagenbau zu erreichen, sollen die objektorientierte Ansätze der Informatik evaluiert werden. In der Automatisierungstechnik ist es bisher nur unzureichend gelungen [9] Module zu definieren und Wiederverwendung in der Breite zu erreichen. Wesentlicher Aspekt ist sicherlich auch die Interdisziplinarität der Module bzw. die Tatsache, dass Module in der Automatisierungstechnik dazu dienen den technischen Prozess zu ermöglichen bzw. eine notwendige Funktion des technischen Prozesses zu realisieren und dabei zu mindestens aus den Sichten technischer Prozess, technisches System und Automatisierungssystem [10] bestehen und somit verschiedene Disziplinen bzw. Gewerke berücksichtigen müssen. Dieser Aspekt ist im morphologischen Kasten nicht separat ausgeführt. In den verschiedenen Domänen werden Module unterschiedlicher Funktionalität als besonders geeignet betrachtet. In der Prozessautomatisierung sind Rezepte und Teilrezepte Module, in der Fertigungsautomatisierung wird hingegen auf modulare Maschinenfunktionen gezielt. Insgesamt wäre ein Ziel einen modularen Prozess zu erreichen, der aus konfigurierbaren, wieder verwendbaren Modulen besteht. Auch hinsichtlich der Größe bzw. Granularität der Module haben wir [9] große Unterschiede gefunden. Basismodule (im Sinne von Reglermodul oder Ventilmodul) finden sich fast in allen Unternehmen, komplexere Module oder applikationsspezifische Module sind eher selten realisiert. Um größere Module einzuführen ist es notwendig die Art der Wiederverwendung zu betrachten und die Verbindung von Modulen untereinander zu lösen. Bei der Art der Wiederverwendung haben wir aus verschiedenen Industrieprojekten folgende eingesetzte Mechanismen abgeleitet: Copy and Paste als das am weitesten verbreitete Prinzip, Copy and Modify bzw. als nächste Stufe die Erstellung eines möglichst alle Anforderungen erfüllenden Universalmoduls, welches durch Parametrierung auf den vorliegenden Anwendungsfall zugeschnitten wird und als bisher nicht beobachtete Variante die Ableitung von Modulen durch Vererbung. Insbesondere bei der Vererbung ist der Zusammenhang zur Objektorientierung offensichtlich, aber auch zur Ableitung und Dokumentation von Universalmodulen können objektorientierte Beschreibungen und insbesondere auch die UML erfolgreich eingesetzt werden. Bezüglich der Parametrierung der Module ist einerseits die Parametrierung der Module selber über Variablen zu unterscheiden und andererseits die Parametrierung der Modulverschaltung. Bei Serviceorientierten Strukturen, die höhere Autonomie aufweisen, ist diese Verschaltung sicherlich einfacher zu realisieren.

Zusammenfassend kann zum Punkt Art der Modularität/Wiederverwendung festgehalten werden, dass eine genauere Untersuchung der in der Praxis erfolgreichen Strategien notwendig wäre. Allerdings ist dies kaum möglich, da aufgrund fehlender Unterstützungswerkzeuge die Anwendungsfälle nicht zahlreich sind und die Wiederverwendung in der Regel auf ein führendes Gewerk beschränkt ist. Die mit der Modularität und dem Modulmanagement verknüpfte Frage des Varianten –und Versionsmanagements ist bisher noch unzureichend hinsichtlich der Anforderungen erfasst.

Der abschließende Punkt, die Berücksichtigung der Qualifikation der Enringeering- bzw. Betrieb-Wartungs-Mitarbeiter, ist zentral für den Erfolg jeglichen Engineering-Ansatzes. In den letzten Jahren lässt sich ein Trend im Detail-Engineering erkennen: die Aufteilung in Modulersteller und Modulanwender, wenn Modularität eingeführt wird. Zur Modulerstellung werden häufig Informatiker eingesetzt, die in der Regel Hochsprachen zur Programmierung bevorzugen. Für die Regelungstechnischen Aufgabenstellungen bzw. Steuerungstechnischen Aufgabenstellungen werden dahingegen eher Maschinenbauingenieure oder Elektroingenieure eingesetzt bzw. in der Verfahrenstechnik PLT-Ingenieure mit den angegebenen besonders geeigneten Sprachen. Bei anspruchsvollen regelungstechnischen Aufgabenstellungen bietet sich insbesondere die Modellierung in Matlab/Simulink an. Für Matlab/Simulink existierten bisher vor allem Codegeneratoren in die Programmiersprache C. Bayrak et al haben nunmehr auch eine erfolgreiche Codetransformation in die CFC und SFC der IEC realisiert [11].

Für Betrieb und Wartung ist sicherlich ein erheblicher Unterschied der Personalqualifikation bei weltweiter Betrachtung und in den verschiedenen Domänen. Der morphologische Kasten fokussiert zunächst auf den deutschen bzw. europäischen Bereich. In der Fertigungsautomatisierung und in der Prozessautomatisierung stehen in der Phase der Betriebsbetreuung und Wartung in der Regel Facharbeiter oder Techniker zur Verfügung, die auf jeden Fall drei der SPS-Sprachen beherrschen. Diese Mitarbeiter sind in der Lage, die unter dem Punkt Betrieb und Wartung (SPS/PLS-Engineering) aufgeführten Aufgaben durchzuführen mit der Ausnahme des sicheren Austauschs von Programmteilen bzw. der Änderung von Variablenzuordnungen im Programm. Dies ist in der Regel dem Servicepersonal des Anlagenlieferenten oder des Engineeringunternehmens vorbehalten.

Tab. 1.2: Einteilung der Modellierungssprachen durch ihre Merkmale (in Anlehnung an VDI/VDE 3681)

Kriterium	Alternativen	Funktionsblöcke nach IEC 61499	UML 2.0	SysML						UML CoDeSys 3.0			UML-PA			
				Requirement Diagramm	Block Definitions Diagramm	Internes Block Diagramm	Parameter Diagramm	Zustandsdiagramm	Aktivitätsdiagramm	Klassendiagramm	state chart	Aktivitätsdiagramm	Klassendiagramm	Instanzenstruktur-diagramm	Timing Sequenzdiagramm	Zustandsdiagramm
Formale Basis	Formal															
	Semi-formal	x	x		x	x		x	x	x	x	x	x		x	x
	Informal			x			x						x	x		x
Verhaltensbeschreibung	Deterministisch	x									x	x			x	x
	Nicht deterministisch		x					x	x						x	x
	Statisch	x	x					x	x						x	x
	Dynamisch	x	x					x	x		x	x			x	x
Explizite Zeitdarstellung	Ereignisgetrieben diskret	x	x					x			x	x			x	x
	Zeitdiskret (taktgetrieben)										x	x			x	x
	Zeitkontinuierlich		x												x	x
Struktur	Hierarchie	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Typen / Instanzen	x	x	x	x	x	x			x		x	x	x	x	x
	Komposition / Dekomposition	x	x	x	x	x	x			x	x		x	x	x	x
	Strukturveränderung		x											x	x	x
Synchronisation (falls verteilte Prozesse möglich)	Synchron										x	x				
	Asynchron	x	x					x	x						x	x
	Nebenläufig	x	x					x	x						x	x
Darstellung	Textuell	x	x	x						x	x	x	x	x	x	x
	Mathematisch-symbolisch															
	Grafisch	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Tools	Forschung		x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Prototypische Anwendung	x								x	x	x	x	x	x	x
	Industrielle Anwendung		x	x	x	x	x	x	x							
Architekturbeschreibung	*Hardwarekomponenten*	x	x		x	x							x	x		
	Kommunikationswege	x	x		x	x							x	x		
	Kommunikationsprotokolle		x					x	x				x	x	x	x
	Mapping	x	x		x	x		x	x				x	x	x	x

Um die Effizienz im Engineering zu erhöhen sind also für die

- verschiedenen Prozesse und Domänen
- angemessene Modellierungs- bzw. Programmiersprachen, -werkzeuge,
- Arten der Modularisierung und Wiederverwendung,
- für die oben ausgearbeiteten Online-Monitoring und –manimpulationsmöglichkeiten
- für die verschiedenen Mitarbeitergruppen

zur Verfügung zu stellen. Von besonderer Wichtigkeit ist dabei, dass nicht nur das Detail-Engineering unterstützt werden muss, sondern auch die Inbetriebnahme und der Betrieb und die Wartung in dieses Konzept mit einbezogen werden müssen. Die Hauptkritik an der Verwendung von UML-Werkzeugen ohne die Integration in den Engineeringworkflow und die Einbettung in SPS-Programmierumgebungen richtet sich auf die Phase der Inbetriebnahme und des Betriebs und der Wartung. Da während der Inbetriebnahme immer wieder Optimierungen im Programm notwendig sind, insbesondere bei Neuentwicklungen bzw. Protoypen ist es wünschenswert bzw. erforderlich während der Inbetriebnahme eine Verbindung zum UML-Modell zu gewährleisten, damit sich Code und Modell nicht auseinanderentwickeln. Dies ist durch eine Werkzeugintegration von UML in eine IEC 61131-3-Umgebung gewährleistet, wie sie im Rahmen dieses Tagungsbandes vorgestellten Projektes protoypisch entwickelt wurde.

Insbesondere für das Re-Engineering bei der Analyse existierenden Programmcodes bietet die UML mit ihren Klassendiagrammen eine gute Möglichkeit der Dokumentation der Programmstruktur.

Der Beitrag Einsatz von UML-Diagrammen in der Steuerungsprogrammierung-Diagramme und ihre Nutzer (D. Witsch-) erläutert die Besonderheiten, die bei der Online-Änderung von Programmen im Falle der Objektorientierten Erweiterung der IEC 61131-3, wie sie bei CoDeSys V3 vorhanden ist, zu beachten sind bzw. die erweiterten Möglichkeiten, die sinnvoll möglich sind.

Grundsätzlich stellt sich die Frage, warum wir für diesen Ansatz die UML ausgewählt haben und welche Diagramme wir aus dem Gesamtumfang der in der UML 2.0 festgelegten Diagramme ausgewählt haben. Alternativen wären auch die IEC 61499 bzw. die SysML gewesen. Der Vergleich (Tab.2) wird anhand der Kriterien zur Bewertung der Steuerungsprogrammierung nach VDI/VDE 3681 [1] durchgeführt. Diese Kriterien wurden um Aspekte der Architekturbeschreibung erweitert (die kursiv geschriebenen Kriterien sind Erweiterung der VDI/VDE 3681). In der Automatisierungstechnik ist ebenso wie bei den verteilten Eingebetteten Systemen im Bereich Automotive die Architekturbeschreibung wesentlich für die Auslegung des Gesamtsystems aufgrund der vom technischen Prozess geforderten Reaktionszeiten. In diesem Fall sind als Detailkriterien die Hardwarekomponenten, Kommunikationswege, Kommunikationsprotokolle und das

Mapping eingeführt. Die Kommunikationswege werden beispielsweise in SA/RT im Architketurflussdiagramm festgehalten.

Die Bewertung der SysML erfolgt entsprechend den Arbeiten am Fachgebiet [10] zur Modellierung des hybriden Labormodells. Außerdem ist eine separate Bewertung der UML-PA eines UML-Derivates für die Prozessautomatisierung, welches am Fachgebiet entwickelt wurde [2]. Für die UML, die in CoDeSys V3 integriert wurde, wurden das Klassendiagramm, state charts und das Aktivitätsdiagramm im Rahmen des Projektes von den Unternehmen priorisiert und realisiert. Das Aktivitätsdiagramm wurde insbesondere von den Technologen als geeignet angesehen, um in den frühen Phasen des Engineering die geforderte Funktion aus ihrer Sicht grob zu spezifizieren. Das Timing Diagramm, welches im Rahmen vorheriger Untersuchungen (Bartels, Vogel at und Katzke Diss) von besonderem Interesse für die Spezifikation von Zeitverhalten angesehen wurde, ist bisher nicht realisiert worden, weil es im vorliegenden Projekt insbesondere für die Spezifikation des Tasking (Sheduling) interessant ist, dies aber die Abbildung der Anbindung an das Betriebssystem notwendig wäre. Dies hätte Systemspezifisch realisiert werden müssen und wurde deshalb nach hinten verschoben.

Die Ergebnisse der Untersuchungen zur Usability der UML, also der Anwendbarkeit und ihres Nutzens in der Steuerungsentwicklung und –programmierung von Friedrich und Katzke [2] zeigen, dass eine Vorgehensweise für die Ingenieure notwendig ist ebenso wie eine Reduzierung der Diagrammarten und Erweiterung für Automatisierungstypische Aufgabenstellungen, um nur einige zu nennen. Katzke konnte zeigen, dass die so veränderte UML-PA der klassischen UML überlegen ist: „Eine definierte Menge zusammenhängender Informationen kann in der aus einem Diagrammtypen bestehenden Instanzendarstellung der UML-PA in kürzerer Zeit erkannt und wiedergegeben werden, als in den über verschiedene Diagrammtypen verteilten Instanzendarstellungen der UML“ ([2], 2009, S. 100ff). Weitere Untersuchungen sollen insbesondere mit Techniker durchgeführt werden und den vorliegenden UML-Protoypen, um den Nutzen der Entwicklung mit einem realen Werkzeug zu zeigen, der sicherlich deutlich höher ausfallen wird als die bisherigen Untersuchungen ohne Ankopplung an die IEC 61131-3.

Zusammenfassend kann festgehalten werden, dass für die Domäne der Fertigungsautomatisierung und die Domäne der hybriden Prozesse im Maschinen- und Anlagenbau die UML nicht nur für das Detailengineering, sondern auch für die Analyse von Fehlern im Betrieb und Wartung hilfreich ist, weil sie gerade in diesen Phasen die schnelle Fehlersuche unterstützt und bei Rückfragen mit der Konstruktion bezüglich eventueller Änderungen die Kommunikation erheblich erleichtert. Bezüglich des Einsatzes in der klassischen Verfahrenstechnik mit Anlagen, die über fünf Jahre nicht angehalten werden und sehr kritischen Prozessen ist sicherlich eine geeignete Unterstützung für den Betrieb und Wartung zu entwerfen, um sichere Änderungen zur Laufzeit zu erlauben. Erste Ansätze für eine Unterstützung solcher Änderungen werden im Beitrag Einsatz von UML-Diagrammen in der Steuerungsprogrammierung- Diagramme und ihre Nutzer vorgestellt.

1.1 Referenzen

[1] VDI Richtlinie: VDI/VDE 3681: Einordnung und Bewertung von Beschreibungsmitteln aus der Automatisierungstechnik. Herausg. vom VDI/VDE, Ausgabe Oktober 2005.

[2] Katzke, U.: Dissertation: Spezifikation und Anwendung einer Modellierungssprache für die Automatisierungstechnik auf Basis der Unified Modeling Language (UML). 2009.

[3] Vogel-Heuser, B.; Friedrich, D.; Katzke, U.; Witsch, D.: Usability and benefits of UML for plant automation – some research results. atp international 3 (2005) No. 1, S. 52-60.

[4] Vogel-Heuser B., Friedrich, D.: Nutzen von Modellierung für die Qualität und Effizienz der Steuerungsprogrammierung in der Automatisierungstechnik. atp 48 (2006), Heft 3, S. 54-60.

[5] Bartels, J.; Vogel, B.: Systementwicklung für die Automatisierung im Anlagenbau. at Automatisierungstechnik 49 (2001), Heft 5, S. 214-224.

[6] VDI Richtlinie: VDI/VDE 3687: Auswahl von Feldbussystemen durch Bewertung ihrer Leistungseigenschaften für industrielle Anwendungsbereiche, Herausg. vom VDI/VDE, Ausgabe November 1999.

[7] Referenz zu den Indizes der *Tab. 1.1*

[1] Logo! Soft Comfort V6.0 Onlinehilfe

[2] Automatisierungssystem S7 200 Systemhandbuch 09/2007

[3] CPU 31xC und CPU 31x Technische Daten Gerätehandbuch 06/2008

[4] Automatisierungssystem S7 400 CPU Daten Gerätehandbuch 09/2008

[5] NanoNavigator Anwenderhandbuch 04/2008

[6] Phoenixcontact Produktkataloge --> Komponenten und Systeme AUTOMATION

[7] http://www.br-automation.com/cps/rde/xchg/br-productcatalogue/hs.xsl/products_5421_DEU_HTML.htm?session_level_1=Software

[8] http://www.beckhoff.de

[9] http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp

[10] http://www.ab.com/programmablecontrol/plc/pico/picosoft.html

[11] http://www.ab.com/en/epub/catalogs/12762/2181376/2416247/9071972/9072282/#

[12] http://www.ab.com/en/epub/catalogs/12762/2181376/2416247/407648/6238138/tab2.html

[13] http://www.sabo.de/deutsch/plm700/plm700_06.html

[14] http://www.sabo.de/deutsch/plm200/plm200_06.html

[15] http://www.sabo.de/pdf/plm500/Produktkatalog_2008_03_PLM_500_op_web.pdf

[16] http://www.sabo.de/deutsch/plm700/plm700_01.html

[17] http://www.sabo.de/deutsch/plm700/plm700_02.html

[8] NAMUR Richtlinie: Na 35: Abwicklung von PLT-Projekten. Ausgabe 2003.

[9] Katzke, U.; Vogel-Heuser, B.; Fischer, K.: Analysis and state of the art of modules in industrial automation. atp international 46 (2004) No. 1, S. 23-31.

[10] Schütz, D.; Wannagat, A.: Domänenspezifische Modellierung für automatisierungstechnische Anlagen mit Hilfe der SysML. atp (2009) Heft 3.

[11] Bayrak, G.; Abrishamchian, F.; Vogel-Heuser, B.: Effiziente Steuerungsprogrammierung durch automatische Modelltransformation von - Matlab/Simulink/Stateflow nach IEC 61131-3. atp 50 (2008) Heft 1

2 Objektorientierung in der Anlagenentwicklung – eine Vision

Ulf Schünemann, Ph.D. (MUN, Kanada)

2.1 Einleitung

Der Maschinen- und Anlagenbau wird herausgefordert durch mehrere gleichzeitige Entwicklungen: Die zu erfüllenden Aufgaben werden immer komplexer. Die Anzahl der Varianten nimmt immer weiter zu, bis keine Maschine mehr genau gleich einer anderen ist. Die Entwicklungsyzklen verkürzen sich. Immer mehr Aufgaben werden auf die Steuerungssoftware verlagert, die damit immer kritischer für den Erfolg wird.

Das gleiche Problem stellte sich der SW-Industrie in den 80er Jahren, als immer komplexere Büroaufgaben und Geschäftsabläufe aus immer mehr Branchen auf PC-Software abgebildet werden sollten. Die nahe liegende Idee der **Modularisierung** innerhalb des alten prozeduralen Programmieransatzes **scheiterte** hier regelmäßig, da sie immer zu starr und zu programmierbezogen war. Erst ein grundsätzlich anderer Ansatz, die **Objektorientierung**, führte zur **Verbesserung** und setzte sich in der Folge bei der PC-Softwareentwicklung durch.

Objektorientierung ist mehr als objektorientiertes Programmieren. Sie ist eine Sammlung von Konzepten bzw. Beschreibungstechniken mit dem Anspruch,

- allgemein verständlich zu sein (nicht nur für Informatiker) und daher für die fachübergreifende Kommunikation zu taugen;
- allgemein anwendbar zu sein zur funktionalen Beschreibung von Systemen (nicht nur Softwaresystemen) und daher Software und ihren Anwendungskontext in einer Beschreibung integrieren zu können;
- in jeder Entwicklungsphase mit den gleichen Konzepten zu arbeiten (kein konzeptioneller Bruch zwischen der Beschreibung der Systemanforderungen, der Architektur, des Systemdesigns und der konkreten Implementierung), so dass sich die Zusammenhänge zwischen den Phasen, vor allem bei Änderungen, leichter nachverfolgen lassen;
- mehrere Abstraktionsebenen in einer Systembeschreibung zu integrieren (statt mehrere Beschreibungen des gleichen Systems auf verschiedenen Ebenen anzufertigen und zu pflegen);
- alle Formen der Wiederverwendung zu unterstützen,
  - nicht nur die Wiederverwendung von Komponenten aus einem Baukasten zur Definition neuer Systeme,
  - sondern auch die Wiederverwendung von Systemen unter Austausch ihrer Komponenten, sowie

- die Anpassung existierender Komponenten durch Ableitung, ohne dass dieses bereits bei der ursprünglichen Definition (in Form von Parameterflags o.ä.) vorgesehen werden musste.

Von jeder dieser Eigenschaften kann auch der Maschinen- und Anlagenbau bei seinen Herausforderungen profitieren. Da hier mehrere Fachgebiete (Softwaretechnik, Steuerungstechnik, Mechanik des Maschinenbaus, Verfahrenstechnik) im gleichen Entwicklungsprojekt zusammenarbeiten, entfaltet sich das volle Potential nur, wenn die Objektorientierung fachübergreifend zur funktionalen Systembeschreibung verwendet wird (und nicht nur in der Programmierung). Daher werden die erarbeiteten Systembeschreibungen (Systemmodelle) nicht in einer objektorientierten Programmiersprache festgehalten, sondern in einer grafischen objektorientierten Modellierungssprache.

Der Standard dafür ist die **UML** (Unified Modeling Language) [1]. Im Forschungsprojekt der Universität Kassel werden UML-Systembeschreibungen ausführbar gemacht innerhalb eines CoDeSys-Projekts [3]. Dabei wird aufgebaut auf den objektorientierte Konzepten, welche CoDeSys 3 für die objektorientierte Programmierung als Erweiterung der IEC 61131-3 eingeführt hat (wie ich an gleicher Stelle vor einem Jahr dargestellt habe [4]).

Dieser Artikel skizziert anhand des Beispiels der Stempelanlage aus [3], in welchen Schritten die funktionale Systembeschreibung und das Steuerungsprogramm einer neuen Anlage entwickelt werden könnte: Doppelte Arbeit durch Übersetzen fachspezifischer Systembeschreibungen in die SPS-Programmierung wird überflüssig; mögliche zukünftige Wiederverwendung wird vorbereitet; Komponenten aus einem Baukasten funktionaler Komponenten werden wiederverwendet. Der Fokus liegt auf den Systembeschreibungen des Verfahrenstechnikers und Maschinenbauers. Zur eigentlichen Programmierung wird nicht mehr viel zu sagen sein.

2.2 Zentrale Konzepte der Objektorientierung

Die zentralen Konzepte der Objektorientierung [4] sollen hier kurz wiederholt werden.

Objekt = Systemkomponente = Daten + Operationen

Klasse = Objekttyp = Datendefinitionen + Operationsdefinitionen

Unterklasse = Spezialfall der Oberklasse = Erben + Differenz

Abb. 2.1: Zentrale Konzepte der Objektorientierung

Erstens, ein System wird betrachtet als hierarchisch aufgebaut aus Komponenten, genannt „**Objekte**“. Jedes dieser „Objekte“ kann seine eigenen Daten und seine eigenen Operationen haben. Unter den Begriff „Daten“ fallen unter anderem: Unterobjekte, Eigenschaften, Zustände, Links mit anderen Objekten. Unter „Operationen“ fallen sowohl

„schnelle“ Operationen, die aktiviert werden, um sofort ein Ergebnis zu bekommen, als auch „langsame“ Operationen, deren Aktivierung eine länger andauernde Aktivität startet bzw. fortsetzt.

Zweitens, ein Typ gleichartiger Objekte wird „**Klasse**“ genannt und die Definition der gemeinsamen Daten und Operationen aller Klassenmitglieder erfolgt bei dieser Klasse. Neue Objekte werden erzeugt, indem ihre Klasse „instanziiert“ wird.

Drittens, ein Klasse, genannt „**Unterklasse**“, kann als Spezialfall einer anderen Klasse, der Oberklasse, modelliert werden. Dann kann ein Objekt einer Unterklasse überall anstelle eines Objekts der Oberklasse eingesetzt werden (Polymorphie). Die Definition der Daten und Operation der Oberklasse gelten auch für die Unterklasse. Man sagt, die Unterklasse „erbt“ diese, und kann sie erweitern oder überschrieben.

2.3 Idealisierter Entwicklungsprozess

Die folgende Beschreibung einer Anlagenentwicklung ist stark skizzenhaft und idealisiert. Es soll nicht behauptet werden, dass das Vorgehen so wie beschrieben tatsächlich praktikabel ist. Aber es soll veranschaulichen und Anregungen gegeben, wie Systeme objektorientiert beschrieben werden könnten und wie dabei Wiederverwendung und Wiederverwendbarkeit berücksichtigt werden könnten.

Das Beispiel spielt nur auf der obersten strukturellen Ebene der Anlage, so dass das Problem der fachspezifisch unterschiedlichen Aufgliederung von Maschinenkomponenten [5] vernachlässigbar sein sollte. Um das Beispiel übersichtlich zu halten, wird auf einem Baukasten von mechatronischen Modulen [2] aufgebaut, deren Abbilder als Software-Komponenten in eine CoDeSys Bibliothek zusammengefasst sind.

2.3.1 Die allgemeine Systemstruktur (wiederverwendbar)

Der funktionale Grundaufbau der neuen Anlage kann von Maschinenbauer und Verfahrenstechniker gemeinsam festgelegt werden. Mit geringem Lernaufwand, sollte es ihnen möglich sein, den Strukturbaum [2] der neuen Anlage in Form eines UML-Klassendiagramm mit Kompositionsbeziehungen festzuhalten. Als Beispiel dient uns die 4-teilige Stempelanlage aus [3] mit Werkstückmagazin (Stapel), Kran, Bearbeitungseinheit (Stempel) und Sortierband (siehe Abb. 2.2).

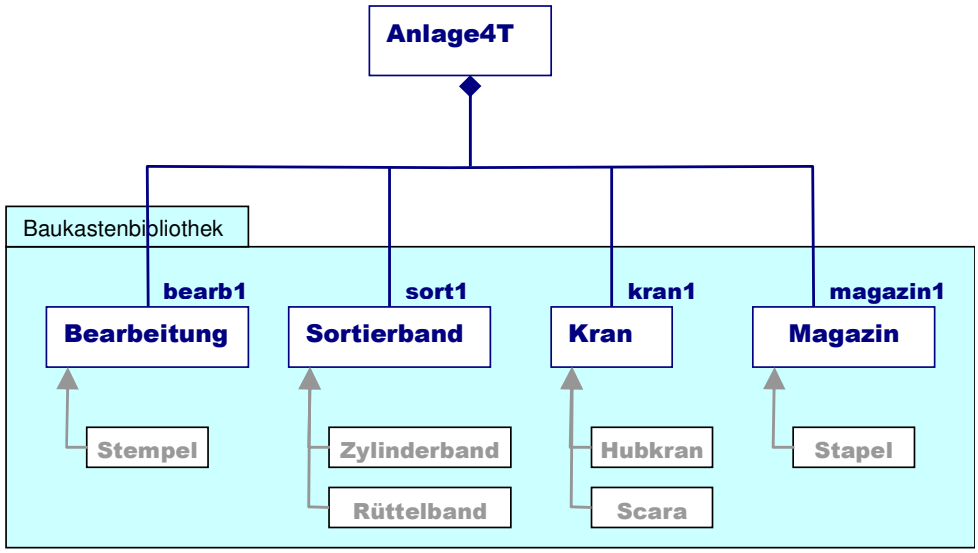


Abb. 2.2: Allgemeine Anlagenstruktur als UML-Klassendiagramm

Wiederverwendung: Wenn mit einem Baukasten von fertig definierten Aggregaten gearbeitet wird, müssen folgende Schritte durchgeführt werden:

Ein UML-Klassendiagramm wird angelegt und über die Toolbox des Diagramms wird die Baukastenbibliothek eingefügt.

Aus der Toolbox wird eine neue Klasse in das Diagramm gezogen und Anlage4T genannt.

Über die Toolbox werden Kompositionsbeziehungen eingefügt, die von der neuen Anlage zu ihren Aggregaten in der Bibliothek gehen (Bearbeitung, etc.). Den Aggregaten können eindeutige Namen im Kontext der neuen Anlage gegeben werden („bearb1“, etc.).

Wiederverwendbarkeit: In diesem Schritt wird die Beschreibung bewusst allgemein gehalten, um in zukünftigen Entwicklungen weiterer Anlagenvarianten eine Wiederverwendung dieses Grundaufbaus und aller darauf definierten Abläufe und Zustandsautomaten zu ermöglichen (nach dem Master-Konzept statt Copy&Modify [2]): Es wird noch nicht festgelegt sein, welches Magazin, welcher Kran, welche Bearbeitungseinheit und welches Sortierband zum Einsatz kommt.

Die Baukastenbibliothek definiert den Satz von Operationen und Eigenschaften, welchen die Aggregate allgemein besitzen, und welche daher in der Definition von allgemeinen Anlage4T-Abläufen und -Zustandsautomaten auftauchen können.

2.3.2 Die allgemeinen Betriebszustände (wiederverwendbar)

Alle Fachgebiete können zusammenkommen und klären, welche Betriebszustände die neue Anlage hat, und wie es zu Wechsel zwischen diesen kommt. So könnte man Initialisierungsphase, Betrieb und Warten auf Magazinnachfüllung unterscheiden. Das Ergebnis kann als UML-Zustandsmaschine festgehalten werden.

```
ACTIVITY Main
VAR
END_VAR
```

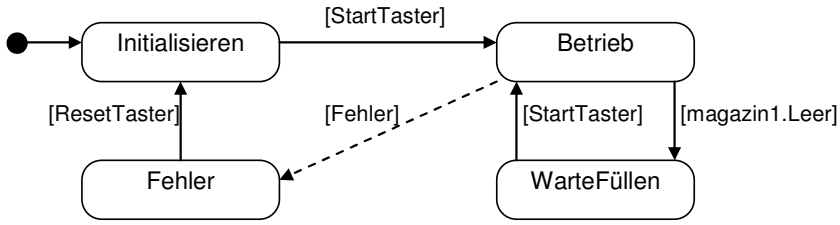


Abb. 2.3: Allgemeine Betriebszustände als UML-Zustandsautomat

Dazu wird aus der Toolbox des Klassendiagramms (Abb. 2.2) ein neuer Zustandsautomat, d.h. eine neue Operation mit UML-StateChart als Sprache, in die Klasse Anlage4T gezogen und „Main“ genannt. Der Editor für „Main“ zeigt ein Aktivitätsdiagramm (Abb. 2.3). Die gewünschten Zustände werden angelegt und miteinander verbunden. Die Transitionsbedingungen werden im Diagramm definiert. Durch Verwendung von „ResetTaster“, „StartTaster“ und „Fehler“ als Bedingung werden diese automatisch als Eigenschaften von Anlage4T definiert. Bei der Bedingung „magazin1.Leer“ wird auf die Eigenschaft „Leer“ der Klasse „Magazin“ aus der Baukastenbibliothek zurückgegriffen.

Wiederverwendbarkeit: Der definierte Zustandsautomat ist wiederverwendbar für alle zukünftigen konkreten Ausprägungen von Anlage4T (siehe 2.3.4). Die in den Bedingungen verwendeten Eigenschaften müssen in den konkreten Ausprägungen definiert sein.

2.3.3 Der allgemeine Betriebsablauf (wiederverwendbar)

Basierend auf der allgemeinen Systemstruktur kann der Verfahrenstechniker für den Betriebszustand „Betrieb“ den Ablauf definieren.

```
ACTIVITY Betrieb
VAR
END_VAR
```

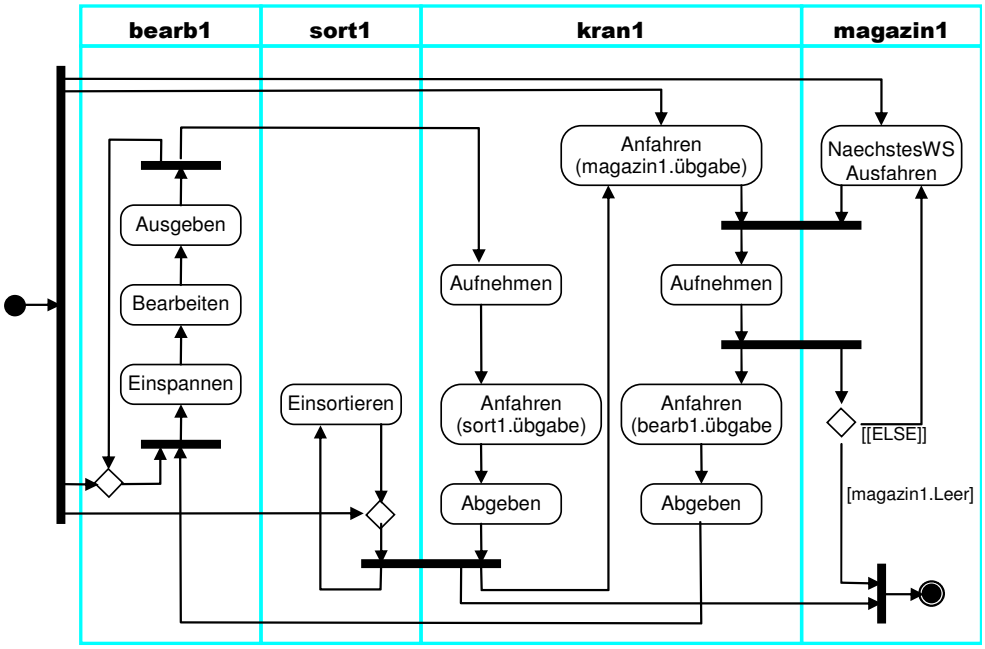


Abb. 2.4: Allgemeiner Betriebsablauf als UML-Aktivitätsdiagramm

Im UML-Klassendiagramm (Abb. 2.2) wird aus der Toolbox ein neuer Ablauf, d.h. eine neue Operation mit UML-Aktivitätsdiagramm als Sprache, in die Klasse Anlage4T gezogen und „Betrieb“ genannt. Der Editor für „Betrieb“ zeigt ein Aktivitätsdiagramm (Abb. 2.4). Es enthält für jedes Aggregat eine so genannte Swimlane zur Definition der Aktivitäten dieses Aggregats innerhalb des Gesamtablaufs von „Betrieb“.

Wiederverwendung mit einem Baukasten von fertig definierten Aggregaten:

- Bei Auswahl einer Swimlane werden in der Toolbox die für das entsprechende Aggregat in der Baukastenbibliothek bereits definierten Aktivitäten aufgelistet. Aus der Toolbox zieht der Verfahrenstechniker die gewünschten Aktivitäten des Aggregats in das Diagramm und verbindet sie zu dem gewünschten Gesamtablauf (z.B. „Aufnehmen“ und „Abgeben“ von Kran).
- Die in der Baukastenbibliothek für ein Aggregat definierten Eigenschaften können verwendet werden

- als Parameter von parametrierten Aktivitäten verwendet werden (hier: die Eigenschaft „übergabe“, d.h. Übergabeposition, von Magazin, Bearbeitung und Sortierband in der Kran-Aktivität „anfahren“);
- zur Formulierung von Bedingungen bei bedingten Verzweigungen (hier: die Eigenschaft „Leer“ von Bevorratung).

Würden nicht die vordefinierten Aktivitäten der Aggregate wiederverwendet werden, so müssten die Aktivitäten neu definiert werden. Als Beschreibungssprache können dabei (unter anderem) Aktivitätsdiagramme und Zustandsautomaten dienen. Also könnte der Verfahrenstechniker die benötigte Aktivität möglicherweise auch selbst definieren.

Wiederverwendbarkeit: Der definierte Ablauf ist wiederverwendbar für alle zukünftigen konkreten Ausprägungen von Anlage4T (siehe 2.3.4), da er nur auf Aktivitäten und Eigenschaften aufbaut, welche alle Ausprägungen der Aggregate besitzen. Was genau passiert im Schritt „NaechstesWSAusfahren“ ist durch das Aktivitätsdiagramm noch nicht festgelegt. Es wird sich automatisch daraus ergeben, welche konkrete Ausprägung von Magazin in der konkreten Ausprägung von Anlage4T zum Einsatz kommt (siehe 2.3.5).

2.3.4 Die konkrete Ausprägung

Nach, oder parallel zu, der Definition der allgemeinen Abläufe kann der Maschinenbauer die konkrete Ausprägung der Anlage (konkret aus funktionaler Sicht) festlegen. In unserem Beispiel ist das die Anlage4T-Variante von Typ Stempelanlage.

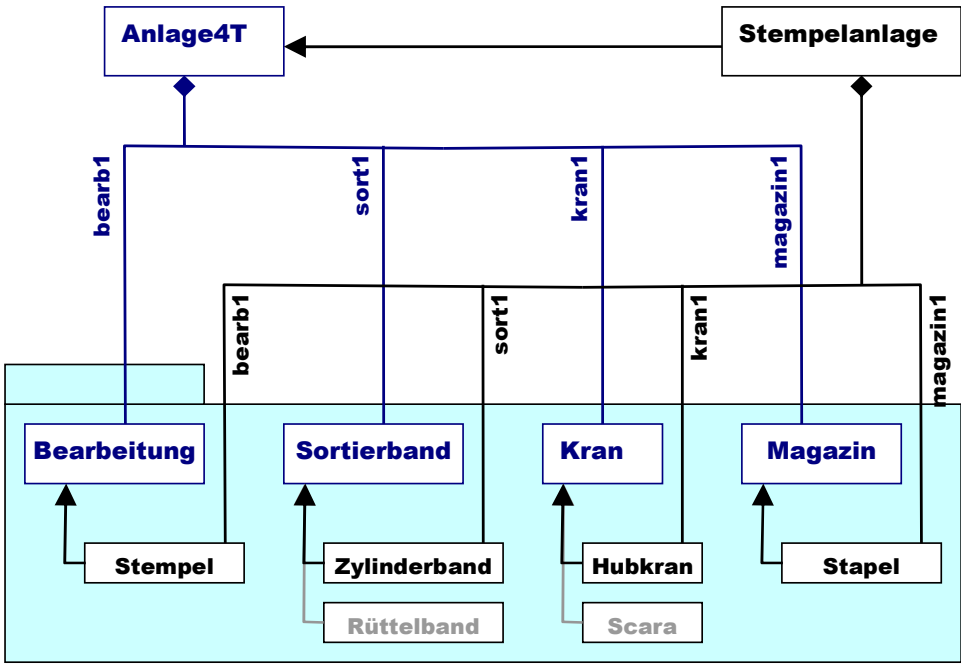


Abb. 2.5: Konkrete Anlagenstruktur als UML-Klassendiagramm

Dazu erweitert er das Diagramm aus dem ersten Schritt (oder eine Kopie davon):

Er erzeugt die neue Klasse Stempelanlage und deklariert sie als Spezialfall von Anlage4T.

Er zieht Kompositionsbeziehungen zu den gewünschten Spezialfällen von Anlage4T's allgemeinen Aggregatklassen (dabei werden die Rollennamen wie in Anlage4T's Komposition verwendet).

Wiederverwendung: Wenn mit einem Baukasten von fertig definierten Aggregaten gearbeitet wird, müssen folgende Schritte durchgeführt werden:

Ein UML-Klassendiagramm wird angelegt und über die Toolbox des Diagramms wird die Baukastenbibliothek eingefügt.

Aus der Toolbox wird eine neue Klasse in das Diagramm gezogen und Anlage4T genannt.

Über die Toolbox werden Kompositionsbeziehungen eingefügt, die von der neuen Anlage zu ihren Aggregaten in der Bibliothek gehen (Bearbeitung, etc.). Den Aggregaten können eindeutige Namen im Kontext der neuen Anlage gegeben werden („bearb1", etc.).

Wiederverwendbarkeit: In diesem Schritt wird die Beschreibung bewusst allgemein gehalten, um in zukünftigen Entwicklungen weiterer Anlagenvarianten eine Wiederverwendung dieses Grundaufbaus und aller darauf definierten Abläufe und

Zustandsautomaten zu ermöglichen (nach dem Master-Konzept statt Copy&Modify [2]): Es wird noch nicht festgelegt sein, welches Magazin, welcher Kran, welche Bearbeitungseinheit und welches Sortierband zum Einsatz kommt.

2.3.5 Das fertige SPS-Projekt

Um nun ein fertiges SPS-Projekt zu erhalten, muss der klassische CoDeSys-Programmierer die bereits gemachten Arbeiten nicht duplizieren, sondern sie nur um die unvermeidlichen SPS-Spezifika ergänzen:

Das eigentliche Steuerungsprogramm besteht nur daraus, eine Applikation anzulegen, dazu eine Task und ein Programm, welches die „Main"-Operation einer Stempelanlagen-Instanz aufruft (siehe Abb. 2.6).

Auf der E/A-Seite müssen die Feldbusgeräte konfiguriert werden und das Mapping der E/A-Signale auf die Eigenschaften „StartTaster" und „ResetTaster" der Stempelanlagen-Instanz.

(Über unsere Betrachtungen hinausgehend könnten natürlich noch zusätzliche Aufgaben auf der gleichen Steuerung auszuprogrammieren sein, wie Datenaustausch, Visualisierung, Überwachungsfunktionen.)

```
PROGRAM PLC_PRG
VAR
  anlage : Stempelanlage;
END_VAR
```

```
anlage.Main();
```

Abb. 2.6: Das ganze Steuerungsprogramm

Wie funktioniert der Zusammenhang zwischen dem Steuerungsprogramm und den vorausgegangenen UML-Diagrammen?

Indem Abb. 2.5 eine Stempelanlage als Spezialfall von Anlage4T definiert hat, erben die Stempelanlagen-Instanzen die allgemeine Definition der Operationen „Main" und „Betrieb" aus Abb. 2.3 und Abb. 2.4. Wenn das geerbte „Main" ausgeführt wird, dann wird darin das geerbte „Betrieb" aufgerufen. Ein darin enthaltener Aufruf wie „NaechstesWSAusfahren" in der „magazin1"-Swimlane führt durch dynamisches Binden zum Aufruf der in Stapel definierten Implementierung von „NaechstesWSAusfahren", weil Stapel der Typ von „magazin1" innerhalb von Stempelanlage ist. Diese Implementierung liegt in der Baukastenbibliothek.

2.4 Fazit

Ohne es zu merken, haben Verfahrenstechniker und Maschinenbauer Teile des Steuerungscodes geschrieben. Das war möglich, weil ihre Systembeschreibungen in Form von UML-Diagrammen nicht aussehen wie Programmierung aber dennoch ausführbar sind. Für den Steuerungsprogrammierer blieb im Beispiel nicht mehr viel zu programmieren. Aber sehr viel seiner Arbeit steckt natürlich bereits als Vorleistung in den tieferen Schichten der Baukastenbibliothek.

Zu den Features des objektorientierten Programmierens von CoDeSys 3 [4] muss der Steuerungsprogrammierer für das objektorientierte Modellieren vor allem einer Verallgemeinerung lernen:

Neben den klassischen Methoden mit einer funktionalen Semantik (die „schnellen" Operationen von Abschnitt 2.2), gibt es jetzt Methoden mit einem lokalen Zustand („**Activity**"), die Abläufe implementiert, der sich über mehrere SPS-Zyklen hinweg erstreckt (beschrieben als UML-Aktivitätsdiagramm oder UML-Zustandsmaschine, oder vielleicht auch einmal in IEC-Ablaufsprache).

2.5 Referenzen

[1] J. Rumbaugh, I Jacobson, G Booch: The Unified Modeling Language Reference Manual, zweite Auflage, 2005.

[2] A. Fritsch: Durch Modularisierung zum mechatronischen Baukastensystem, 503-510 im Tagungsband SPS/IPC/DRIVES 2008.

[3] D. Witsch, U Schünemann, B Vogel-Heuser: Steigerung der Effizienz und Qualität von Steuerungsprogrammen durch Objektorientierung und UML, 511-519 im Tagungsband SPS/IPC/DRIVES 2008.

[4] U. Schünemann: Objektorientierung in CoDeSys 3, in: Automation & Embedded Systems, Oldenbourg Verlag , 2008

[5] F. Capelle: Modulare Maschine, in: Automation & Embedded Systems, Oldenbourg Verlag , 2008

3 Einsatz von UML-Diagrammen in der Steuerungsprogrammierung

Daniel Witsch, Birgit Vogel-Heuser

Fachgebiet Eingebettete Systeme, Universität Kassel

Zusammenfassung: Immer mehr Funktionen von Maschinen und Anlagen werden heute durch Software realisiert. Steuerungssoftware wird immer komplexer. Daher kommt dem effizienten Entwurf und der Qualität der Steuerungs-Software eine immer größere Bedeutung zu. Neben textuellen Sprachen haben sich grafische Sprachen etabliert, mit denen der Bruch der prozeduralen Programmierung zwischen Design und Codierung sowie zwischen Programmierung und anderen Fachbereichen, überbrückt wird. Die Unified Modeling Language (UML) ist eine solche grafische Sprache zur Spezifikation, Darstellung, Konstruktion und Dokumentation von objektorientierten Systemen. Die Anwendbarkeit von UML und die Umsetzung von Klassendiagrammen, Statecharts und Aktivitätsdiagrammen in einer weit verbreiteten Programmierumgebung für Steuerungen wurden in einem Forschungsprojekt untersucht. Dieser Beitrag stellt die wesentlichen technischen Ergebnisse dieses Projektes vor.

3.1 Einleitung

Dieser Beitrag stellt die Projektergebnisse des von der Stiftung Industrieforschung geförderten Projektes "Steigerung der Effizienz und Qualität im Software-Engineering der Automatisierungstechnik für die Domäne des Maschinen- und Anlagenbaus" (http://www.es.eecs.uni-kassel.de/uml2iec61131/) zusammenfassend vor. Das Projekt lief vom 01.02.2007 bis zum 31.01.2009 und wurde in Zusammenarbeit mit den Unternehmen 3S-Software, Beckhoff Automation, Elau, SIG-Combibloc und teamtechnik sowie einer Reihe weiterer interessierter Unternehmen (Industriebeirat) mit dem Ziel durchgeführt, eine angepasste und auf wenige Diagrammtypen reduzierte Untermenge der UML in CoDeSys V3 prototypisch zu implementieren.

Die Unified Modeling Language (UML) [OMG04] ist eine grafische Sprache zur Spezifikation, Darstellung, Konstruktion und Dokumentation von objektorientierten Systemen. Sie hat den Anspruch eine allgemein verständliche Diskussionsbasis für Systementwicklungen zu sein. In ihr sind 13 unterschiedliche Diagrammarten integriert, die teilweise bereits vorher in anderen Bereichen ähnlich eingesetzt wurden. Die UML wird von der Object Management Group (OMG, http://www.omg.org/), einem internationalen Industriekonsortium, gepflegt und ist in unterschiedlichen Versionen verfügbar. Eine dieser Versionen ist ebenso als Standard ISO/IEC 19501 [ISO05] verabschiedet.

Die Anwendung der UML kann mit zwei unterschiedlichen Zielen verfolgt werden.

Der übliche Anwendungsfall ist die Systemmodellierung bzw. –analyse. Hierbei sind die resultierenden Modelle dadurch gekennzeichnet, dass diese unvollständig, mehrdeutig und oft skizzenhaft ausgeführt sind. Sie dienen in erster Linie als Diskussionsbasis, für die Dokumentation bestimmter Teilaspekte oder zur Spezifikation der Systemanforderungen.

Ein andere Möglichkeit der Nutzung von UML sind ausführbare Modelle. Hierbei werden die Modelle soweit ausformuliert, dass diese eine abstrakte Programmiersprache darstellen, aus der direkt der Anwendungscode generiert werden kann. Dieses Vorgehen wird auch oft als Model-Driven-Architecture (MDA) [MDA09] bezeichnet. Aus industrieller Sicht sind beide Anwendungsszenarien nützlich. Ein besonderer Mehrwert lässt sich jedoch durch die Verknüpfung der beiden Anwendungsszenarien erzielen. Dieser Ansatz wurde auch in dem hier vorgestellten Projekt verfolgt.

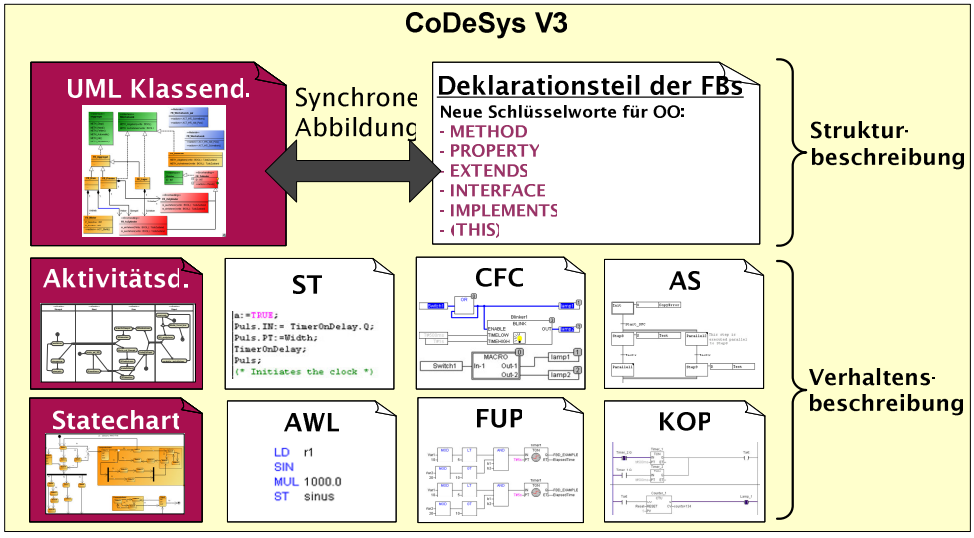


Abb. 3.1: Projektidee: Integration von UML-Editoren in Programmierumgebung für Steuerungen

Neben der Modellierung von Verhalten an der Schnittstelle zwischen Technologe und Programmierer oder für Motion-Control Anwendungen, wurde ein großes Augenmerk auf die Modellierung von Software-Strukturen gelegt. Die integrierten Klassendiagramme können objektorientierte Design-Aspekte in der Programmierumgebung grafisch darstellen. Für die Modellierung von Verhalten werden Zustandsautomaten und Aktivitätsdiagramme realisiert (s. Abb. 3.1). Die grafischen UML-Editoren verschmelzen mit den existierenden IEC 61131-3 Editoren. Diese starke Integration ermöglicht z.B. das Debugging in den Diagrammen, eine unmittelbare Abbildung zwischen Code und Diagramm und die Verwendung der UML in der für den Steuerungsprogrammierer gewohnten Umgebung. Die UML-Diagramme können einerseits als reine Modellierung (ohne Code zu erzeugen) dienen oder andererseits direkt zur Codeerzeugung im Sinne einer ausführbaren UML eingesetzt werden.

Nachfolgend sollen die einzelnen Editoren und deren Einsatzgebiete dargestellt werden.

3.2 Das Klassendiagramm

Ein Klassendiagramm stellt im Wesentlichen einen Ausschnitt der Klassen und deren jeweilige Beziehungen untereinander in einem objektorientierten Software-System dar.

Eine Klasse ist eine logische Software-Einheit in der Objektorientierung. In einer Klasse werden Daten und Operationen, die auf diesen arbeiten, gekapselt. Sie stellt einen Variablentyp dar. Klassen bilden somit Vorlagen, die - analog zu allen Typen (z.B. Integer, String) - instanziiert werden. Die Ausprägungen einer Klasse werden in der Regel als Objekt oder Instanz bezeichnet. Für die Modellierung mit der UML im CoDeSys Kontext wird außerdem die Bezeichnung Rolle synonym für eine Instanz verwendet, die einer Klasse als Variablenbestandteil zugeordnet ist. In CoDeSys 3 wurde der Begriff des IEC 61131-3 Function Blocks zur Klasse erweitert und es stehen zur Definition von Objekten bzw. Klassen verschiedene Sprachmittel zur Verfügung. Diese sind in Abb. 3.2 dargestellt. (Die nachfolgenden Aufzählungsnummern entsprechen den Nummern in Abb. 3.2.)

1. **Interne Objektdaten**: Eine Klasse verfügt zunächst - analog zum bisherigen Funktionsblock - über eine eigene Variablendeklaration. Durch diese werden die internen Objektdaten festgelegt. Intern bedeutet hierbei, dass nur Elemente der Klasse selbst (z.B. Methoden oder Eigenschaften, s.u.) auf diese Daten direkt zugreifen können. Andere Objekte dürfen nur über klar definierte Zugriffspunkte interne Daten verändern oder lesen.
2. **Eigenschaften:** Diese Zugriffspunkte von außen auf die internen Daten eines Objektes sind die Eigenschaften (engl. Properties) eines Objektes bzw. die Get- und Set-Methoden dieser Eigenschaft. Eigenschaften stellen eine abstrakte, externe Sicht auf die Objektdaten dar. Ein Property erhält bei seiner Deklaration eine Get- und eine Set-Methode. Diese werden implizit bei einem lesenden oder schreibenden Zugriff aufgerufen und ermöglichen bspw. die Überprüfung der Datenkonsistenz oder von Werteanpassungen. Das Property an sich stellt keine Variable der Klasse dar, kann aber auf die internen Variablen dieser zugreifen und diese verändern. Eine Klasse kann über beliebig viele Eigenschaften verfügen.
3. **Methoden**: Methoden stellen die Operationen des Objekts dar und werden mit dem Schlüsselwort METHOD eingeleitet. Sie sind spezielle POE-Typen mit Zugriff auf die internen Variablen ihres Objektes, einem Rückgabewert, und einem eigenen Deklarationsteil. Eine Klasse kann beliebig viele Methoden beinhalten.
4. **Initialisierung des Objekts** (mit Parametern): Es kann eine spezielle Methode namens FB_Init definiert werden, welche zur Initialisierung des Objekts aufgerufen wird. Hierdurch können die internen Objektdaten so vorbereitet werden, dass die abstrakte Sicht und die Operationen auf ihnen korrekt arbeiten können.
5. **Erzeugen von Objekten (Instanziierung)**: Das Erzeugen von Objekten erfolgt durch Deklaration einer Variablen vom Typ der Klasse (Klassenbasierte Objektdefinition). Falls in FB_Init Initialisierungsparameter definiert sind, müssen diese in der Deklaration angegeben werden (vgl. Punkt 4).

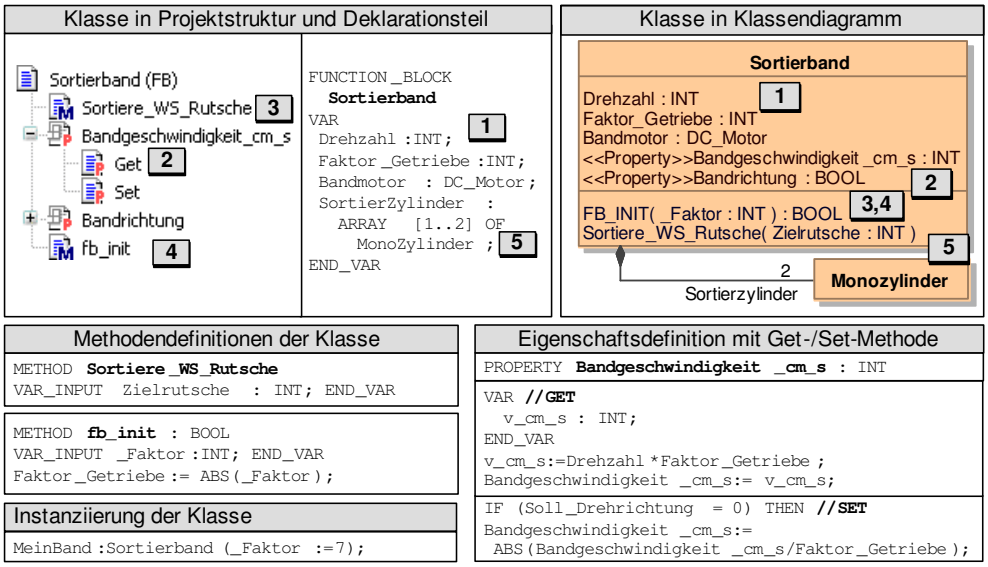


Abb. 3.2: *Klasse mit Methoden, Eigenschaften (Properties) und Initialisierungsmethode in CoDeSys Projektstruktur und Code sowie in der Darstellung des UML Klassendiagramms).*

Abb. 3.2 beinhaltet zusätzlich die Klassendefinition (rechts, oben) in Form eines UML-Klassendiagramms. Hier dargestellt ist die Klasse „Sortierband“ mit ihren internen Variablen, ihren Eigenschaften (gekennzeichnet durch <<Property>>) und ihren Methoden samt Parametern. Neben der Klasse Sortierband ist ebenfalls die Klasse Monozylinder (Variablen, Eigenschaften und Methoden sind ausgeblendet) in diesem Klassendiagramm und die Beziehung der beiden Klassen untereinander dargestellt. Dieses Beispiel zeigt, dass die Klasse Sortierband ein Array von Monozylindern der Größe zwei beinhaltet und dass dieses Array den Instanznamen Sortierzylinder trägt. Die Darstellung als Klasse stellt dabei die ansonsten über die Projektstruktur und Textfenster verteilten Klasseninformationen in einer aggregierten Form dar und sorgt so für höhere Übersichtlichkeit.

3.2.1 Semantik der Elemente im Klassendiagramm

Um die Beziehungen zwischen Klassen auszudrücken, bietet der im Rahmen des Projektes erstellte Klassendiagramm-Editor die in Abb. 3.3 dargestellten Relationstypen an. Wird eine solche Relation in einem Klassendiagramm gezeichnet, wird der entsprechende Code unmittelbar erstellt. Wird innerhalb des ausprogrammierten Codes eine solche Beziehung gefunden (oder wird eine Änderung im Code festgestellt), wird die grafische Darstellung automatisch synchronisiert.

Solche Beziehungen können Enthalten-Beziehungen (Kompositionen) oder Beziehungen sein, die ein gerichtetes oder beidseitiges "Kennen" unterschiedlicher Klassen beschreiben (Assoziationen). Diese Beziehungen werden in UML-Klassendiagrammen durch Relationen (Linien) zwischen Klassen ausgedrückt. Den Relationen des Klassendiagramms

wurden im Rahmen des Forschungsprojektes eindeutige Entsprechungen im Code des Deklarationsteils der Klassen bzw. der Funktions-Blöcke zugeordnet.

So entspricht eine Assoziationsbeziehung einem Pointer und eine Kompositionsbeziehung einer Instanziierung einer anderen Klasse (s. Abb. 3.3, obere Zeile). Jede Relation kann außerdem Vielfachheitsinformationen (Kardinalitäten) besitzen, die anzeigen, wie viele Pointer bzw. Instanzen angelegt werden. Wird eine Kardinalität größer als Eins angegeben, wird ein Array angelegt (s.Abb. 3.3, untere Zeile).

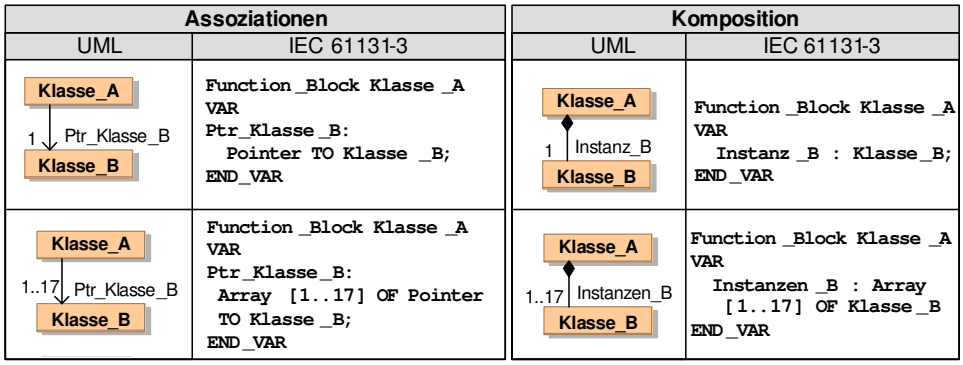


Abb. 3.3: Assoziations- und Kompositionsbeziehungen in Klassendiagrammen

Neben den Klassen gibt es so genannte Interfaces (Schnittstellen). Ein Interface ist eine Menge von Methoden- und Eigenschaftsdeklaration, die in Form einer Klasse zusammengefasst werden. Diese Interface-Klasse enthält keinerlei Implementierung. Ein solches Interface ist als Vereinbarung mit Typencharakter zu sehen. Ein Interface entspricht einem Funktionsbaustein ohne Implementierungs- und Deklarationsteil und mit optionalen unterlagerten Methoden und Properties, die keinen Implementierungsteil und keine lokalen Variablen besitzen. Im Klassendiagram sind Interfaces markiert mit <<Interface>> (s. Abb. 3.4).

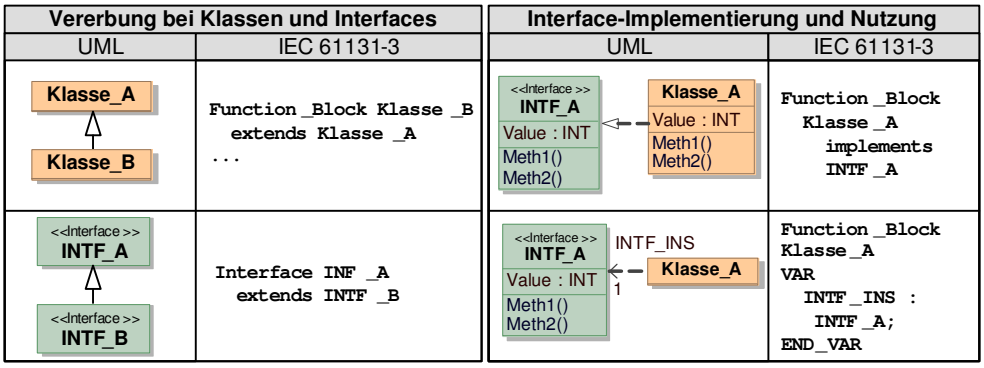


Abb. 3.4: Vererbungs- und Implementierungsbeziehungen

Auch zwischen Interfaces kann eine Spezialisierungsbeziehung bestehen (EXTENDS). Wenn ein Funktionsblock von einem Interface erbt, dann wird dies mit dem Schlüsselwort IMPLEMENTS codiert. Der erbende Funktionsblock muss dann alle Methoden und Eigenschaften, die in dem Interface definiert wurden, übernehmen und mit Inhalt füllen. Im Gegenzug kann er dann an allen Stellen im Programmcode eingesetzt werden, an denen als Typ einer Variablen das Interface benannt wurde. Die Anwendung, die Vorteile sowie die entsprechende Modellierung von Interfaces in der UML wurden in Witsch et al. [WWV08] anhand des gleichen Anwendungsbeispiels ausführlicher vorgestellt. Ebenso werden dort Ansätze aufgezeigt, wie ausgehend von einer maschinenbaulichen bzw. automatisierungstechnischen Betrachtung objektorientierte Steuerungssoftware entwickelt werden kann.

Pakete (Packages)
Pakete stellen ein wichtiges Strukturierungsmittel in der UML dar. Pakete können komplexe Sub-Strukturen von Software-Projekten oder aber Fremdkomponenten (z.B. Bibliotheken) enthalten. Pakete können hier einerseits Ordner der Projektstruktur aber auch externe Bibliotheken darstellen. Oft ist es wichtig zu erkennen, welche Abhängigkeiten zwischen Paketen und Klassen oder anderen Paketen bestehen. Hierfür stehen spezielle Funktionen bereit, die die Abhängigkeiten von Klassen zu Bibliotheken visualisieren können.

3.2.2 Anwendungsszenarien für das Klassendiagramm

Das Klassendiagramm kann in vier unterschiedlichen Arten verwendet werden.

Dokumentation
Klassendiagramme können als Analyse-Diagramme erstellt werden. Dann wird aus ihnen kein Code erzeugt. Sie dienen dann zur Formulierung von Anforderungen oder groben Ideen in einer frühen Entwicklungsphase. Wenn ein Dokumentations-Klassendiagramm soweit ausgereift ist, dass eine Implementierung darauf aufbauend vorgenommen werden soll, kann dieses automatisch in ein Implementierungs-Klassendiagramm umkopiert werden (es wird eine Kopie in der Projektstruktur mit dem gleichen Layout und direkt die entsprechende Software-Struktur angelegt).

Reengineering
Das Klassendiagramm ist in der Lage automatisch vorhandene Projektstrukturen (mit/ohne Objektorientierung) einzulesen und grafisch darzustellen. Damit können existierende Projekte leicht analysiert und hinsichtlich der Software-Struktur überarbeitet werden. Auch für die Erstellung einer Dokumentation oder für die Einarbeitung in eine fremde Software ist diese Funktion hilfreich. Nach dem Einlesen des Klassendiagramms stehen dem Anwender alle Möglichkeiten des Forward-Engineering zur Verfügung.

Forward-Engineering
Wird ein Klassendiagramm für die Implementierung genutzt, wird die OO-IEC 61131-3 Software-Struktur immer synchron zum Diagramm gehalten. Alle Änderungen, die grafisch im Diagramm vorgenommen werden, haben einen unmittelbaren Effekt in der Programmierumgebung. Andersherum können Änderungen in der Software-Struktur zu beliebigen Zeitpunkten mit der grafischen Darstellung synchronisiert werden.

Online-Ansicht
Ist die Programmierumgebung online mit der Steuerung verbunden, bietet das Klassendiagramm eine Online-Ansicht auf die Software-Struktur an. Es können aktuelle Attributwerte der jeweiligen Instanzen von Klassen angezeigt werden.

3.2.3 Oberfläche des Klassendiagramm-Editors

Die oben dargestellten Modellierungselemente können mittels einer Werkzeugleiste auf der Diagrammfläche entsprechend angeordnet und verbunden werden. Des Weiteren sind textuelle Eingaben direkt in Diagrammelementen (z.B. Klassennamen, Methodenname, Property-Namen) und über ein Eigenschaften-Fenster, welches immer zum aktuell markierten Element angezeigt wird, möglich. Abb. 3.5 zeigt die Oberfläche des Klassendiagrammeditors.

Stereotypen und Sichten
Um für unterschiedliche Anwendergruppen (Systemarchitekt, Programmierer, Dokumentation etc.) verschiedenen Sichten auf ein Diagramm zu ermöglichen oder andere situationsabhängige Auswertungen eines Diagramms vorzunehmen, steht ein stereotypen-basierter Mechanismus zur Verfügung. Diagrammelemente können mit frei definierbaren Stereotypen (textuelle Auszeichnungen) versehen werden. Über einen Sichten-Editor können beliebige Sichten definiert werden, die die Stereotypen auswerten und so Diagrammelemente in bestimmten Sichten gezielt ausblenden. So können reduzierte Sichten auf komplexe Diagramme für Dokumentationszwecke erzeugt werden.

Außerdem können farbliche Kennzeichnungen und Grafiken mit Stereotypen verknüpft werden, sodass Diagramme auch mit leicht wiedererkennbaren Bildern übersichtlich und ansprechend gestaltet werden können, was eine wesentliche Anforderung von Technologen darstellt.

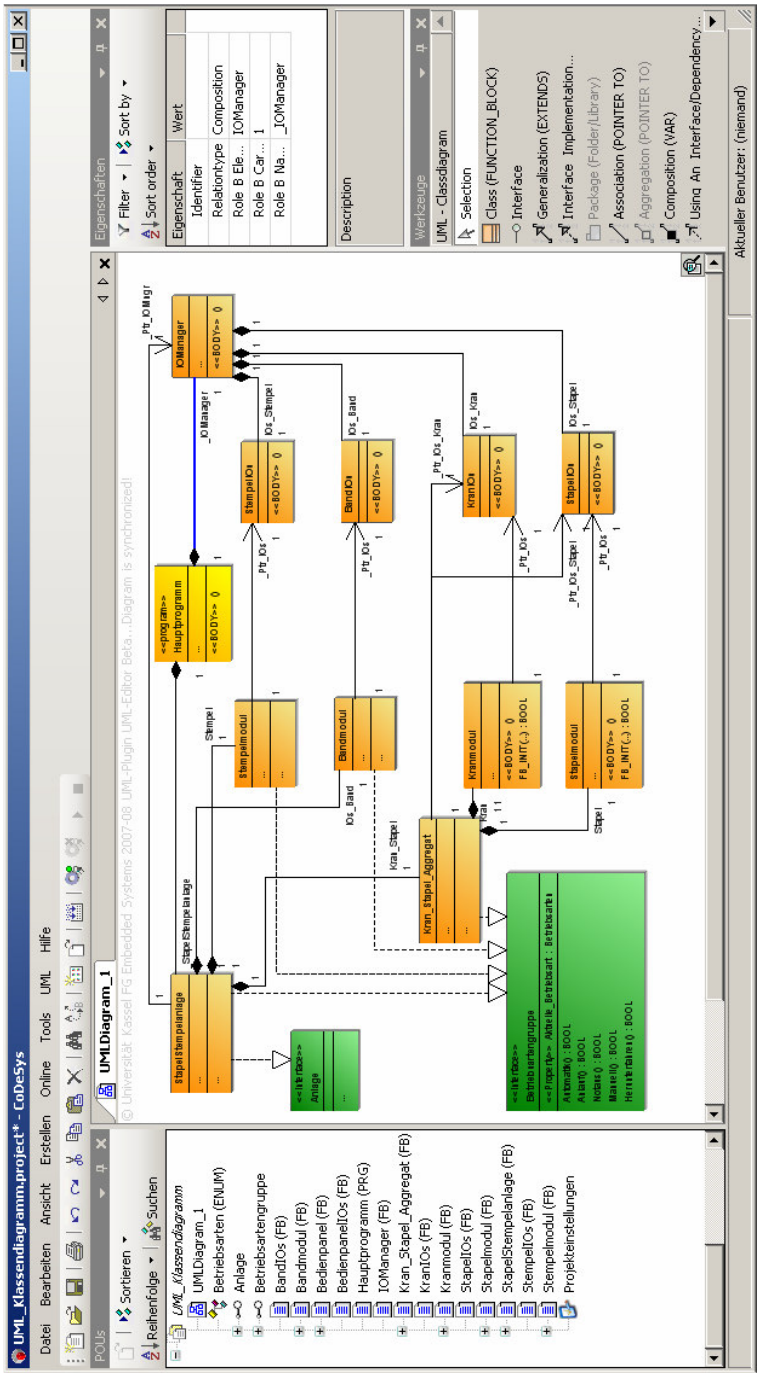


Abb. 3.5 Klassendiagramm in CoDeSys V3

3.2.4 Anwendungsbeispiel für die Systemmodellierung mit Klassendiagrammen

Nachfolgend soll anhand eines einfachen Systemmodells der Software-Entwicklungsprozess mittels Klassendiagrammen kurz skizziert werden. Zunächst wird dazu ein Analysemodell aus technologischer (mechatronischer) Sichtweise abgeleitet und dieses nachher aus Sicht der Software-Entwicklung konkretisiert.

Das zugrunde liegende technische System stellt einen einfachen, diskreten, fertigungstechnischen Prozess nach (s. Abb. 3.6 und Abb. 3.10). Aus einem Materialvorrat (Stapel) werden Werkstücke hinausgeschoben und anschließend analysiert. Drei verschiedene Werkstück-Typen werden hierbei unterschieden: Werkstücke aus hellem Metall, Werkstücke aus dunklem Kunststoff und Werkstücke aus hellem Kunststoff. Nach der Werkstückanalyse können die Werkstücke mittels einer Pick&Place-Einheit (Kran) aufgenommen und - abhängig von dem erkannten Werkstoff - zu den anderen beiden Prozessstationen verfahren werden. Eine Prozessstation, der Stempel, dient der Bearbeitung der Werkstücke. Hier kann mittels eines pneumatischen Kolbens mechanischer Druck auf die Werkstücke ausgeübt werden. Die dritte Prozessstation dient der Sortierung der Werkstücke nach der Analyse oder der Weiterbearbeitung. Die Sortierung findet auf einem Förderband (Band) statt, auf welchem der Kran die Werkstücke absetzen kann. An dem Band befinden sich zwei Ausschleuser (pneumatische Kolben), die vorbeilaufende Werkstücke vom Band in eine Rutsche stoßen können.

Erstellung des Analyse-Klassendiagramms: Ableitung mechatronischer Klassen
Bei der objektorientierten Vorgehensweise, lässt sich in einer frühen Software-Entwicklungsphase die maschinenbauliche Struktur direkt auf die Software-Struktur übertragen. Diese software-technische Nachbildung dient dann als Ausgangsmodell für den Software-technischen Entwurf, bei dem dieses Abbild z.B. um Schnittstellen, Vererbungen und andere Design-Aspekte erweitert wird. Die Klassen, die eine direkte Entsprechung im maschinenbaulichen Entwurf der Anlage haben, sollen im Folgenden „mechatronische Klassen“ und die korrespondierenden maschinenbaulichen Subsysteme „mechatronische Module“ heißen. Nachfolgend soll dieses Vorgehen am Beispiel der Sortier- und Stempel-Anlage demonstriert werden.

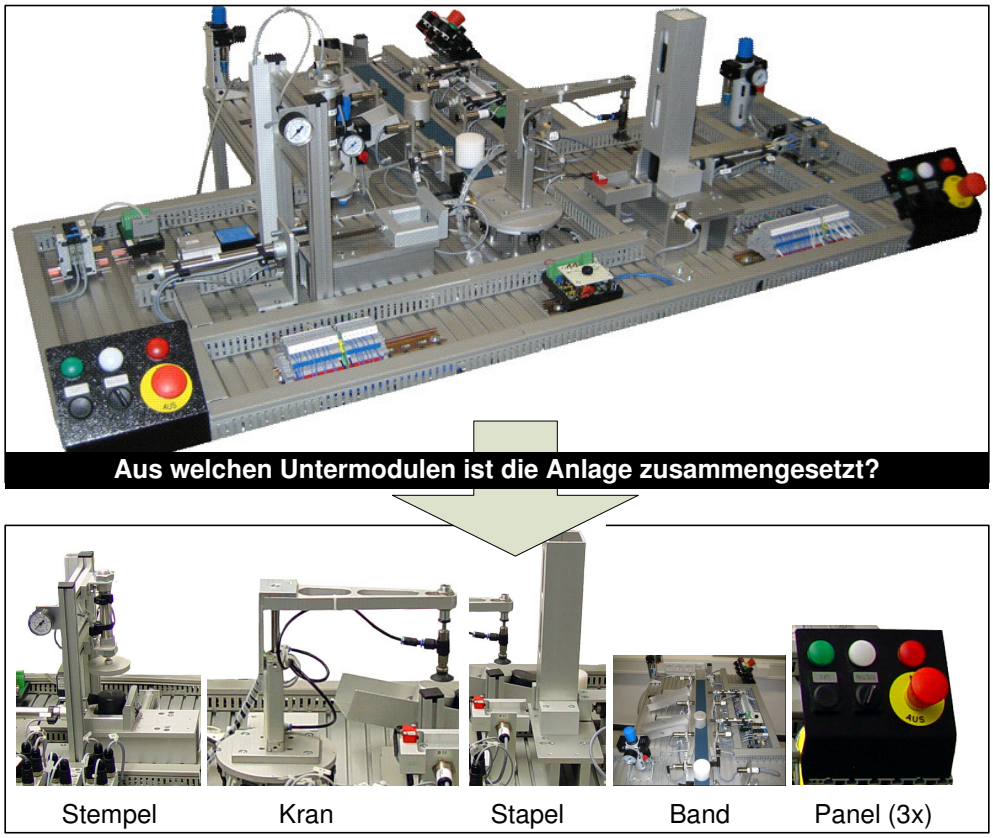


Abb. 3.6 Unterteilung der Gesamtanlage in Subsysteme

Die Zerlegung des Gesamtsystems kann auf Grundlage verschiedener Indizien erfolgen. Oftmals weisen die folgenden Merkmale auf ein Subsystem hin:

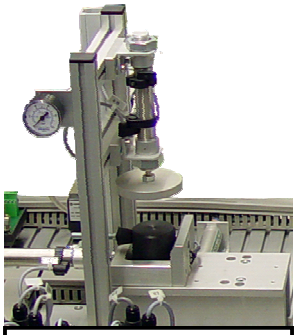
- Der betrachtete Systemausschnitt kann auch sinnvoll in einem anderen Gesamtsystem eingesetzt werden.
- Der betrachtete Systemausschnitt wird von einer eigenen Abteilung/Zulieferer hergestellt.
- Der betrachtete Systemausschnitt verfügt über einen eigenen Kennzeichnungsbereich (z.B. bei Anlagenkennzeichnungssystemen)
- Der betrachtete Systemausschnitt verfügt über einen eigenen Stromkreis, Notauskreis, bildet eine eigene Betriebsartengruppe (verfügt über ein eigenes Bediengerät).
- Der betrachtete Systemausschnitt ist für die Durchführung eines für den Gesamtprozess charakteristischen Prozessschritts verantwortlich.
- Der betrachtete Systemausschnitt existiert mehrfach in gleicher oder ähnlicher Art im System.

In Abb. 3.6 ist diese Einteilung exemplarisch vorgenommen: Der Stempel, das Band und der Stapel zusammen mit dem Kran verfügen über ein eigenes Bedienpanel und können daher als eigene Betriebsartengruppe betrachtet werden. Der Kran könnte ebenso in einem anderen Kontext eingesetzt werden. Die Bedienpanels kommen in dreifacher Ausfertigung vor.

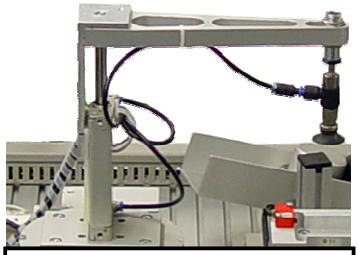
Nach der Zerlegung des Gesamtsystems in mechatronische Klassen, gilt es, das Wissen, welches jede mechatronische Klasse über sich selber hat und ihre Fähigkeiten zu sammeln (vgl. Abb. 3.7).

Das Wissen der mechatronischen Module erstreckt sich typischerweise auf den eigenen Zustand (z.B. Klasse Kran: Drehwinkel, links drehend, rechts drehend usw.). Die Information über den eigenen Zustand wird in der Regel aus den dazugehörigen Sensorwerten und den vorangegangenen Zuständen abgeleitet. Ein Beispiel hierfür ist der Winkel des Krans. Der Wert selber wird über einen Analogeingang ermittelt, muss jedoch über einen bestimmten Faktor aus der Integer-Variable in das Winkelmaß umgerechnet werden. Andere Zustandsinformationen wie z.B. „eingespannt“ oder „ausgefahren“ beim Stempel können unmittelbar mit dem entsprechenden digitalen Eingang verknüpft werden. Das Wissen der Klassen wird in Form von Attributen bzw. Eigenschaften (Properties) im oberen Bereich der Klasse notiert (vgl. Abb. 3.7). Im unteren Bereich der Klasse werden die spezifischen Fähigkeiten der mechatronischen Klassen als Methoden zusammengefasst. Bei der heutigen Steuerungsprogrammierung wird ein Satz an Funktionsbausteinen und Funktionen für die Ansteuerung eines Moduls verwendet. Im objektorientierten Ansatz wird dieser Satz von Funktionalitäten als Methoden in einer Klasse zusammengefasst.

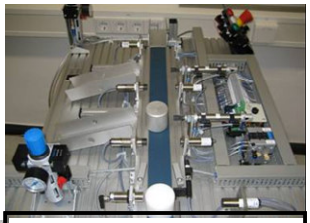
Eine besondere Stellung in diesen Überlegungen nimmt die mechatronische Klasse der Gesamtanlage ein. Diese enthält ihre unterlagerten mechatronischen Klassen und kann für die Erfüllung der Gesamtfunktionalität auf diese zugreifen. Die Methoden der mechatronischen Klasse „Anlage“ (vgl. Abb. 3.7, unten rechts) bilden die unterschiedlichen Betriebsarten der Anlage bzw. die vom Nutzer angefragte Funktionalität ab.



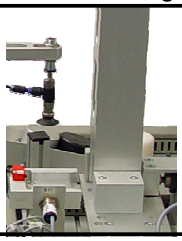
Stempel
• eingespannt • ausgefahren • Werkstück vorhanden • Stempel oben/unten • In Grundstellung fahren
• Werkstück einspannen • Werkstück ausfahren • Werkstück stempeln • Auf Notaus reagieren • In Grundstellung fahren



Kran
• Winkel • Position Stapel, Stempel, Band • Kran oben, unten • links drehend, rechts drehend • Werkstück aufgenommen
• Werkstück hochheben • Werkstück absetzen • Transportieren zu Position • Auf Notaus reagieren • In Grundstellung fahren

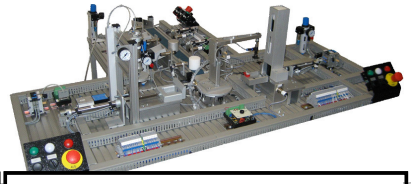


Band
• dreht vorwärts/rückwärts
• Einsortieren in Rutsche 1 • Einsortieren in Rutsche 2 • Einsortieren in Rutsche 3 • Anhalten • Auf Notaus reagieren • In Grundstellung fahren



Stapel
• Werkstück vereinzelt • Werkstück dunkel • Werkstück metall • Vorrat leer
• Werkstück rausschieben • Werkstück analysieren • Auf Notaus reagieren • In Grundstellung fahren

Panel
• Start gedrückt • Schalter manu/auto • Notaus geöffnet
• Lampe_Start_an/aus • Lampe_Manu an/aus • Lampe Notaus an/aus



Anlage
Weiß alles was ihre Objekte (Stapel, Stempel, Kran, Band Panel1, Panel2, Panel3) wissen!
• Stempeln und Sortieren in Automatik • Stempeln und Sortieren in Manuell • Auf Notaus reagieren • In Grundstellung fahren

Abb. 3.7: Grobentwurf der mechatronischen Klassen

Die in Abb. 3.7 dargestellten mechatronischen Klassen stehen in Beziehung zueinander. Die meisten Beziehungen zwischen Klassen (wie in Abb. 3.3 Abb. 3.4 dargestellt) beziehen sich auf eine software-technische und nicht auf eine mechatronische Sichtweise. Ausnahme bildet hier die Kompositionsbeziehung (Enthalten-Beziehung). So kann für das

vorliegende Beispiel ein einfaches Klassendiagramm aus mechatronischer Sichtweise erstellt werden, welches die Enthalten-Beziehungen mit ihren Vielfachheiten ausdrückt (vgl. Abb. 3.8). Hier wurde ausgedrückt, dass die mechatronische Klasse „Anlage“ jeweils eine Instanz (im Kontext von Objektorientierung auch Objekt genannt) vom „Stempel“, vom „Band“, vom „Kran“ und vom „Stapel“ enthält, jedoch drei Instanzen vom „Panel“. Zusätzlich wurden die verschiedenen Instanzen benannt (z.B. „Panel_Band“, „Panel_Stempel“ usw.).

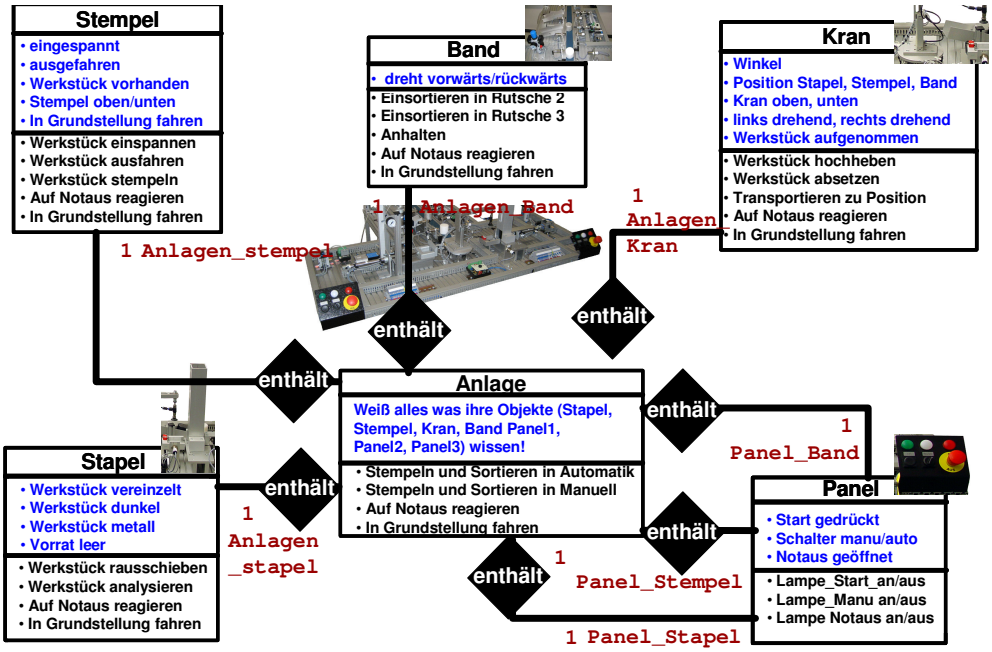


Abb. 3.8 : Klassendiagramm-Skizze mit Kompositionsbeziehungen zwischen mechatronischen Klassen

Die Skizze aus Abb. 3.8 kann direkt in dem Klassendiagramm-Editor umgesetzt werden. Abb. 3.9 zeigt das Ergebnis. Indem das Klassendiagramm mit dem Klassendiagramm-Editor gezeichnet wurde, ist auch die entsprechende Software-Struktur angelegt worden.

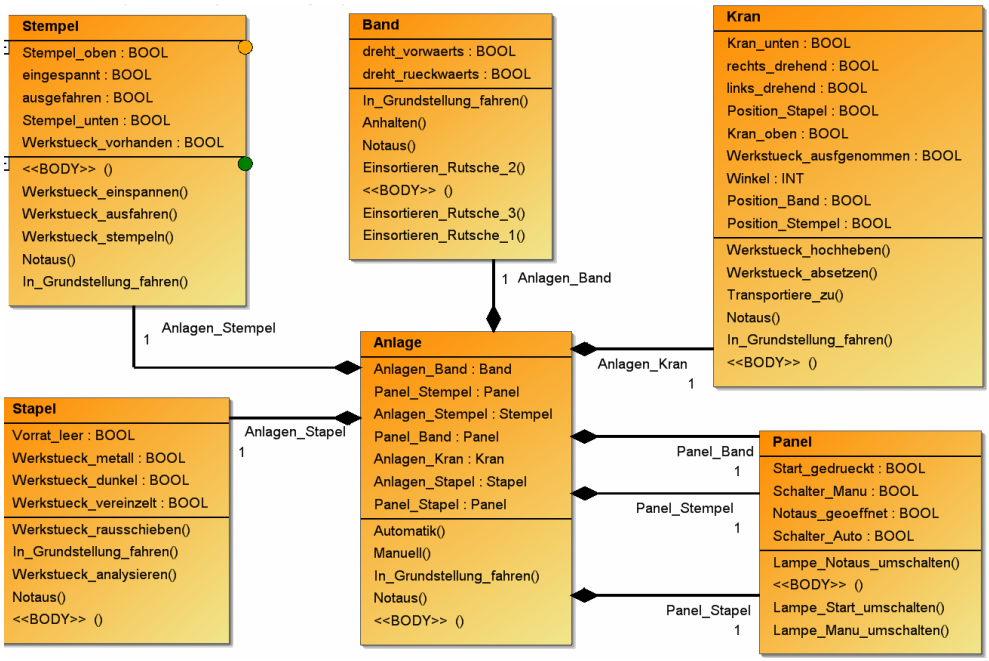


Abb. 3.9: Initiales Klassendiagramm nach Analyse der Maschinen-/Anlagenstruktur (Analyse-Modell) mit mechatronischen Klassen

Die Suche nach Subsystemen ist ein iterativer Vorgang, da solche Subsysteme in verschiedenen Granularitäten vorkommen. So kann ein Subsystem einmal ein komplexes Aggregat umfassen genauso aber einen einzelnen Aktor. Empfehlenswert ist ein Top-Down vorgehen bei dem die Suche nach mechatronischen Modulen von großen Modulen hin zu immer kleineren Untermodulen voranschreitet. Die unterste Ebene der Module wird oft von einzelnen Aktoren (Antriebe, Ventile, Zylinder) gebildet, die oberste von Betriebsartengruppen.

Die schematische Betrachtung des Anwendungsbeispiels zeigt, dass das immer wiederkehrende Element der Pneumatik-Zylinder ist (Abb. 3.10). Dieser taucht in zwei unterschiedlichen Arten auf: als monostabiler und als bistabiler Zylinder. Folglich bilden auch diese Zylinder mechatronische Klassen, die in den bereits identifizierten mechatronischen Klassen vorhanden sind. Dies ändert jedoch nicht den bisherigen Entwurf der mechatronischen Klassen. Diese behalten ihre Fähigkeiten (Methoden) und ihr Wissen über sich selber (Eigenschaften), sie greifen jedoch für die Erfüllung ihrer Methoden und für die Abfrage ihrer Eigenschaften u.U. die Methoden und Eigenschaften der unterlagerten Zylinder auf, d.h. die Ausprogrammierung der einzelnen Methoden muss entsprechend gestaltet sein.

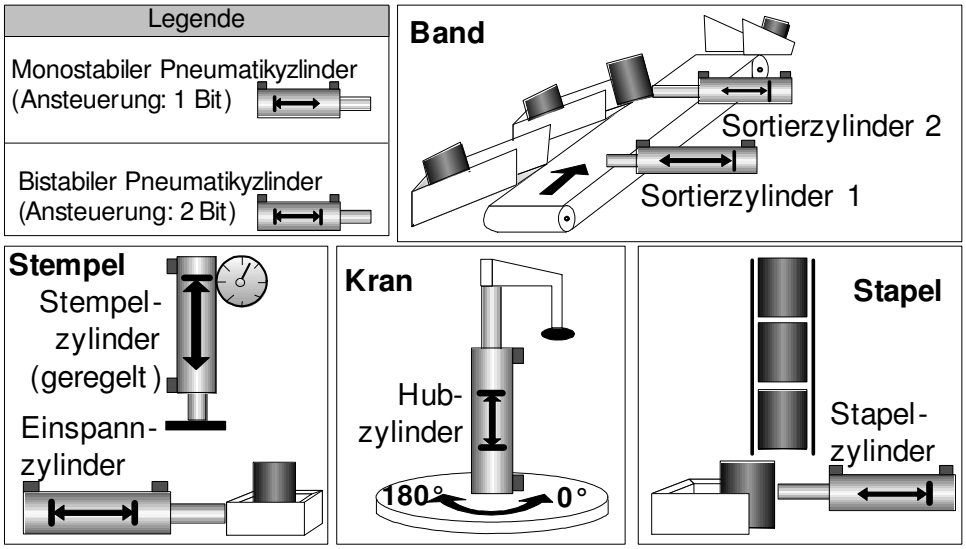


Abb. 3.10: Schematische Darstellung des Anwendungsbeispiels (Hervorhebung der Zylinder)

Abb. 3.11 zeigt ein angepasstes Klassendiagramm. Die einzelnen mechatronischen Klassen beinhalten nun bistabile und monostabile Zylinder und benennen (instanziieren) diese entsprechend ihrer jeweiligen Funktion (z.B. Einspanner, Stempel, Hubzylinder…).

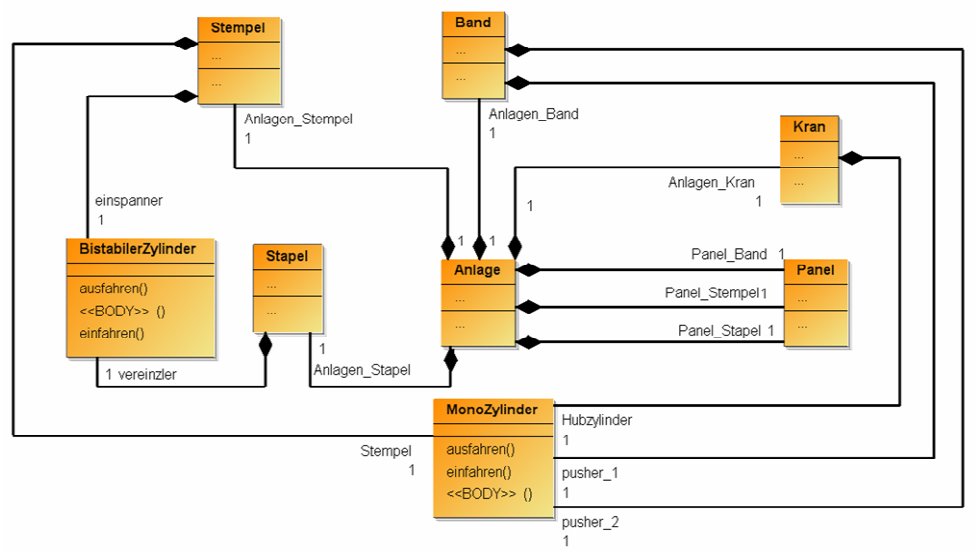


Abb. 3.11: Klassendiagramm der Analyse-Phase mit mechatronischen Sub-Klassen (die Klassen, die bereits in Abb. 3.9 vorhanden waren, sind hier zugeklappt dargestellt)

Erstellung des Software-Modells aus dem Analyse-Modell

Nach der Erstellung des Analyse-Modells kann dieses hinsichtlich softwaretechnischer Gesichtspunkte überarbeitet bzw. konkretisiert werden. Diese Überarbeitung kann z.B. eine Anpassung an firmenspezifische Software-Standards sein, die sich in Interface-Spezifikationen ausdrückt (s. Abb. 3.12, Interfaces Fehlercontroller, BetriebsartenController, PickAndPlaceEinheit) oder einer Umgestaltung von Zugehörigkeiten. Diese Umgestaltung von Zugehörigkeiten bildet keinen Gegensatz zur physikalischen Struktur, sondern ist oft Folge von der Einführung neuer Software-Objekte, die keine direkte physikalische Entsprechung haben (z.B. Abb. 3.12, Gesamtsteuerung). Solche Klassen stellen (im Kontext dieser hier vorgestellten Vorgehensweise) keine mechatronischen Klassen dar. Des Weiteren können Gemeinsamkeiten unterschiedlicher Klassen über die Bildung einer gemeinsamen Vaterklasse und anschließender Vererbung aufgelöst werden (Generalisierung) wodurch redundanter Code vermieden wird.

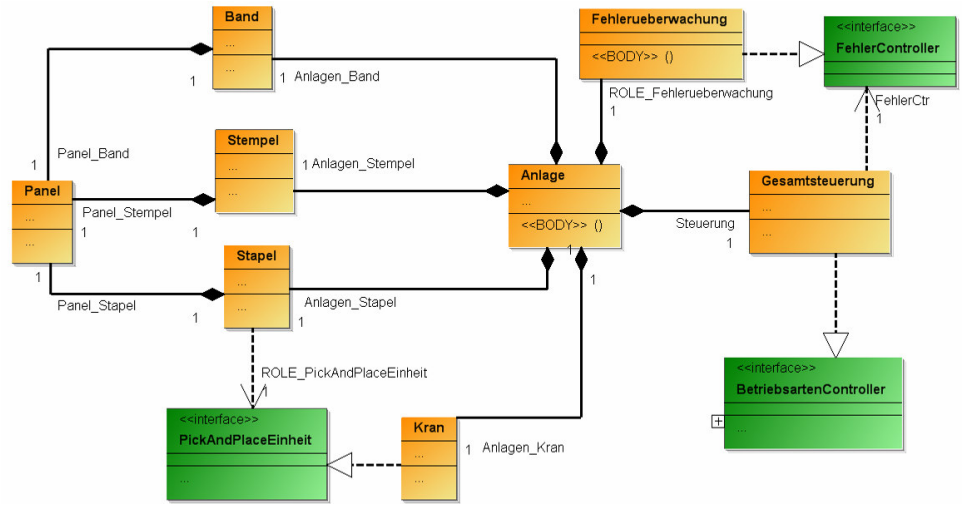


Abb. 3.12 Klassendiagramm nach Überarbeitung entsprechend software-technischen Gesichtspunkten

Die Ausprogrammierung der einzelnen Klassen, die aus den maschinenbaulichen Modulen abgeleitet wurden, kann sehr gut auf mehrere Personen aufgeteilt werden, da diese Grundfunktionen (bei einem guten Design) nur wenige Abhängigkeiten untereinander aufweisen. Starke Abhängigkeiten zwischen unterschiedlichen Klassen deuten unter Umständen auf ein unvorteilhaftes Design hin. Die konkrete Programmiersprache, in der die Ausprogrammierung der einzelnen Methoden erfolgt, kann frei gewählt werden. Auch hier kann wieder zwischen einem Analysemodell und einer Implementierung unterschieden werden. Das Analysemodell kann eine mehr oder weniger grobe Spezifikation der zu

realisierenden Funktionalität darstellen und so als Implementierungsvorlage und/oder Dokumentation dienen.

3.3 Das Zustandsdiagramm

Zustandsdiagramme (engl. Statecharts) sind im Bereich der Spezifikation und Programmierung von reaktiven Systemen weit verbreitet. Zustandsautomaten existieren in vielen unterschiedlichen Varianten. Verschiedene kommerzielle und nicht kommerzielle Werkzeuge bieten Codegeneratoren basierend auf Statecharts für unterschiedliche Zielplattformen an. Im Bereich der Steuerungsprogrammierung auf IEC 61131-3 Basis gelten bestimmte Einschränkungen, die bezüglich der Implementierung von UML-Statecharts eine Besonderheit darstellen.

Die beiden wesentlichen Faktoren sind die Ausführung auf einem zyklisch arbeitenden System und die Tatsache, dass keine Events (wie in modernen Hochsprachen und in der UML vorgesehen) zur Verfügung stehen.

Eine weitere wesentliche Anforderung an die Modellierungsumgebung für Statecharts ist die Unterstützung von Fehlerbehandlungsmechanismen, da diese einen großen Anteil an der Gesamtkomplexität der Steuerung ausmachen. Außerdem muss die Kombinierbarkeit mit anderen IEC 61131-3 Sprachelementen möglich sein.

3.3.1 Semantik der Modellierungselemente im Statechart und daraus resultierendes Zeitverhalten

Für die Modellierung von Statecharts/Zustandsautomaten stehen die in Abb. 3.13 dargestellten Modellierungselemente zur Verfügung.

Zustände (normal und zyklusintern)

Entry-Aktion

Die Entry-Aktion gehört zu den eingehenden Transitionen und sorgt für die korrekte Initialisierung des Zustandes. Die graphische Zuordnung der Entry-Aktion zum State sorgt dafür, dass nicht alle eingehenden Transitionen die vorbereitenden Anweisungen für das korrekte Betreten des States redundant aufführen müssen.

Do-Aktion/Do-Aktivität

Die Do-Aktivität wird solange ausgeführt, wie der Zustand aktiv ist. Eine Do-Aktion kann beliebige Anweisungen in Strukturiertem Text enthalten oder aber alternativ ein komplettes unterlagertes Statechart, welches im Zuge der Codegenerierung inline generiert wird. Hierdurch ist auch eine Hierarchiebildung von Statecharts möglich.

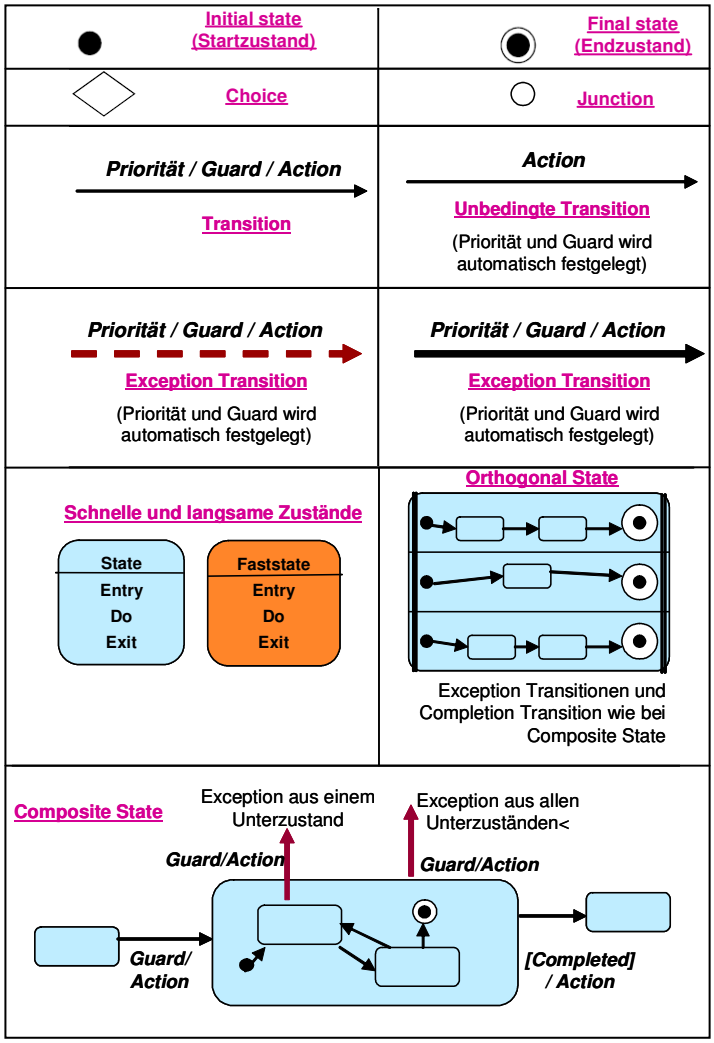


Abb. 3.13: Sprachumfang der Statecharts bei der Modellierung

Exit-Aktion

Die Exit-Aktion sorgt dafür, dass der Zustand in einem gültigen Zustand verlassen wird. Analog zur Entry-Aktion gehört die Exit-Aktion logisch zu den abgehenden Transitionen.

Unterschied zwischen zyklusinternen und normalen Zuständen

Bei zyklusinternen Zuständen wird kein Zykluswechsel ausgeführt und direkt (im gleichen Zyklus) zum nachfolgenden Zustand gewechselt. Sollte nach Beendigung der Do-Aktion eines zyklusinternen Zustandes keine abgehende Transition wahr sein, dann wird künstlich ein Zykluswechsel eingeführt und dies als „FastExecutionFault“ in die Statusvariablen des

Zustandes geschrieben. Normale Zustände ziehen einen Zykluswechsel nach sich, um die Ausgangswerte auszuschreiben.

Position des Zykluswechsels

Die Exit-Aktion und die Entry-Aktion gehören semantisch zur Transition und nur modellierungstechnisch/logisch zum Zustand[1]. Damit bilden Exit-Aktion, Transitionsaktion und Entry-Aktion eine untrennbare Ausführungsfolge. Die Kette zwischen Exit-Aktion, Transitionsaktion und Entry-Aktion darf daher nicht durch einen Zykluswechsel unterbrochen werden.

Die Entry-Aktion bereitet ihrerseits den Eintritt in einen Zustand vor und ist u.U. Voraussetzung dafür, dass eine Do-Aktion eines Zustandes auf korrekt initialisierten Variablen aufsetzt. Daher ist auch die Folge von Entry-Aktion und dem erstmaligen Betreten eines Zustandes nicht unterbrechbar. Dies entspricht auch der Codegenerierung aus Sequential Function Charts (SFC), die in Codesys V3 umgesetzt ist.[2]

Composite-States

Composite-States stellen Zustandsgruppen dar. Neben der logischen, visuellen Einteilung kann über die Bildung von Zustandsgruppen auch gemeinsames Fehlerverhalten modelliert werden. Composite-States haben keine Entry-, Do-, oder Exit-Aktion.

Orthogonale Zustände

Orthogonale Zustände modellieren parallele Abläufe. Da das Statechart tatsächlich pseudoparallel abläuft, muss das modellierte parallele Verhalten sequenzialisiert werden. In welcher Reihenfolge dies geschieht, hängt von der Einteilung der parallelen Abläufe in unabhängige Regionen (s.u.) und deren Priorität ab. Grundsätzlich wird versucht in der Reihe absteigender Regionspriorität innerhalb eines Zyklusses soweit jede Region auszuführen, wie dies die Übergangsbedingungen der Transitionen zulassen. Das heißt ein orthogonaler Zustand mit nur schnellen Zuständen (Intra Cycle States) und mit ausschließlich wahren Übergangsbedingungen wird innerhalb eines Zyklusses ausgeführt. Wenn ein orthogonaler Zustand in seinen Regionen z.B. nur normale (langsame, den Zyklus wechselnde) Zustände beinhaltet und alle (bzw. die aktuell nachfolgenden) Transitionsbedingungen in allen Regionen wahr sind, dann wird innerhalb eines SPS-Zyklusses ein (der aktuelle) langsame Zustand aus jeder Region (zyklusintern in der Reihenfolge der Regionsprioritäten) ausgeführt.

---

[1] „Entry and Exit activities are not semantically essential (the entry action could be attached to all incoming transitions)“ [RJB04, S. 333]
„the exit action could be attached to all outgoing transitions“ [RJB04, S. 345]

[2] „An entry activity is useful for performing an initialization that must be done when a state is first entered.“ [RJB04, S. 333]

Orthogonale Zustände können nicht geschachtelt werden. Die Kombination mit Composite States ist jedoch möglich. Orthogonale Zustände haben keine Entry-, Do-, oder Exit-Aktion.

Regionen in Orthogonalen Zuständen

Regionen stellen unabhängige Teilabläufe in einem orthogonalen Zustand dar. Jede Region hat innerhalb des orthogonalen Zustandes eine eindeutige Priorität, die über die interne Abarbeitungsreihenfolge entscheidet. Außerdem kann jede Region über einen beliebigen Namen verfügen. Regionen können beliebige andere Modellierungselemente enthalten (jedoch keine weiteren orthogonalen Zustände, s.o.).

Startzustand

Startzustände können auf dem Diagramm direkt, in Composite-States oder in Regionen orthogonaler Zustände verwendet werden. Auf Diagrammebene, in einem Composite-State und in Regionen orthogonaler Zustände wird jeweils genau ein Startzustand benötigt. Startzustände auf Diagrammebene stellen selber keinen echten Zustand dar, sondern markieren die erste Transition die ausgeführt werden soll. In Composite-States werden Startzustände als zyklusinterner Zustand (ohne Aktionen) umgesetzt.

Entscheidungspunkte (Choice)

Ein Entscheidungspunkt entspricht semantisch einem zyklusinternen Zustand (Intra Cycle State), der keine Entry-, Do-, Exit-Aktion haben kann. Intern wird ein Entscheidungspunkt auch wie ein solcher behandelt.

Junction Points

Junction Points stellen als Pseudozustände keine semantischen Entitäten des Statecharts dar. Vielmehr sind sie eine vereinfachte Notation für Transitionen. Durch Junction Points lassen sich mehrstufige Entscheidungen einfacher modellieren. Junction Points bilden keine echten Zustände sondern werden vor der Codegenerierung in mehrere Transitionen aufgelöst (vgl. Abb. 3.14). So werden alle auf einen Junction Point eingehenden Transitionen mit allen ausgehenden Transitionen multipliziert, d.h. ihre Transitionsbedingungen werden verundet, ihre Aktionen werden nacheinander ausgeführt und ihre Priorität ergibt sich aus der Priorität der eingehenden und der ausgehenden Transitionspriorität.

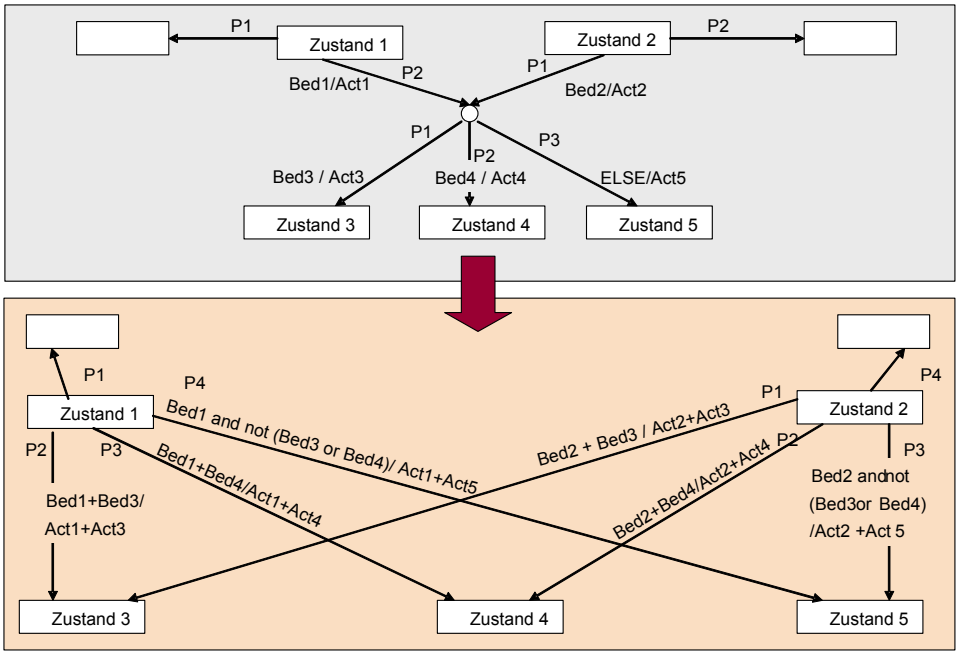


Abb. 3.14: Auflösung von Junction Points (Kreuzungspunkten)

Endzustand

Endzustände werden in verschiedenen Kontexten (auf Diagrammebene, in Composite-States oder in Regionen orthogonaler Zustände) verwendet.

Semantik auf Diagrammebene

Wird ein Endzustand erreicht, ist das Statechart beendet. Es wird bei einem erneuten Aufruf auch nicht wieder ausgeführt, sondern gibt die Ausführungskontrolle unmittelbar wieder zurück. Das Statechart kann über das Setzen der boolschen Variable „ReInit“ wieder reinitialisiert werden und startet dann beim nächsten Aufruf wieder von vorne. Wenn ein nicht endender Ablauf beabsichtigt ist, sollte dies über die Rückführung von Transitionen realisiert werden.

Semantik in Composite-States

Ein Endzustand in einem Composite State führt nicht zu einer Beendigung des Ablaufs. Auch ist danach keine Reinitialisierung notwendig. Ein Endzustand in einem Composite State markiert den Aussprungpunkt aus dem Composite State. Intern werden die End- und Anfangszustände in Composite-States wie zyklusinterne Zustände behandelt.

Semantik in Orthogonalen Zuständen

In orthogonalen Zuständen führen die Endzustände in den einzelnen orthogonalen Regionen die unabhängigen Abläufe zusammen. Erst wenn alle Regionen ihren Endzustand erreicht haben, kann der gesamte orthogonale Zustand verlasst werden. Beim nächsten Aufruf beginnt die Abarbeitung des orthogonalen Zustandes wieder von vorne. Es ist keine Reinitialisierung notwendig.

Transitionskanten

Es gibt unterschiedliche Transitionstypen. Transitionen können allgemein über eine Bedingung (Guard) und eine Aktion (Action) verfügen. Wenn die Bedingung an der Transition wahr ist, kann diese überschritten und der vorhergehende Zustand verlassen werden. Beim Übergang wird dann eine evtl. vorhandene Transitionsaktion einmal ausgeführt.

Normale Transition

Die Normale Transition verfügt über eine Aktion und eine Bedingung. Die Bedingung muss angegeben werden. Die Angabe einer Aktion ist optional.

Completion Transition

Die Completion Transition ist unbedingt, d.h. sie kann überschritten werden, wenn das vorgehende Statechart-Element fertig bearbeitet wurde (z.B. orthogonale Zustände) oder wenn das vorhergehende Element ein Start-Zustand ist. Sie besitzt demnach keine Bedingung. Die Angabe einer Aktion ist optional.

Exception Transition

Exception Transitionen werden rot, gestrichelt dargestellt. Sie verfügen über eine Übergangsbedingung und eine Aktion. Wenn eine Exception Transition überschritten werden kann, wird dies ohne Zykluswechsel durchgeführt.

Pseudo-Exception Transition

Manchmal ist es übersichtlicher, anstelle eines Endzustandes in einem Composite-State, der mit sehr vielen Transitionen verbunden ist, eine Transition am Rande des Composite-State zu zeichnen, unter deren Bedingung der gesamte Composite-State verlassen wird. Dies geschieht analog zur Exception Transition, aber ohne die inhaltliche Bedeutung, dass dies ein Fehlverhalten darstellt und wird daher als Pseudo-Exception Transition bezeichnet. Diese Transition hat (im Vergleich zur Exception Transition) keinen Einfluss auf das zyklische Ausführungsverhalten. Die Pseudo-Excpetion Transitionen werden als dicke schwarze Pfeile dargestellt.

Prioritäten

Wenn mehrere Transitionen vom gleichen Startelement abgehen, werden automatisch, veränderbare Transitionsprioritäten angegeben, die darüber entscheiden, welche Transition überschritten wird, wenn mehrere Bedingungen gleichzeitig wahr sind.

3.3.2 Syntaxdefinition der Statecharts

Die hier vorgestellten Statecharts repräsentieren eine grafische Programmiersprache. Wesentliche Voraussetzung für eine Codegenerierung aus Statecharts ist neben der Semantikspezifikation die Definition einer eindeutigen Syntax. Bei der Syntaxspezifikation müssen alle erreichbaren Kombinationen der möglichen Modellierungselemente auf ihre Gültigkeit hin überprüft werden. Erst wenn eine gültige Syntax vorliegt, kann der Prozess der Codegenerierung sicher angestoßen werden. Die Betrachtung der Kombinatorik zwischen den Modellierungselementen und den möglichen Beziehungen zwischen diesen (enthalten, verbunden sein, schneiden) führt schnell auf eine sehr große Anzahl zu betrachtender Fälle. Um diese Anzahl möglichst einzuschränken, wurden einige komplexere Modellierungselemente auf andere einfachere Elemente zurückgeführt. Diese Rückführung wurde für Composite-States, Exception Transitionen bzw. Pseudo-Exception Transitionen (vgl.Abb. 3.16) und Junction Points (vgl. Abb. 3.14) durchgeführt.

Composite-States werden vollständig aufgelöst (vgl. Abb. 3.15). Hierzu werden die eingehenden Transitionen auf den Rand des Composite-States auf den obligatorischen Startzustand innerhalb des Composite-States umgebogen. Der Start-Zustand ist ein normaler zyklusungebundener Zustand ohne Entry-/Do-/Exit-Aktion. Die Auflösung der Final-States bzw. der Completion-Transition erfolgt analog hierzu. An Stelle des Final-States wird ein zyklusungebundener Zustand ohne Entry-/Do-/Exit-Aktion eingeführt. Die Completion Transition wird dann durch eine gewöhnliche Transition ersetzt, die von diesem Zustand ausgeht.

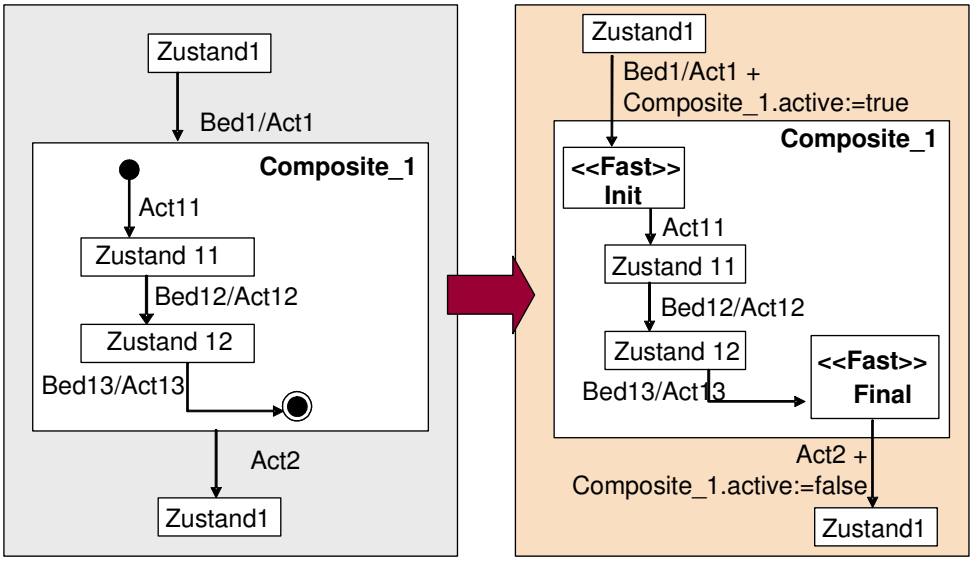


Abb. 3.15 : Auflösung von Composite States

Eng mit der Auflösung der Composite-States ist auch die Auflösung der Exception-Transition und Pseudo-Exception-Transitionen verbunden (vgl. Abb. 3.16). Wird eine solche Transition vom Rand eines Composite-States gezeichnet, so wird im Zuge der Auflösung, diese Transition mit jedem in dem Composite-State liegenden Element „multipliziert“. Die vorherigen Transitionsprioritäten werden dahingehend angepasst, dass die neuen Transitionen, die Infolge der Auflösung der Exception-Kanten entstanden sind, immer höherprior sind als die Transitionen, die sich ursprünglich innerhalb des Composite-States befanden untereinander jedoch die vorher definierte Rangfolge einhalten.

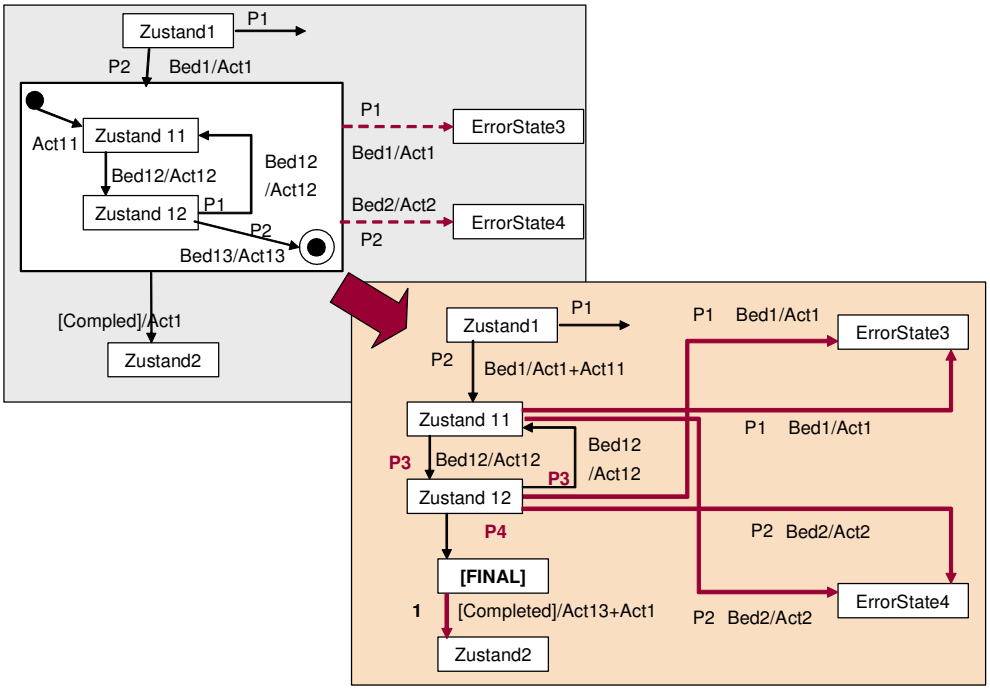


Abb. 3.16 : Auflösung von Fehlertransitionen

Syntaxdefinition mit reduziertem Sprachumfang

Nach den oben beschriebenen Auflösungsvorgängen, hat sich der Umfang der Modellierungselemente reduziert.

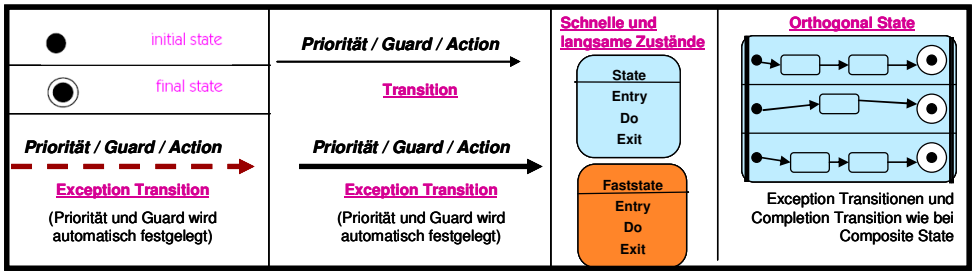


Abb. *3.17 : Reduzierter Sprachumfang der Statecharts für die Codegenerierung*

Für diese reduzierte Menge müssen Syntaxregeln definiert werden. Hierzu werden nachfolgend zuerst solche direkten, paarweisen Beziehungen zwischen den Elementen definiert, die einer gültigen Syntax entsprechen. Diese gültigen Beziehungen sind in Abb. 3.18 dargestellt.

A\ B	slow state/ faststate	composite-state	orthogonal-state	Transition	completion Transition	Exception/ Pseudo Exception Transition	startstate	finalstate	region	Choice
slow state/ faststate				A --> 1*B						
compositestate	A <>- 0*B	A <>- 0*B	A <>- 0*B	A --> 0*B, A <>- 0*B	A --> 01B, A <>- 0*B	A--> 0*B A<>- 0*B	'A <>- 1B	'A <>- 01B		A <>- 0*B
orthogonalstate				A --> 0*B	A --> 01B	'A--> 0*B			A <>- 2*B	
transition	A --> 01B	A --> 01B , A -/> 01B	A --> 01B , A -/> 01B					A --> 01B		A --> 1*B
completion Transition	A --> 01B	A --> 01B , A -/>01 B	A --> 01B , A -/> 01B					A --> 01B		A --> 01B
Exception/ Pseudo Exception Transition	A --> 01B	A --> 01B , A -/>01 B	A --> 01B , A -/> 01B					A --> 01B		A --> 01B
startstate					A --> 1B					
finalstate										
orhogonal region	A <>- 0*B	A <>- 0*B	A <>- 0*B	A <>- 0*B	A <>- 1*B	A<>- 0*B	A <>- 1B	A <>- 01B		A <>- 0*B

Legende: X --> Y: Element Y folgt auf Element X

X <>- Y: Element X beinhaltet Element Y

X -/> Y: Element X schneidet Element Y (z.B. Transition über Region)

Kardinalitäten: Element A hat immer die Kardinalität 1.

Die Kardinalität von Element B variiert: 1=genau ein, 01=kein oder ein, 0*=kein bis beliebig viele, 1*=ein bis beliebig viele, n=genau n, n*=mindestens n maximal beliebig viele

Abb. 3.18: Definition der gültigen direkten Folgebeziehungen für das Statechart

Aus der Definition der direkten Folgebeziehungen lassen sich des Weiteren kombinatorisch die gültigen indirekten Folgebeziehungen ableiten. Diese sind in Abb. 3.19 dargestellt.

Mögliche Kombinationen aus den direkten Folgebeziehungen								
slowstate/faststate	1	-->	1*	Transition	1	-->	1	slowstate/faststate
Composite State	1	-->	0*	Transition	1	-->	1	slowstate/faststate
Orthogonalestate	1	-->	0*	Transition	1	-->	1	slowstate/faststate
Choice	1	-->	1*	Transition	1	-->	1	slowstate/faststate
Junction	1	-->	1*	Transition	1	-->	1	slowstate/faststate
slowstate/faststate	1	-->	1*	Transition	1	-->	1	compositestate
Composite State	1	-->	0*	Transition	1	-->	1	compositestate
Orthogonalestate	1	-->	0*	Transition	1	-->	1	compositestate
Choice	1	-->	1*	Transition	1	-->	1	compositestate
Junction	1	-->	1*	Transition	1	-->	1	compositestate
slowstate/faststate	1	-->	1*	Transition	1	-->	1	orthogonal State
Composite State	1	-->	0*	Transition	1	-->	1	orthogonal State
Orthogonalestate	1	-->	0*	Transition	1	-->	1	orthogonal State
Choice	1	-->	1*	Transition	1	-->	1	orthogonal State
Junction	1	-->	1*	Transition	1	-->	1	orthogonal State
slowstate/faststate	1	-->	1*	Transition	1	-->	1	Finalstate
Composite State	1	-->	0*	Transition	1	-->	1	Finalstate
Orthogonalestate	1	-->	0*	Transition	1	-->	1	Finalstate
Choice	1	-->	1*	Transition	1	-->	1	Finalstate
Junction	1	-->	1*	Transition	1	-->	1	Finalstate
slowstate/faststate	1	-->	1*	Transition	1	-->	1	Choice
Composite State	1	-->	0*	Transition	1	-->	1	Choice
Orthogonalestate	1	-->	0*	Transition	1	-->	1	Choice
Choice	1	-->	1*	Transition	1	-->	1	Choice
Junction	1	-->	1*	Transition	1	-->	1	Choice
slowstate/faststate	1	-->	1*	Transition	1	-->	1	Junction
Composite State	1	-->	0*	Transition	1	-->	1	Junction
Orthogonalestate	1	-->	0*	Transition	1	-->	1	Junction
Choice	1	-->	1*	Transition	1	-->	1	Junction
Composite State	1	-->	1	Completion Transition	1	-->	1	SlowState/FastState
Orthogonalestate	1	-->	1	Completion Transition	1	-->	1	SlowState/FastState
StartState	1	-->	1	Completion Transition	1	-->	1	SlowState/FastState
Composite State	1	-->	1	Completion Transition	1	-->	1	Composite State
Orthogonalestate	1	-->	1	Completion Transition	1	-->	1	Composite State
StartState	1	-->	1	Completion Transition	1	-->	1	Composite State

Abb. 3.19 : Gültige indirekte Folgebeziehungen (wird fortgesetzt in Abb. 3.20)

Composite State	1	-->	1	Completion Transition	1	-->	1	Orthogonal State
Orthogonalestate	1	-->	1	Completion Transition	1	-->	1	Orthogonal State
StartState	1	-->	1	Completion Transition	1	-->	1	Orthogonal State
Composite State	1	-->	1	Completion Transition	1	-->	1	Final State
Orthogonalestate	1	-->	1	Completion Transition	1	-->	1	Final State
StartState	1	-->	1	Completion Transition	1	-->	1	Final State
Composite State	1	-->	1	Completion Transition	1	-->	1	Choice
Orthogonalestate	1	-->	1	Completion Transition	1	-->	1	Choice
StartState	1	-->	1	Completion Transition	1	-->	1	Choice
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	SlowState/FastState
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	SlowState/FastState
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Composite State
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Composite State
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Orthogonal State
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Orthogonal State
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Final State
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Final State
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Choice
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Choice
Composite State	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Junction
Orthogonalestate	1	-->	0*	(Pseudo-)Exception-Trans.	1	-->	1	Junction

Abb. 3.20: Gültige indirekte Folgebeziehungen (Fortsetzung von Abb. 3.19)

Schnitte

Ein Schnitt liegt vor, wenn Start- und Endelement einer Transition oder einer Completion Transition in unterschiedlichen Regionen liegen.

Die theoretisch möglichen Schnittbeziehungen der Transitionen und Composite-States wurden für den praktischen Gebrauch weiter eingeschränkt. Die getroffenen Einschränkungen stellen sich in den folgenden vier Regeln dar:

1.) Orthogonale Zustände dürfen nicht geschachtelt werden, d.h. es darf keinen orthogonalen Zustand als übergeordnetes Element eines anderen orthogonalen Zustandes geben.

2.) Eine Transition / Completion Transition darf immer nur einen Composite-State schneiden, d.h. es ist nur ein Sprung in die direkt unterlagerte oder in die direkt übergeordnete Ebene möglich (s. Abb. 3.21).

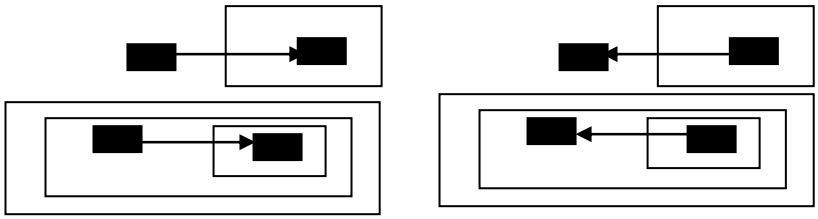


Abb. 3.21 :Gültige Schnitte mit Composite-States

3.) Es kann nicht in einen orthogonalen Zustand hereingesprungen werden, nur heraus.

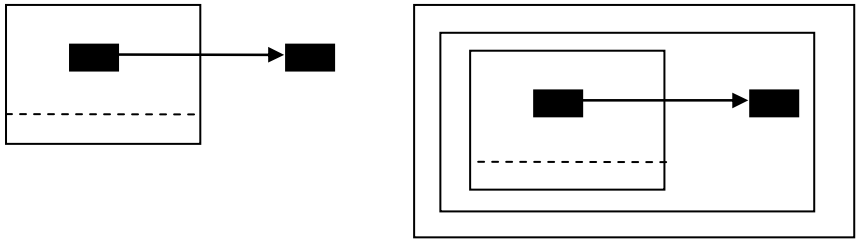


Abb. 3.22: Gültige Schnitte mit orthogonalen Zuständen

4.) Es darf keine Transition oder Completion Transition geben, die zwei Elemente verbindet, die in unterschiedlichen orthogonalen Regionen sind.

Anwendungsszenarien für das Statechart

Das Statechart kann in vier unterschiedlichen Arten verwendet werden.

Dokumentation/Analyse

Statecharts können als Analyse-Diagramme erstellt werden. Dann wird aus ihnen kein Code erzeugt. Sie dienen dann zur Formulierung von Anforderungen oder groben Ideen in einer frühen Entwicklungsphase. Wenn ein Dokumentations-Statechart soweit ausgereift ist, dass eine Implementierung darauf aufbauend vorgenommen werden soll, kann dieses automatisch in ein Implementierungs-Statechart umkopiert werden (es wird eine Kopie mit gleichem Layout angelegt).

Als Programmiersprache

Statecharts können als Programmiersprache für Funktions-Blöcke, Methoden und Aktionen eingesetzt werden.

Debugging in Online-Ansicht

Wird der Steuerungscode auf die Steuerung geladen und stellt der Anwender eine Online-Verbindung zur Steuerung her, schaltet das Statechart von der Programmier-Ansicht in eine

Online-Ansicht. In der Online-Ansicht werden die aktiven Zustände farbig hervorgehoben. Außerdem ist das Setzen von Breakpoints jeweils vor die Ausführung der

- Entry-Aktionen,
- Do-Aktionen,
- Exit-Aktionen,
- Guard-Prüfungen von Transitionen,
- Aktionen von Transition

möglich.

Debugging Offline mit Trace

Um die korrekte Funktion zu prüfen bzw. um Fehler im Zustandsautomaten zu finden, steht ein Trace zur Verfügung der optional eingeschaltet werden kann. Mit dem Trace können nahezu beliebig viele Zustandswechsel zur Laufzeit aufgezeichnet und nachher offline in beliebiger Zeitdehnung oder –stauchung abgespielt werden (Trace-Player). Hierbei werden im Statechart die durchlaufenen Wege angezeigt. Zudem können zu einem Statechart beliebig viele Traces aufgezeichnet und als Dokumentation gespeichert werden.

Die Bedienoberfläche des Statechart Editors

Der Statechart-Editor unterstützt den Anwender bei der Modellierung in der Art dass grobe Modellierungsfehler ausgeschlossen sind (z.B. Transitionen auf einen Startzustand). Verstöße gegen Syntaxregeln werden direkt im Meldungsfenster der Programmierumgebung dargestellt. Das Statechart kann nur Code generieren, wenn keine Syntaxfehler vorliegen. Bei Schachtelungssituationen (z.B. States in einem Composite-State) wird grafisch hervorgehoben, wenn sich ein Element innerhalb eines anderen befindet. Des Weiteren bietet der Editor Funktionalitäten, die die Arbeit beschleunigen (z.B. einfaches Umhängen von Transitionen). Zeitgesteuerte Zustandswechsel können über den Ausdruck „after(Zeitangabe)“ in Transitionsbedingungen modelliert werden (z.B. after(300ms)). Für das Statechart stehen die gleichen Funktionen hinsichtlich Stereotypen und Sichten zur Verfügung wie beim Klassendiagramm (vgl. 3.2.3). *Abb. 3.23* zeigt den Einsatz der verschiedenen Statechart-Elemente im Zusammenhang.

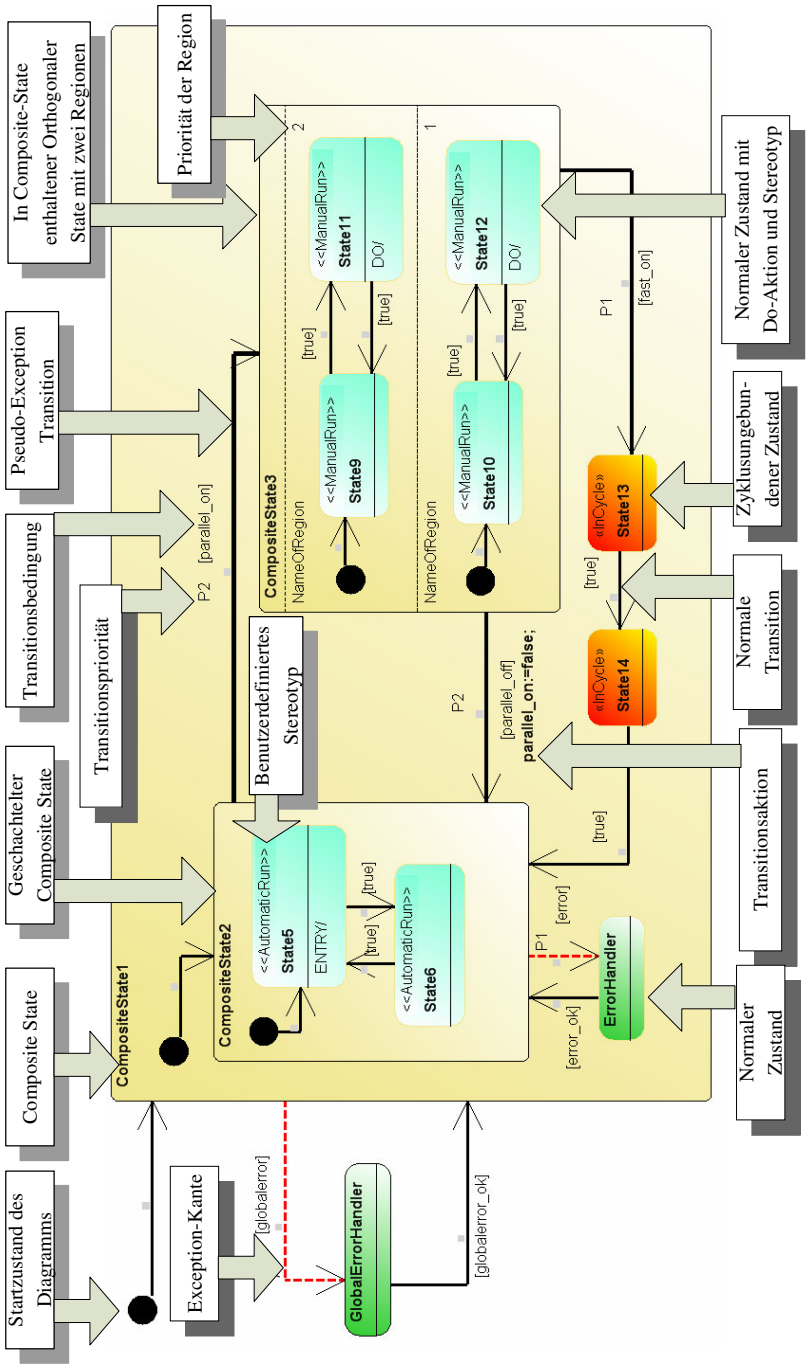


Abb. 3.23: Modellierungselemente des Statechart-Editors in der Anwendung

3.3.3 Anwendungsbeispiel bistabiler Pneumatik-Zylinder

Die grundlegende Idee hinter der Verwendung von Zustandsdiagrammen, ist das betrachtete System in seinen unterschiedlichen möglichen Zuständen und Zustandsübergängen abzubilden. Im Kontext mechatronischer Systeme sind die Zustände oft auch physikalisch erfassbar. So sind die statischen Zustände eines zugrunde liegenden Systems, die sich oft auch in einer bestimmten Belegung von Sensorwerten ausdrücken auch Zustände in einem Zustandsdiagramm. Als Beispiel sei ein bistabiler Pneumatikzylinder angenommen. Der Pneumatikzylinder verfügt über zwei Endlagen-Sensoren, die auf die Variablen IX_eingefahren und IX_ausgefahren gemappt sind. Aus der logischen Kombination der Sensorwerte resultieren mehr als die beiden statischen Zustände „Ausgefahren“ und „Eingefahren“. Auch die Bewegungen „Einfahrend“ und „Ausfahrend“ lassen sich einer Sensorkombination zuordnen, wenn der vorhergehende Zustand bekannt ist. Tritt die Sensorkombination (0/0) auf, nachdem der Zylinder vorher ausgefahren war, fährt dieser offensichtlich ein und andersherum. Solche Übergänge, die durch eine veränderte Variablenkonfiguration gekennzeichnet sind und ohne weitere Steuerungsanweisungen nach einer gewissen Dauer zu einem statischen Zustand führen, können als transiente Zustände[3] des Zustandsdiagramms betrachtet werden. Anhand dieses Beispiels soll nachfolgend eine mögliche Vorgehensweise bei der Programmierung von mechatronischen Klassen bzw. Modulen mit Hilfe von Statecharts vorgestellt werden.

1. **Aufstellen einer Zustandstabelle** (s. Tab. 3.1) mit Betrachtung aller möglichen Kombinationen der Sensorwerte des betrachteten mechatronischen Moduls (Analogwerte sind in disjunkte Wertebereiche aufzuteilen). Hierbei ist darauf zu achten, dass das Modul nicht zu groß gewählt wird, da sonst der Zustandsraum unüberschaubar wird.[4]
2. **Ableitung der statischen Zustände, der transienten Zustände und der ungültigen Variablenkombinationen,** die auf einen Fehler hindeuten, aus der Zustandstabelle.
3. **Abschätzung einer maximalen Dauer für transiente Zustände.** Transiente Zustände sollten sich selber beenden, daher kann eine maximale Zeit abgeschätzt werden, nach der der Zustand verlassen sein sollte (t_{max}). Wenn der transiente Zustand nicht innerhalb dieser Zeit verlassen wurde, kann eine Fehlerbehandlung greifen.

[3] Der Begriff „transienter Zustand“ wird in der Automaten-Theorie als auch in verschiedenen Bereichen der Automatisierungstechnik und Naturwissenschaften mit unterschiedlichen Bedeutungen verwendet. Hier soll das Adjektiv transient auf die Eigenschaft hinweisen, dass der Zustand zeitabhängig wieder verlassen wird.

[4] Die Anzahl der Zustände wächst exponentiell 2^x, wobei x die Anzahl der binären Variablen ist. Hinzu kommen die Wertebereiche eventueller Analogwerte.

Tab. 3.1: Zustandstabelle Pneumatikzylinder

IX_eingefahren	IX_ausgefahren	Zustand
0	0	• Einfahrend (transient, t_{max}=500ms), • Ausfahrend (transient, t_{max}=500ms), • Stehend zwischen den Endlagen (statisch)
0	1	Ausgefahren (statisch)
1	0	Eingefahren (statisch)
1	1	Sensorfehler (Fehler)

4. **Beschreibung aller Zustandsübergänge in der mechatronischen Klasse in Form eines geschlossenen Analyse-Statecharts.** Ziel im Sinne der objektorientierten Programmierung ist es einen Satz von Methoden für die Ansteuerung des mechatronischen Moduls zur Verfügung zu stellen. Daher reicht es nicht ein großes, geschlossenes Statechart als Programmcode für ein mechatronisches Modul zur erzeugen. Auf der anderen Seite ist es schwierig die zustandsbasierte Analyse eines mechatronischen Moduls von vorne herein auf mehrere teils überlappende Zustandsdiagramme aufzuteilen. Ein sinnvoller Ansatz ist es daher, zunächst im Sinne eines Analyse-Statecharts ein geschlossenes Zustandsdiagramm mit allen das System beschreibenden Zuständen und Zustandsübergängen zu modellieren. Im Zuge der Umsetzung dieses Anlayse-Modells in eine entsprechende Implementierung können dann aus der geschlossenen Darstellung die jeweiligen Methodenimplementierungen als einzelne Statecharts extrahiert werden. Statecharts für die einzelnen Methoden beinhalten dann eine Untermenge der Zustände und Zustandsübergänge des Analyse-Modells.

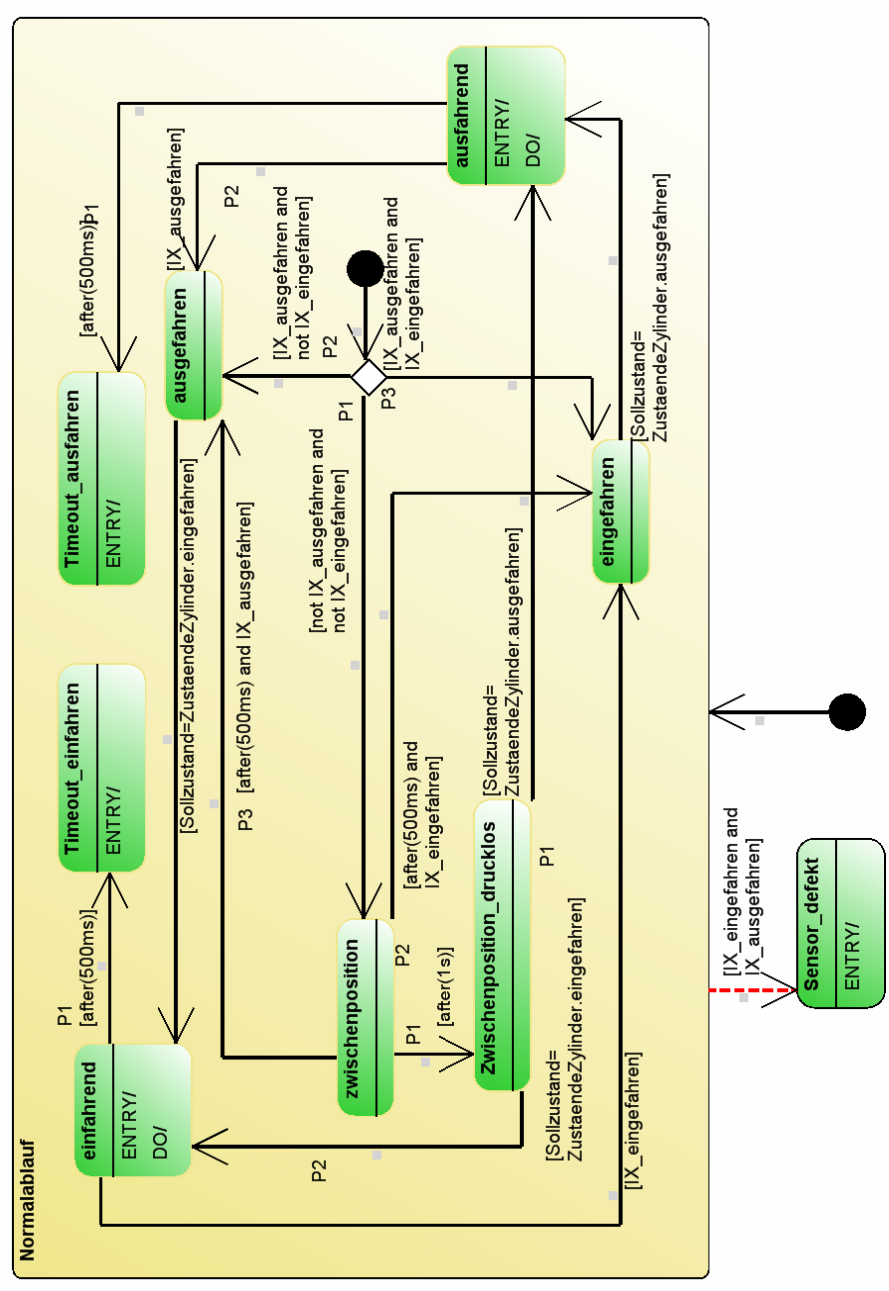


Abb. 3.24: Statechart-Analyse-Modell des Zylinders. Um von den unterschiedlichen Methoden zu abstrahieren, die hieraus gebildet werden sollen, wurde angenommen, dass diesem Statechart ein Sollzustand übergeben wird und dass es die Aufgabe des Statecharts sei, das mechatronische Modul vom aktuellen Zustand in diesen Sollzustand zu überführen.

5. **Modellierung der Fehlerfälle.** Anhand der Informationen aus der Zustandstabelle (s.o.) können einige Standard-Fehlerfälle identifiziert werden. So kann z.B. von jedem transienten Zustand aus eine zusätzliche Transition gezeichnet werden, die den Fehler der Zeitüberschreitung (t_{max}) behandelt (s. Abb. 3.24, Zustand „Timeout_einfahren“). Außerdem kann, wenn unmögliche Sensorkombinationen identifiziert wurden, der gesamte Normalablauf in einem Composite-State zusammengefasst werden, von dessen Rand eine Exception-Transition abgeht, die die unmögliche Sensorkombination als Bedingung trägt (s. Abb. 3.24: Statechart-Analyse-Modell des Zylinders. Um von den unterschiedlichen Methoden zu abstrahieren, die hieraus gebildet werden sollen, wurde angenommen, dass diesem Statechart ein Sollzustand übergeben wird und dass es die Aufgabe des Statecharts sei, das mechatronische Modul vom aktuellen Zustand in diesen Sollzustand zu überführen.Abb. 3.24, Zustand „Sensor_defekt“).

6. **Feststellung des initialen Zustandes und Nutzung von Systemzuständen als Vorbedingungen für Methoden.** Bei der zustandsbasierten Programmierung hängt die ausgeführte Aktion von der aktuellen Belegung der Eingangsvariablen und dem aktuellen Zustand in dem Zustandsautomaten ab. Die korrekte Bestimmung des Zustandes ist beim erstmaligen Aufruf eines Zustandsautomaten (z.B. beim Hochlaufen der Maschine/Anlage) nicht immer eindeutig möglich, da zu diesem Zeitpunkt keine Informationen zum vorausgehenden Zustand zur Verfügung stehen und nur aufgrund der aktuellen Sensorbelegung entschieden werden muss, welcher Zustand vorliegt (Tab. 3.1 zeigt, dass eine Sensorkombination in unterschiedlichen Zuständen auftreten kann). Daher ist es sinnvoll für jede mechatronische Klasse eine Initialisierungsmethode zu implementieren, die einen definierten Anfangszustand aktiv herstellt. Da je nach Verwendung der mechatronischen Klasse ein anderer Anfangszustand eingenommen werden muss (vgl Abb. 3.10: beim Stempelmodul, sollte der Einspannzylinder zu Beginn ausgefahren sein, der Stempelzylinder jedoch eingefahren), ist es hilfreich den jeweilig gewünschten Initialzustand bereits bei der Instanziierung der mechatronischen Klasse mit zu übergeben.[5] Im Zuge der Gesamtinitialisierung der Maschine/Anlage muss diese Initialisierungsmethode aufgerufen werden.

[5] In CoDeSys V3 ist dies mit der FB_Init Methode möglich.

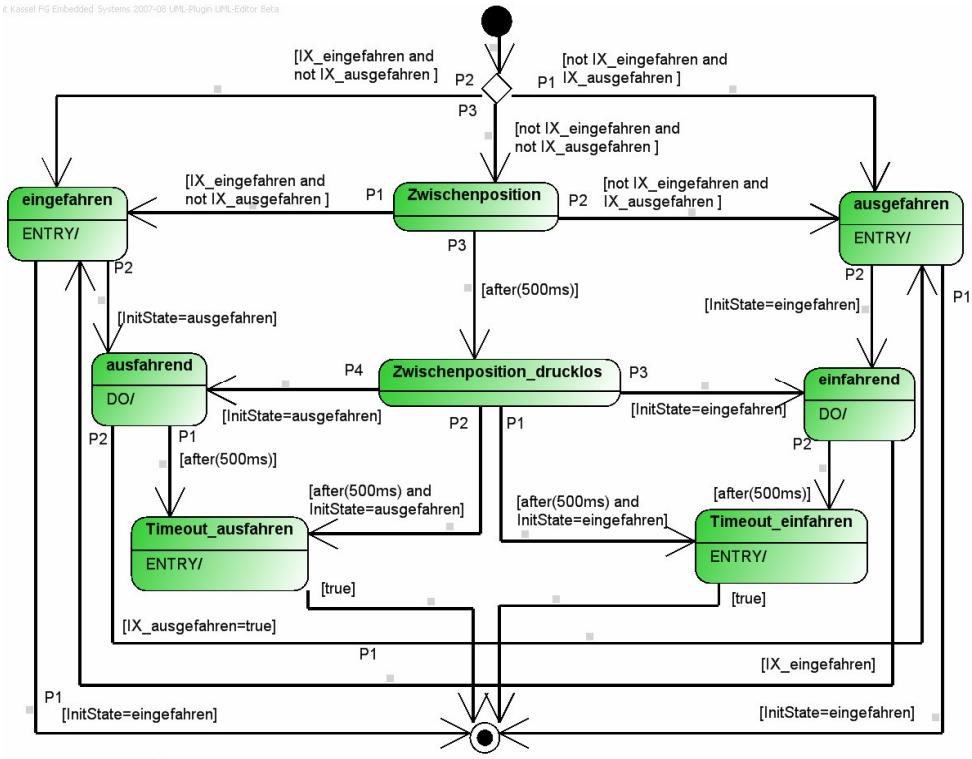


Abb. 3.25: Init-Methode des Zylinders (in den Entry-Aktionen wird jeweils die Variable „Istzustand“ der Klasse auf den gleichnamigen Zustand gesetzt)

7. **Verwendung des Modulzustandes als Vorbedingung in Methoden.** Nachdem die Initialisierungsmethode durchgeführt wurde, befindet sich das mechatronische Modul in einem bekannten Zustand. Durch die Programmierung mit Zustandsautomaten wird das mechatronische Modul von einem bekannten Zustand in einen anderen bekannten Zustand versetzt. Es kann also ab der Durchführung der Initialisierungsmethode bestimmt werden, in welchem Zustand sich das mechatronische Modul befindet. Methoden der mechatronischen Klasse, die deren Zustand verändern, können den Systemzustand zu Beginn ihres Ablaufs im Sinne einer Vorbedingung abfragen (z.B. könnte die Methode „ausfahren()“ des Zylinders die Vorbedingung stellen, dass sich der Zylinder im Zustand „Eingefahren“ oder im Zustand „Ausgefahren“ befindet, vgl. Abb. 3.26).

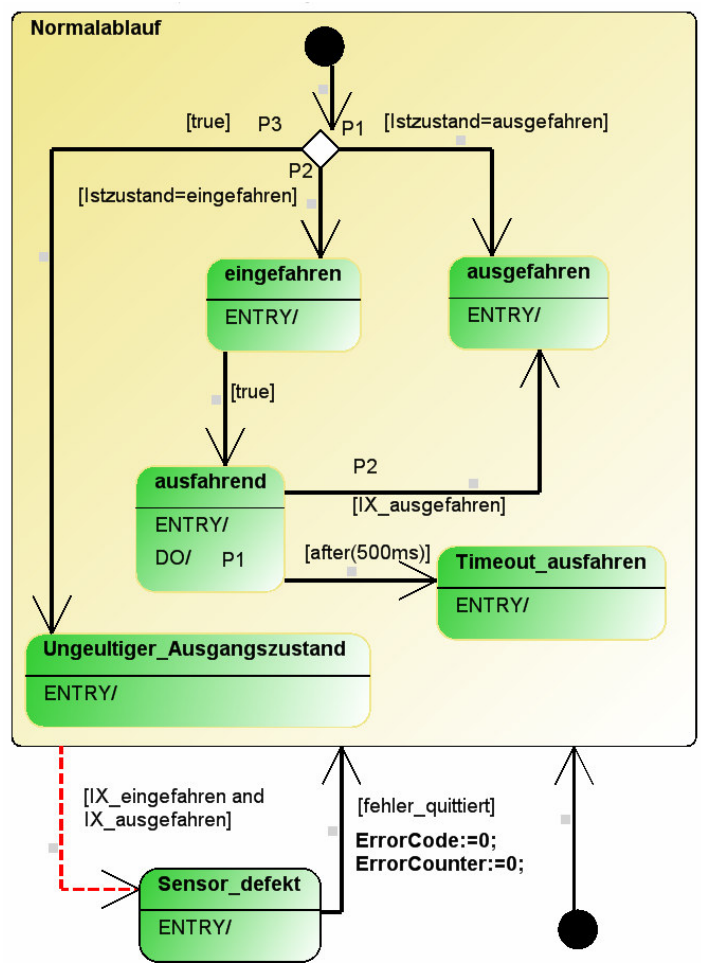


Abb. 3.26 : Ausfahren-Methode des Zylinders (In den Entry-Aktionen wird jeweils die Variable "Istzustand" der Klasse auf den gleichnamigen Zustand gesetzt. In den Zuständen "ausgefahren", "Ungueltiger_Ausgangszustand", und „Timeout_ausfahren“ wird am Ende der Entry-Aktionen zusätzlich das Statechart Reinitialisiert.)

8. **Speicherung des aktuellen Zustandes in der Klasse.** Der aktuelle Zustand, in dem sich ein mechatronisches Modul (z.B. eine mechatronische Klasse aus 3.2.4) befindet, ist eine Eigenschaft des Moduls bzw. der Klasse. Daher sollte deren aktueller Zustand in einer Variablen abgespeichert und gelesen werden können (z.B. als Output-Variable, die innerhalb und außerhalb des Funktionsblocks gelesen und innerhalb des Funktionsblocks geschrieben werden kann).

```
FUNCTION_BLOCK Zylinder
VAR_OUTPUT
      Istzustand : States_Zylinder;
END_VAR
…
```

Für die Lesbarkeit ist es hierfür sinnvoll einen Enumerations-Datentyp mit den Zustandsnamen anzulegen.

```
TYPE States_Zylinder :
(einfahrend, ausfahrend, zwischenposition, sensor_defekt,
ausgefahren, eingefahren, fehler);
END_TYPE
```

Jeder Zustand im Zustandsdiagramm jeder Methode sollte in seiner Entry-Aktion die Variable „Istzustand“ entsprechend beschreiben.

```
Istzustand:=States_Zylinder.einfahrend;
```

3.4 Das Aktivitätsdiagramm

Durch die objektbasierte Struktur verteilen sich die meisten Abläufe über mehrere Objekte. Diese lassen sich in UML Aktivitätsdiagrammen gut abbilden. Aktivitätsdiagramme stellen sequentielle und parallele Abläufe auf einfache Weise dar. Ihr dynamisches Verhalten und ihre Semantik sind an die der Petri-Netze [Pet62] angelehnt. Aktivitätsdiagramme sollen eine einfache, intuitive Modellierung für Technologen ermöglichen, daher wird in der Diagrammdarstellung von der dahinterliegenden Steuerungssoftware abstrahiert. Ziel ist es eine einfache, grafische Programmiersprache zu bieten, die den Technologen ohne Kenntnisse der Steuerungsprogrammierung in die Lage versetzt, seine Abläufe eindeutig zu formulieren. Daher wurde bei der Umsetzung der Aktivitätsdiagramme die technologische Sichtweise und nicht diejenige des Programmierers in den Vordergrund gerückt. Aktivitätsdiagramme werden in zwei unterschiedlichen Rollen verwendet.

1.) Entwickler von Aktivitäten (Programmierer)

Der Programmierer entwickelt Bibliotheken mit Klassen und Methoden, die die steuerungstechnischen Grundfunktionen realisieren. Zusätzlich definiert er um diese Methoden der Klassen Aktivitäten, die nachher von Technologen einfach verwendet werden können und legt diese Aktivitätsbibliotheken mit der Klassenstruktur in einer entsprechenden Bibliothek ab.

2.) Anwender von Aktivitäten (Technologe)

Der Anwender von Aktivitäten greift auf die vom Entwickler definierten Aktivitätsbibliotheken zu. Er kann die vorher definierten Aktivitäten aus der Bibliothek in das aktuelle Diagramm einfügen und durch die Verkettung und Parametrierung der einzelnen Aktivitäten den technologischen Ablauf festlegen.

Nachfolgend werden die einzelnen Elemente der Aktivitätsdiagramme vorgestellt (Abb. 3.29 zeigt diese im Zusammenhang).

3.4.1 Aktivitäten

Eine Aktivität ist nicht direkt eine Methode bzw. ein Methodenaufruf (so wie diese in der Anwendungsentwicklung umgesetzt werden), sondern ein Prozess mit einer realen Dauer (im Sinne des technischen Systems). Diese reale Dauer (z.B. Zeit von der Aktivierung einer Achse bis zur Erreichung der entsprechenden Endlage) ergibt sich, indem zusätzlich zu einem parametrierten Methodenaufruf das Eintreten einer dazugehörigen Endbedingung abgeprüft wird.

Grundlage für die Verwendung von Aktivitätsdiagrammen ist daher die Einführung eines neuen Programmierkonstrukts, der Aktivität. Eine Aktivität bildet ein Tupel aus einer Methode und einer Endbedingung. Darüber hinaus können zur Fehlerüberwachung Erwartungswerte hinsichtlich einer maximalen und minimalen Dauer der Aktivität angegeben werden sowie Reaktionen, falls diese Erwartungswerte über- bzw. unterschritten werden. Eine Untermenge der Methodenparameter kann als Datenpin der Aktivität definiert

werden und es kann ein (für den Technologen) lesbarer Name für diese Aktivität vergeben werden. Diese Eigenschaften einer Aktivität können über eine entsprechende Eingabemaske definiert werden (s. Abb. 3.27).

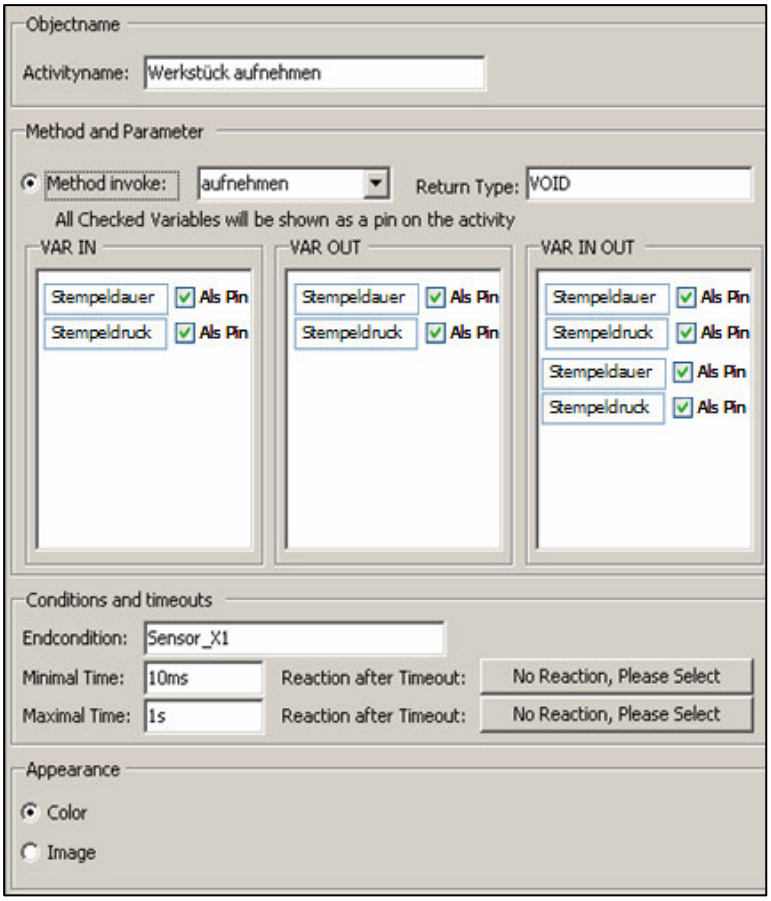


Abb. 3.27: Definitionsfenster für Aktivitäten um Methoden

Aktivitätsbibliotheken

Die Erstellung von Aktivitäten erfolgt nachdem die Methoden der einzelnen Grundmodule ausprogrammiert wurden, d.h. die Aktivitäten werden aufbauend auf den Klassen mit ihren Methoden definiert. Dazu ist es möglich, sich für eine Klasse oder eine Menge zusammengehöriger Klassen, eine Aktivitätsbibliothek automatisch erzeugen zu lassen (s. Abb. 3.28). Manuell müssen dann noch für jede Aktivität die Eintragungen, wie in Abb. 3.27 zu sehen, vorgenommen werden. Die Aktivitätsbibliothek bildet die Grundlage für die Programmierung mit Aktivitätsdiagrammen.

Abb. 3.28 Aktivitätsbibliotheken

Ein Aktivitätsdiagramm wird (wenn es als Programmiersprache eingesetzt wird) immer einer Methode bzw. einer POU zugeordnet (s. Abb. 3.30, Methode Automatik). Aus der Zuordnung des Aktivitätsdiagramms zu einer bestimmten Methode und damit auch einer bestimmten Klasse (die diese Methode enthält) wird festgelegt, welche Aktivitätsbibliotheken in diesem Aktivitätsdiagramm eingesetzt werden können. Der Technologe erhält automatisch die korrekte Menge an Aktivitäten, die er für die Realisierung der Funktionen einsetzen kann, in Form einer Werkzeugleiste eingeblendet.

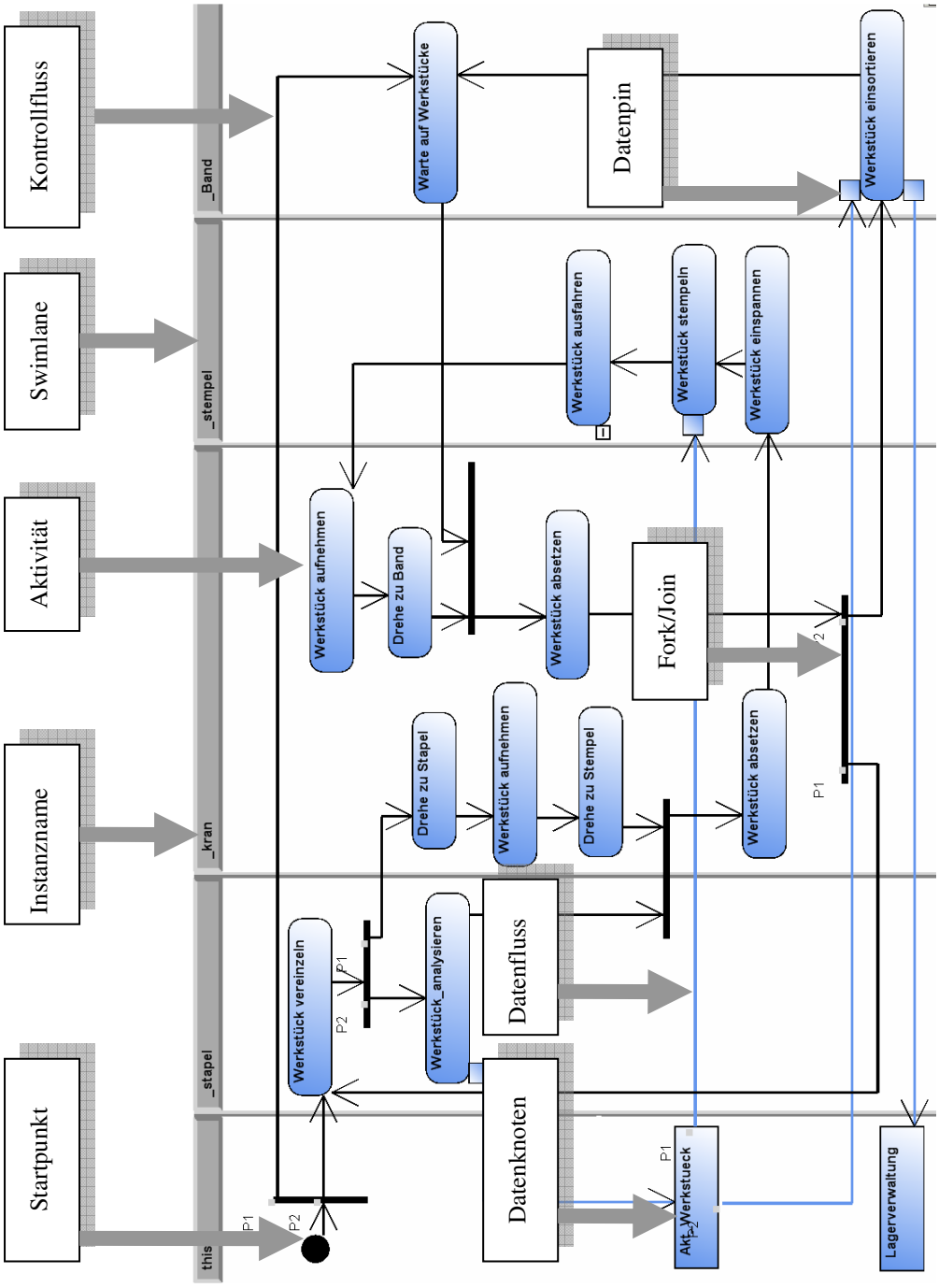


Abb. 3.29: Beispiel Aktivitätsdiagramm (Stapel-Stempel-Anlage, Automatik-Methode)

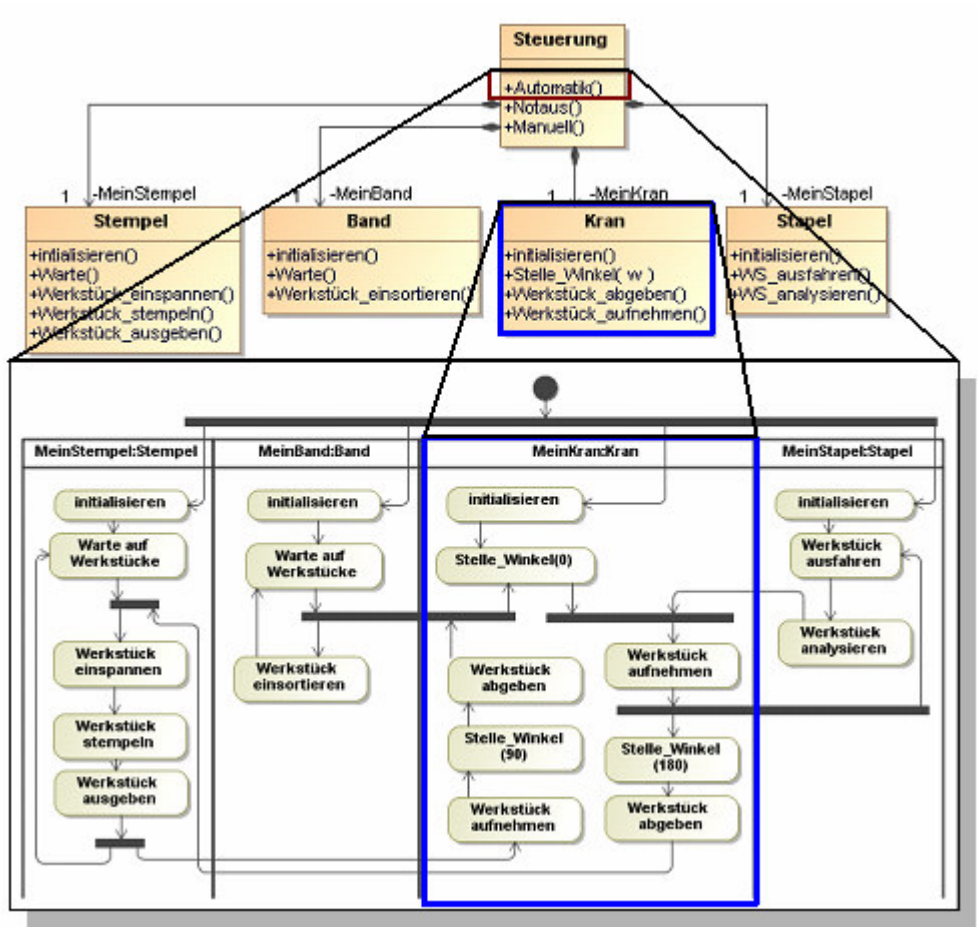


Abb. 3.30 Zuordnung Aktivitätsdiagramm zu Methode und Swimlane zu Instanz

3.4.2 Swimlanes

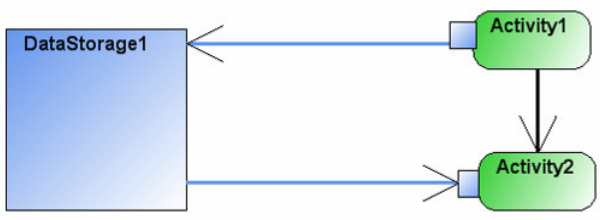
Die Swimlanes (vertikale Spalten) in Aktivitätsdiagrammen stellen Verantwortlichkeitsbereiche dar. Im Rahmen des Forschungsprojektes werden diese Verantwortlichkeitsbereiche als Namensraum für Instanzen interpretiert, d.h. alle Aktivitäten in einer Swimlane gehören zu der Instanz einer Klasse, welche in der Kopfzeile der Swimlane notiert ist. In Abb. 3.30 wird bspw. die Aktivität Initialisieren in allen Swimlanes aufgerufen. Jeder Aufruf dieser Aktivität löst die Initialisierung in der Instanz aus, die in der Kopfzeile steht.

3.4.3 Kontrollflüsse

Der Kontrollfluss im Aktivitätsdiagramm wird durch schwarze Pfeile zwischen den Diagrammknoten (Aktivitäten, Startpunkt, Fork/Join-Balken, etc.) modelliert. Der Kontrollfluss beginnt bei dem Startpunkt und bildet Sequenzen von Aktivitäten, die nacheinander aufgerufen werden. Mittels Fork/Join-Balken kann ein sequentieller Kontrollfluss in mehrere parallel laufende aufgespalten werden. Teilmengen aus den parallel ablaufenden Kontrollfluss-Strängen können wieder zusammengeführt oder weiter aufgespalten werden. Durch Entscheidungsknoten können Alternativ-Verzweigungen eingefügt werden. Von einem Aktivitätsknoten kann immer nur ein Kontrollflusspfeil abgehen. Es können jedoch mehrere Kontrollflüsse in einem Aktivitätsknoten enden. Der Ablauf des Kontrollflusses arbeitet nach dem Prinzip des Markenspiels von Petri-Netzen.

3.4.4 Datenflüsse

Aktivitäten bzw. die Methoden, die darin gekapselt sind, benötigen in der Regel einen Satz von Parametern. Diese Parameter erhalten sie über eingehende Datenpins. Ausgabevariablen und Rückgabewerte, die nach der Ausführung einer Aktivität zur Verfügung stehen, können (über Ausgabge-Pins) in explizite Datenspeicher abgelegt werden (s. Abb. 3.31) oder direkt an die Eingabe-Pins anderer Aktivitäten übergeben werden. Hierbei wird implizit bei der Codegenerierung eine Variable für die Zwischenspeicherung angelegt.




```
VAR
  DataStorage1 : INT;
END_VAR

Activity1(OutPar=>DataStorage1);
Activity2(ParIn:=DataStorage1);
```

Abb. 3.31 : Datenfluss im Aktivitätsdiagramm

Im Vergleich zum Kontrollfluss sind Datenflüsse farbig dargestellt, wobei jeder Datentyp eine eigene Farbe zur besseren Unterscheidung erhält. Es können auch komplexe Datentypen als Datenflüsse verwendet werden. Diese können analog zur Programmiersprache CFC in ihre Unterelemente aufgespalten bzw. zusammengesetzt werden.

3.4.5 Anwendungsszenarien für Aktivitätsdiagramme

Während Zustandsdiagramme vermehrt dafür eingesetzt werden, gekapseltes Verhalten von wiederverwendbaren Klassen zu beschreiben und zu programmieren, beziehen sich Aktivitätsdiagramme auf das Zusammenspiel mehrerer Klassen. Aktivitätsdiagramme modellieren Abläufe aus einer äußeren Sichtweise auf die beteiligten Klassen heraus. Daher haben Aktivitätsdiagramme viele objektübergreifende Querbezüge in einer Software-Struktur und lassen sich auf vielfältige Weise in den Software-Engineering Ablauf integrieren. Im Folgenden werden unterschiedliche Einsatzszenarien für Aktivitätsdiagramme betrachtet.

Entwicklung eines neuen technologischen Ablaufs, teilweise auf Basis von Standard-Maschinenmodulen nach dem Maschinen/Anlagen-Design

Zunächst kann die mechatronische Struktur der Maschine/Anlage in Form eines oder mehrerer Klassendiagramme abgebildet werden (s. 3.2.4). Die Klassen sollten bereits die Attribute/Eigenschaften und Methoden enthalten und die wesentlichen Enthalten-Beziehungen sollten im Klassendiagramm definiert worden sein.

Die Klasse, die die Gesamtmaschine/-anlage darstellt, enthält typischerweise die Methoden, die die technologischen Schritte enthält (z.B. Automatik, Manuell, Notaus, Anlauf etc.). Für jede dieser Methoden kann ein Analyse-Aktivitätsdiagramm angelegt werden. In diesen Aktivitätsdiagrammen können bereits die Swimlanes, die sich aus den Enthalten-Beziehungen ergeben, hinzugefügt werden. Innerhalb dieser können die Aktivitäten, die für diese Swimlanes zulässig sind, angeordnet und mit anderen Aktivitäten verknüpft werden. Durch die Definition von Datenflüssen und neuen Aktivitäten wird der Entwurf schrittweise konkretisiert.

Entwicklung eines neuen technologischen Ablaufs auf Basis von Standard-Maschinenmodulen vor dem Maschinen/Anlagen-Design

Zunächst stellt der Technologe ein oder mehrere Analyse-Aktivitätsdiagramme als Vorlage für deren software-technische Implementierung aus vorhandenen Aktivitätsbibliotheken zusammen und identifiziert dabei durch Swimlanes und durch die Verwendung von Aktivitäten automatisch die zu verwendenden Klassen. Methoden werden durch Aktivitäten automatisch ausgewählt und parametriert. In einem iterativen Prozess kann das Analyse-

Modell des Technologen dazu genutzt werden, die notwendigen mechatronischen Module mit ihren Klassen zu identifizieren.[6] In enger Absprache mit dem Programmierer kann dann das Analyse-Modell in einen software-technisch, fachlichen Entwurf überführt werden. Hierfür kann das Analyse-Modell in ein Implementierungsdiagramm umkopiert werden (das Analyse-Modell bleibt für die Dokumentation und spätere Nachverfolgbarkeit erhalten).

Neuentwicklung in einer frühen Entwurfs-Phase (Verhaltensbasiert)

Zunächst kann in Form eines Analyse-Aktivitätsdiagramms der Automatik-Ablauf der Gesamtanlage/-maschine grob modelliert werden. Einzelne Aktivitäten, hinter denen sich ein komplexeres Verhalten verbirgt können als einzelnes Aktivitätsdiagramm beschrieben werden. Hieraus ergibt sich eine hierarchische Struktur von Aktivitätsdiagrammen. In diesem Top-Down-Ansatz können die einzelnen Aktivitätsdiagramme immer weiter verfeinert werden. Aus den unterschiedlichen Diagramm-Elementen der Aktivitätsdiagramme, lassen sich Rückschlüsse auf die benötigte Software-Struktur ziehen. Swimlanes deuten auf Instanzen hin. Diese Instanzen benötigen entsprechende Klassen mit den Methoden, die die Grundlage für die im Aktivitätsdiagramm aufgerufenen Aktivitäten bilden. Darüber hinaus lassen sich aus den Datenknoten Variablen in den Klassen ableiten. Die Schachtelungsstruktur der Aktivitätsdiagramme deutet auf Kompositionsbeziehungen in der Software-Struktur hin. Das Analyse-Modell der Software-Struktur und Verhaltensbeschreibung kann im Zuge des Software-technischen Entwurfs verfeinert werden. Die Methoden der Aktivitäten müssen ausprogrammiert werden und können den in dem Aktivitätsdiagrammen verwendeten Aktivitäten zugeordnet werden. Wiederverwendbare Methoden bzw. Klassen können in Form von Aktivitätsbibliotheken für zukünftige Verwendungen abgelegt werden.

3.5 Zusammenfassung und Ausblick

In diesem Beitrag wurden die, im Rahmen des Projektes "Steigerung der Effizienz und Qualität im Software-Engineering der Automatisierungstechnik für die Domäne des Maschinen- und Anlagenbaus" entstandenen technischen Ergebnisse zusammenfassend vorgestellt. Entstanden sind Editoren für Klassendiagramme, Statecharts und Aktivitätsdiagramme. Das Klassendiagramm bietet eine völlig neuartige Sicht auf Steuerungssoftware. Statecharts und Aktivitätsdiagramme bieten Verbesserungen im Vergleich zu existierenden grafischen Programmiersprachen der IEC 61131-3 bzw. verhalten sich zu diesen komplementär, wie **Tab. 3.2** zusammenfassend darstellt.

[6] Dieser Beitrag ist auf die Software-Entwicklung im Engineering fokussiert. Bei einer Neuentwicklung folgt in der Regel nicht direkt der Software-Entwurf nach der Definition des grundsätzlichen Technologie-Konzepts, sondern oft ein maschinenbaulicher Entwurf. Die Rückschlüsse, die hier auf die Steuerungssoftware bezogen sind, lassen sich in Teilen auch auf den maschinenbaulichen Entwurf übertragen, da mechatronische Klassen auch ein maschinenbauliches Ebenbild besitzen.

Tab. 3.2:*Vergleich Statechart, Aktivitätsdiagramm, SFC und CFC (CoDeSys V3)*

Kriterium	Statechart	Aktivitätsdiagramm	SFC	CFC
Alternative Pfade	+ (Choices, Junction Points, Transitionen)	+ (Entscheidungs-knoten)	+ (Alternativ-Verzweigung)	~ (EN/ENO)
Orthogonale Abläufe	+ (Orthogonale Zustände mit Regionen)	+ (Petri-Netz Semantik)	+ (Parallel-verzweigung)	~ (EN/ENO)
Fehler-konzepte	+ (Exception-Kanten von Composite-States)	+ (Unterbrechungs-bereiche)	-	-
Nutzung für Methoden-Impl.	+	+	-	+
Zyklus-wechsel steuerbar	+ (frei wählbar durch "schnelle" Zustände)	-	- (Zykluswechsel nach Step-Wechsel)	- (gesamter CFC in einem Zyklus)
Frei-graphische Anordnung	+	+	-	+
Datenflüsse	-	+	-	+
Auflösung von Schalt-konflikten	+ (explizit, deterministisch durch Prioritäten an Transitionen)	+ (Petrinetz-Semantik)	~ (implizit über graphische Anordnung)	~ (EN/ENO)
Modellierung Objektbezug	+ (Modellierung der Objektzustände)	+ (Objekt-referenzierung über Swimlanes)	-	-
Modellierung Kontrollfluss	+	+	+	~ (EN/ENO)
Mehrere Sichten	+ (Stereotypenbasiertes Sichtenkonzept)	+ (Stereotypenbasiertes Sichtenkonzept)	~ (Detail-Ausblendung über Makros)	-

Erste Versuchsreihen mit Berufsschülern, zur Anwendung der Statecharts wurden durchgeführt und werden zurzeit ausgewertet, weitere Versuchsreihen mit Experten, Studierenden von Hochschulen, Fachhochschulen und berufsbildenden Schulen sollen folgen um die Eignung für die unterschiedlichen Zielgruppen zu belegen. Die Funktionsfähigkeit der Editoren insbesondere der Codegenerierung aus den Statecharts wurde bisher für ein kleines Prozessmodell gezeigt. Diese praktische Erprobung der Editoren und Codegeneratoren soll anhand einer wesentlich komplexeren Laboranlage und anhand von realen Anwendungen weiter vertieft werden. Neben der Evaluation der

bisherigen Ergebnisse sollen die vorhandenen Editoren weiter vorangetrieben werden mit dem Ziel diese baldmöglichst als industriell einsetzbare Ergänzung zur bisherigen Steuerungsprogrammierung anbieten zu können.

3.6 Referenzen

[OMG04] Object Management Group: Unified Modeling Language, Infrastructure, V2.1.2, Online Verfügbar http://www.omg.org/docs/formal/07-11-04.pdf

[ISO05] Object Management Group: Unified Modeling Language, V1.4.2, ISO/IEC 19501, online verfügbar, http://www.omg.org/cgi-bin/doc?formal/05-04-01.

[MDA09] http://www.omg.org/mda/.

[Pet62] Petri, C.A.: Kommunikation mit Automaten, Dissertation, Universität Bonn, 1962.

[RJB04] Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Second Edition, Addison-Wesley, Boston, 2005.

[WWV08] Witsch, D.; Wannagat, A.; Vogel-Heuser, B.: Entwurf wieder verwendbarer Steuerungssoftware mit Objektorientierung und UML. In: atp, Heft 5/08, Oldenbourg-Industrieverlag, München, 2008, Seite 54-60.

Danksagung
Wir danken der Stiftung Industrieforschung sowie 3S-Software, Beckhoff, ELAU, SIG Combibloc und teamtechnik für die Unterstützung der Arbeiten.

4 Objektorientierung in der Automatisierungstechnik - Sinn und Nutzen anhand eines konkreten Beispiels

Dr. J. Papenfort, Beckhoff

Die Zeichen der Zeit in der Automatisierungstechnik sind gestellt. Objektorientierte Programmierung ist das Thema für die nächsten Jahre. Wie bei so manchem Hype stellt sich natürlich die Frage ob Objektorientierung in der Automatisierungstechnik überhaupt etwas bringt. Was sind die Vorteile für den Programmierer? Welche Vorteile hat der Endkunde? Kann man mit der Einführung von objektorientierten Verfahren die Qualität und Zuverlässigkeit von Software verbessern? Am einfachen Beispiel einer Waschmaschine sollen die Vorteile der Objektorientierung gezeigt werden.

4.1 Die Aufgabe

Eine Waschmaschine soll automatisiert werden. Die Funktion ist einfach zu beschreiben. Nach dem Befüllen der Maschine und dem Schließen der Tür soll durch die Betätigung eines Knopfes ein bestimmtes Programm ausgeführt werden. Durch den Startknopf wird dann die Anlage inbetriebgesetzt. Abhängig vom Programm werden bestimmte Sequenzen ausgeführt.

Um die Aufgabe komplexer und realitätsnäher zu formulieren, gibt es eine einfache Maschine mit zwei Programmen und die komplexe Luxuswaschmaschine mit noch einem Energiesparprogramm. Die Trommel muss natürlich mit unterschiedlichen Drehzahlen je nach Programmpunkt bewegt werden. In bestimmten Programmsequenzen muss die Heizung eingeschaltet werden, um die passende Wassertemperatur einzustellen. Zusätzlich gilt es Zu- und Abwasser passend zu steuern.

Alles in allem eine überschaubare und sicherlich mit wenig Aufwand erstellbare Applikation. Zum schnellen Test dient eine einfache Visualisierung mit integrierter Simulation.

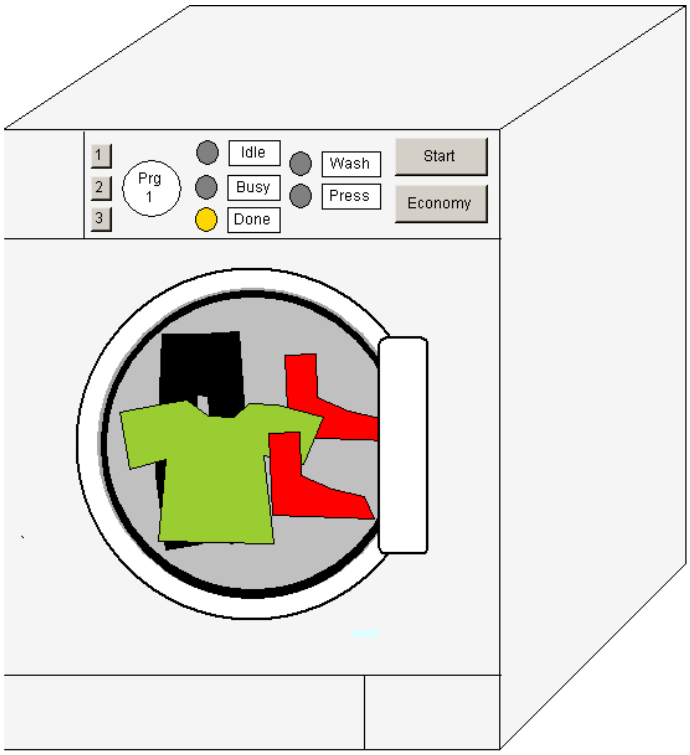


Abb. 4.1: Die Waschmaschine

4.2 Der funktionale Ansatz

Entweder Top-Down oder Bottom-Up kann man mit den Mitteln der IEC61131-3 diese einfache Maschine programmieren. Nähern wir uns dem Problem per Bottom-Up dann könnte man mit der Realisierung der einzelnen Aggregate starten. Einfach auszumachen sind die Aggregate für Heizung und Trommelbewegung. Hier können zwei FBs erstellt werden. Danach sollen die Sequenzen für die einzelnen Programme erstellt werden. In dem SPS-Programm müssen an den einzelnen Positionen des Waschprogramms die Funktionsbausteine für Heizung und Trommelbewegung aufgerufen werden. Die Implementierung dieser einfachen Applikation ist schnell gemacht. Da es kein standardisiertes Vorgehen bei der Erstellung der Software gibt, werden n Programmierer n verschiedene Varianten an Programmcode erzeugen. Soll Jahre nach der Erstellung der Waschmaschinensoftware eine Änderung oder Erweiterung gemacht werden, wird sich der dann zuständige Programmierer erst in die bestehende Software einarbeiten müssen. Eine vorgegebene Programmierkonvention wird sicherlich die Lesbarkeit der Software erhöhen. Der persönliche Programmierstil wird aber immer vorhanden sein. Es ist auch die Frage, ob die Software schon für alle Eventualitäten, die noch kommen werden, gerüstet ist. Vielleicht werden in Zukunft neue Programme entwickelt, neue Regler notwendig und

weitere Sensoren eingesetzt. Die Modifikation und die Erweiterung von klassisch geschriebener Software ist immer ein aufwändiges Thema.

4.3 Der objektorientierte Ansatz

Bereits vor vielen Jahren hat die Objektorientierung in der Programmierung Einzug gehalten. Die Vorteile dieser Art der Programmierung haben sich nach und nach in allen Bereichen der Softwareerstellung etabliert. Viele verschiedene Programmiersprachen unterstützen objektorientierte Programmierung. Allen voran C++, Java und die .NET Sprache C#. In der Welt der Automatisierungstechnik hat diese Technik bisher nur schwer Zugang gefunden. Zum Einen liegt das an der Ausbildung der SPS Programmierer, zum Anderen aber auch an den Programmiersprachen. Die IEC61131-3 hat einige Dinge aus der objektorientierten Programmierung übernommen. Funktionbausteindefinitionen können wie Klassen aufgefasst werden, Instanzen von Funktionsbausteinen sind dann die entsprechenden Objekte. Methoden, Vererbung und Interfaces sind aber der IEC61131-3 fremd. Erst in jüngster Vergangenheit haben erste Implementierungen zu einer Erweiterung der IEC61131-3 geführt. Diese Erweiterung – die demnächst auch in einer 3rd Edition der IEC61131-3 normiert werden soll – definiert einige wenige neue Schlüsselworte. Mit diesen neuen Schlüsselworten können Methoden ausgedrückt, Vererbung genutzt und Interfaces definiert werden.

Damit steht der Nutzung von objektorientierten Ansätzen nichts mehr im Wege. Was fehlt ist die einfache – möglichst grafische Möglichkeit zum Design und zur Analyse komplexer Software. Einerseits muss natürlich die Möglichkeit einer Strukturanalyse gegeben sein und andererseits auch die einfache Beschreibung des Programmverhaltens. Hierfür wird in der Welt der C++ und C# Programmierer die Unified Modeling Language (UML) genutzt. Bei der UML werden verschiedene Strukturdiagramme und verschiedene Verhaltensdiagramme definiert. Zu den gebräuchlichsten und anschaulichsten Strukturdiagrammen zählt das Klassendiagramm. Bei den Verhaltensdiagrammen haben sich das bekannte Ablaufdiagramm (StateCharts) und das etwas weniger bekannte Aktivitätendiagramm durchgesetzt.

Kann nun die Anwendung der UML und die Programmierung mit objektorientierten Spracherweiterung das Leben der Programmierer einfacher machen? Am Beispiel der Waschmaschine soll gezeigt werden, dass Objektorientierung, sinnvoll eingesetzt, die Software in der Automatisierungstechnik positiv verändern kann.

4.4 Klassenhierachie und Interfaces

Die Waschmaschine fasst als zentrale Klasse die Maschinenkomponenten zusammen. Die Steuerung der verschiedenen Programme übernimmt die Programmsteuerung. Diese ist wiederum eine Klasse. Die Wascheinheit – auch als Klasse ausgeführt – enthält Unterklassen für die wichtigsten und einfach zu erkennenden Komponenten.

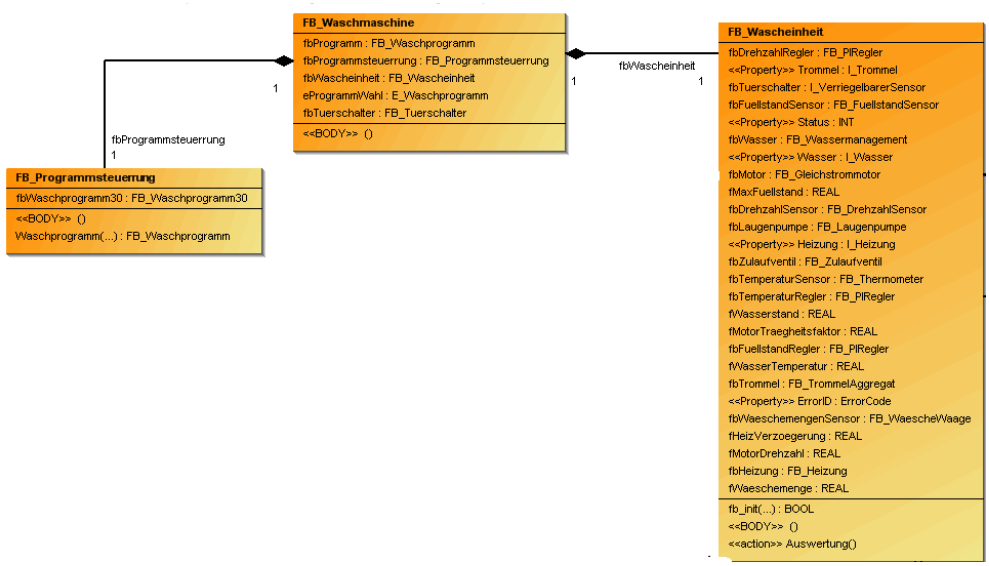


Abb. 4.2: Klassenhierarchie der Waschmaschine

Temperatur-, Drehzahl- und Füllstandsregelungen sollen aus Gründen der Wiederverwendung eines bereits bewährten Reglers alle die gleiche Reglerklasse nutzen. Auch die Sensoren und Aktoren werden alle durch eigene Klassen repräsentiert. Einige – wie z.B. der Türschalter - gehören zu der Klasse der verriegelbaren Sensoren. Um allen dieser Klasse zugehörigen Schaltern eine gleiche Aufrufschnittstelle zu geben, wird hier ein Interface eingeführt. Das Interface ‚I_VerriegelbarerSensor' ist für alle Klassen, die dieses Interface implementieren, gleich.

Auch für die Aktoren Trommel und Wasser werden Interfaces definiert. In der Definition der Klasse ‚Wascheinheit' sind alle physikalischen Komponenten – wie Aktoren und Sensoren – zu einer grossen Klasse zusammengeführt.

Die logische Steuerung der Waschmaschine – unter Nutzung der Sensoren und Aktoren – erfolgt in der Klasse ‚Programmsteuerung'.

Was noch fehlt, ist die Steuerung der Tür. Diese Funktion ist übergreifend. Sie ist sowohl mit Sensoren – Türschalter – als auch Aktoren – Zuhaltung – ausgestattet. Diese Funktion greift natürlich auch sehr stark in den Programmablauf ein. Hier ist also wieder eine Klasse definiert worden.

Die Wasserversorgung mit Zu- und Ablauf stellt ein weiteres Modul oder eine weitere Unterklasse dar.

Die elektrische Heizung mit einer Temperaturregelung kann wieder als Klasse aufgefasst werden. Der Regler für die Temperatur kann als PI oder als Fuzzy Regler realisiert sein. Ein definiertes Interface für einen generellen Regler erleichtert den Umstieg zwischen verschiedenen Reglertypen. Der gleiche Regler – die gleiche Klasse mit den implementierten Interfaces - soll auch für die Regelung der Trommel genutzt werden. Hier sieht man klar die Vorteile der Wiederverwendung und Generalisierung von Softwarekomponenten.

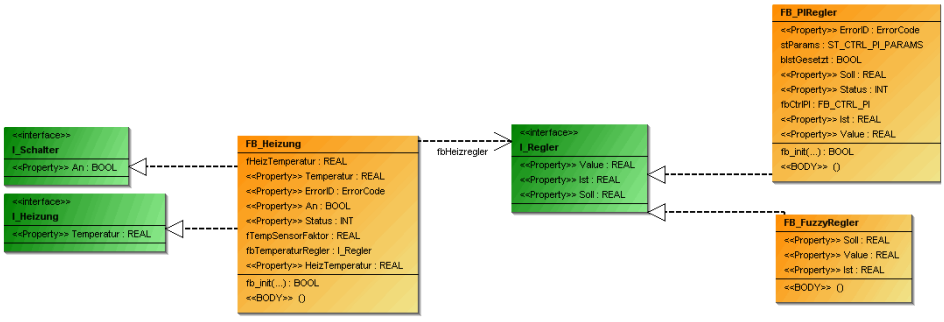


Abb. 4.3: Die Heizung und deren Unterklassen

Die Trommel mit Motor und Drehzahlregler ist wieder eine Klasse.

Wasserversorgung, Heizung und Trommel können zur Wascheinheit zusammengefasst werden. Die Trommelbewegung darf natürlich nur gestartet werden, wenn die Tür geschlossen und verriegelt ist. Nach dem Beenden des Programms darf die Tür erst nach einer Wartezeit wieder entriegelt werden.

Die Bedienung der Waschmaschine ist vergleichsweise einfach gehalten. Neben dem Ein-Ausschalter gibt es nur noch den Programmwahlschalter.

Zentrale Klasse einer Waschmaschine ist die Programmsteuerung. Sie wählt die über den Programmwahlschalter eingestellten Programme aus und startet den Programmablauf. Jedes Programm soll über ein Interface verfügen. Dieses Interface stellt die Ausführung der Teilprogramme (Vorwäsche, Hauptwäsche, Spülen, Schleudern) zur Verfügung. Die einzelnen Unterprogramme sind als Zustandsmaschinen anzusehen. Das Unterprogramm für das Schleudern könnte aus einem Schließen des Zulaufs, einem Start der Trommel mit langsamer Drehzahl, dann dem Beschleunigen auf volle Drehzahl, dem Halten der Drehzahl für eine bestimmte Zeit und dem Abbremsen der Trommel bestehen.

Natürlich sollen Fehler (Wasserzufuhr unterbrochen, Leckage usw.) zentral gemeldet und verarbeitet werden.

4.5 Software strukturieren

Die Vorteile einer Strukturierung des Softwareprogramms in einzelne größere funktionale Teile ist einfach ersichtlich. Das UML Klassendiagramm zeigt das sehr anschaulich. Je weiter die Strukturierung fortschreitet, desto schwerer wird erkennbar, welche Klasse von welcher abgeleitet oder instanziiert worden ist.

Neben der reinen Verteilung von Software in Klassen kann die Funktionalität noch in Methoden ausgelagert werden. Man erhöht damit die Übersichtlichkeit von Software. Getrennte funktionale Einheiten werden in eigenen Methoden zusammengefasst. Dies ist besonders deutlich am Beispiel der Klasse Waschprogramm zu sehen.

Abb. 4.4: Struktur des FB „Waschprogramm30“

Jede Teilfunktion wird in eine eigene Methode implementiert. Das Schleudern ist eine Methode. Hier wird die Trommel bis zur geforderten Drehzahl beschleunigt und dann für eine gewisse Zeit gehalten. In allen Waschprogrammen wird diese Methode aufgerufen.

4.6 Sequenzen programmieren

In der IEC 61131-3 steht mit der Ablaufsprache (AS, SFC) eine Programmiersprache zur Verfügung mit der Abläufe oder Sequenzen einfach programmiert werden können. Diese Sprache ist aber sehr starr und unflexibel. Auch die UML bietet im Bereich der Verhaltensdiagramme einen Diagrammtyp für das Design von Sequenzen an. Das Statechart ist wesentlich flexibler als die Programmiersprache SFC. Für das einfache Waschprogramm kann die Programmierung in SFC oder Statechart durchgeführt werden.

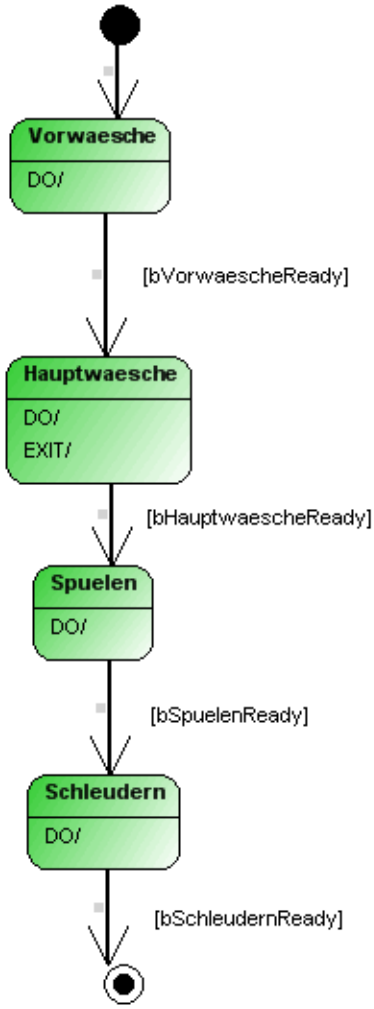


Abb. 4.5: Statechartvariante

4.7 Nebenläufige Sequenzen darstellen und programmieren

Für die Programmierung und/oder Darstellung von nebenläufigen Sequenzen gibt es in der IEC 61131-3 keine Programmiersprache und auch kein Konzept. Natürlich finden wir in der Maschinenprogrammierung häufig nebenläufige Sequenzen. Und natürlich kann man auch diese Aufgabe mit den Mitteln der IEC61131 meistern. Es wird nur meist sehr unanschaulich und damit schlecht wartbar. Die UML bietet mit dem Aktivitätsdiagramm genau einen Diagrammtyp der sich sehr gut dafür eignet, nebenläufige Sequenzen einfach darzustellen. Wenn man dann noch die Möglichkeit hat, aus so einem Aktivitätsdiagramm

automatisch Code zu erzeugen, dann kann der Programmierer den Vorteil der einfachen Darstellung und Programmierung nutzen. Laut Wikipedia definiert sich ein Aktivitätsdiagramm wie folgt:

„Das Aktivitätsdiagramm ist ein Verhaltensdiagramm. Es zeigt eine bestimmte Sicht auf die dynamischen Aspekte des modellierten Systems. Ein Aktivitätsdiagramm stellt die Vernetzung von elementaren Aktionen und deren Verbindungen mit Kontroll- und Datenflüssen grafisch dar. Mit einem Aktivitätsdiagramm wird meist der Ablauf eines Anwendungsfalls beschrieben, es eignet sich aber zur Modellierung aller Aktivitäten innerhalb eines Systems.“ Das Waschprogramm ist hier einmal als Aktivitätsdiagramm entworfen worden:

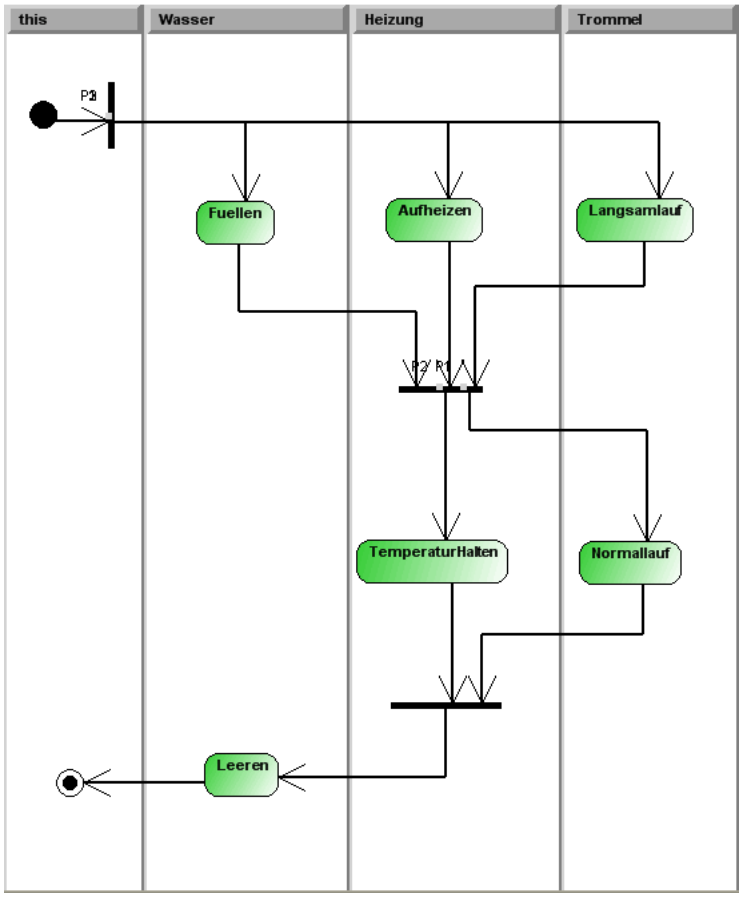


Abb. 4.6: Aktivitätsdiagramm der Maschine

4.8 Software mehrfach nutzen

Das Waschprogramm ist auch ein gutes Beispiel für eine der wichtigsten Eigenschaften objektorientierten Programmierens. Das Schlagwort lautet Vererbung. Darunter ist die Eigenschaft von Klassen zu verstehen, ihre Funktionalität – inklusive aller Methoden und aller Variablen – weiterzugeben. In diesen so genannten abgeleiteten Klassen können alle diese Methoden und Variablen verwendet und erweitert werden. Das heißt in unserem Beispiel, dass die Klasse Waschprogramm die wesentlichen Eigenschaften eines Waschprogramms definiert. Die speziellen Waschprogramme (z.B. das Programm für die 95° Wäsche) haben natürlich die grundlegenden Funktionen, können diese aber erweitern oder parametrieren. Mit dem neuen Schlüsselwort EXTENDS können Basisklassen in abgeleiteten Klassen genutzt werden.

```
1  FUNCTION_BLOCK FB_Waschprogramm95 EXTENDS FB_Waschprogramm
2  VAR_INPUT
3  END_VAR
4  VAR_OUTPUT
5  END_VAR
6  VAR
7  END_VAR
8
```

Abb. 4.7: FB „Waschprogramm“

Erfahrene Programmierer hätten im Fall des Waschprogramms eine andere Strategie gewählt. Unter Verzicht auf Vererbung hätte man dem Waschprogramm auch einfach ein Rezept übergeben können. Dieses Rezept in Form einer Tabelle würde für jedes Waschprogramm individuell steuern, welche Drehzahl, welche Temperatur und welche Zeiten einzuhalten sind. An diesem Beispiel sieht man deutlich, dass man nicht immer Objektorientierung einsetzen muss. Eine klassische Lösung hat hier durchaus ihre Berechtigung und ist in diesem Fall vielleicht auch die besser lesbare Lösung.

4.9 Interfaces

Wenn man Mehrfachvererbung vermeiden will, dann können so genannte Interface-Klassen, also Klassen ohne eigentliche Implementation, sehr hilfreich sein.

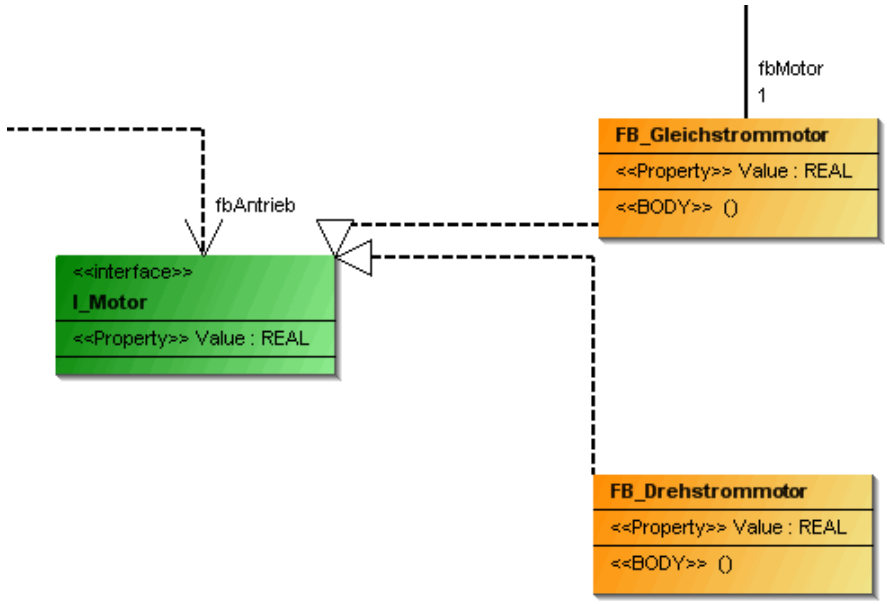


Abb. 4.8: Klassenhierarchie des Motors

In diesem Ausschnitt benutzen sowohl der Gleichstrommotor als auch der Drehstrommotor das gleiche Interface I_Motor. Sollte noch ein weiterer Motortyp notwendig sein, so sollte auch dieser das Interface I_Motor nutzen. Damit sind die Motoren untereinander – wenigstens in der Software – austauschbar.

4.10 Zusammenfassung

Die Waschmaschine ist sicherlich ein vergleichsweise einfaches Beispiel. Dennoch sieht man deutlich, dass mit Strukturierung von Software eine bessere Übersichtlichkeit und Wartbarkeit selbst bei kleinen und einfachen Applikationen erreicht werden kann. Die neue ‚Sprache' Statediagramm erlaubt die einfache grafische Definition von Abläufen. Mit der ‚Sprache' Aktivitätsdiagramm können auch komplexe Abläufe mit parallelen Vorgängen einfach ausgedrückt werden.

Natürlich erfordert die Verwendung dieser neuen Technologien ein Umdenken. Es erfordert auch ein Umlernen. Bestehende Software muss analysiert und zu großen Teilen neu geschrieben werden. Hier ist ein gewisser Aufwand notwendig. Die Vorteile der objektorientierten Programmierung wird man auch nicht sofort sehen. Aber bei Erweiterungen der Software oder bei der Wartung der Software wird es gerade bei sehr komplexen Programmen deutlich einfacher sein.

Einige werden sicherlich auch sagen: Genau so hätte ich es auch mit einer Standard-IEC61131 programmieren können. Viele haben das sicherlich auch schon so getan. Neu ist die Erweiterung der Norm, um näher an Sprachen wie C++ und C# heranzukommen. Damit macht dann auch der UML Ansatz Sinn. Und auch dieser Ansatz macht nur Sinn, wenn man dem SPS Programmierer alle notwendigen Tools in Form von Editoren und Compilern in einer Suite anbietet. Eine neue Technik wird nur Erfolg haben, wenn sie von einfachen Tools unterstützt wird.

In unmittelbarer Zukunft wird es die Aufgabe sein, die Programmierer an die neue Technik heranzuführen, die Vorteile darzustellen und bei den Entscheidern den Vorteil positiv darzulegen um wiederum Ressourcen für einen neuen objektorientierten Weg frei zu machen.

4.11 Referenzen

[1] Bernd Oestereich, Analyse und Design mit UML 2.1, Oldenbourg Wissenschaftsverlag München, 2006

5 Anwendungsmöglichkeiten der UML-Editoren im Verpackungsmaschinenbau aus Sicht eines Systemanbieters

Dipl. Inf. (FH) Sebastian Diehm, ELAU GmbH, Software Entwicklung

5.1 Engineering komplexer mechatronischer Systeme

Der Siegeszug der Mechatronik in vielen Bereichen des Maschinenbaus hat eine starke Zunahme des Anteils an Elektronik und Software am Gesamtwert von Maschinen zur Folge. Der damit einher gehende Anstieg der Komplexität verlangt nach veränderten Engineering-Prozessen und neuen Software-Konzepten, um steigenden Aufwendungen im Engineering entgegen zu wirken.

In solchen Software-Konzepten ist die durchgehende Standardisierung von Detaillösungen bis hin zur Gesamtlösung ein Aspekt mit hohem Stellenwert: Sowohl im Grob- als auch im Feingranularen spielen hier nicht nur die jeweils ausgeführten Funktionen, sondern auch die dabei durchlaufenen Betriebszustände von Funktionsbausteinen, Software-Teillösungen oder ganzen Maschinen eine große Rolle. Auf Maschinenebene erlangen daher mit steigender Tendenz Richtlinien wie beispielsweise PackML oder der Weihenstephaner Standard Bedeutung.

Komplexe Mechaniken erfordern oft komplexe Software-Lösungen. Auch wenn der Maschinenbauer einen Bibliotheksbaustein einfach einbinden und parametrieren kann, so arbeiten im Hintergrund oft mehrere Bausteine auf komplexe Weise zusammen, um die gewünschte Funktion zu realisieren. In Verbindung damit bringen neue objektorientierte Sprachmittel in der IEC 61131-3 zwar auch neue Lösungsmöglichkeiten ein. Gleichzeitig erhöhen diese aber auch die Gesamtkomplexität des Systems und wecken den Bedarf, das System modellieren bzw. grafisch abbilden zu können. Daher ist die Entwicklung von Editoren für UML [1] als PlugIn für Programmiersysteme wie CoDeSys als großer Fortschritt zu bewerten. Sie würden die entscheidenden Funktionalitäten bieten, die eine vielseitige und rationelle Anwendung von UML bei der Erstellung von Bibliotheken und ebenso von Maschinenprogrammen erfordert.

In Kooperation mit dem Fachgebiet „Eingebettete Systeme“ von Frau Prof. Vogel-Heuser an der Universität Kassel und mit weiteren Partnern aus der Industrie wurden verschiedene UML-Editoren als Ergebnis des Forschungsprojektes [2] realisiert. Dabei fanden neben Plattform-Charakteristika (zyklische Ausführung, Echtzeitbetrieb…) auch die neuen objektorientierten Erweiterungen der IEC 61131-3 Programmiersprachen Berücksichtigung, wie sie die Firma 3S in der neuesten Version ihres Programmiersystem CoDeSys definiert. Diese fließen bereits in den Standardisierungsprozess ein und werden in absehbarer Zeit offiziell freigegeben.

Mit CoDeSys und den UML-Editoren als PlugIn steht damit ein komplettes Paket zur Verfügung, das dem Software-Entwickler zum einen neue Funktionen zugänglich macht.

Zum anderen umfasst es notwendige Hilfsmittel der „klassischen“ Informatik, um diese Features und die damit realisierten komplexen Systeme auch zu beherrschen.

5.2 Anwendungsmöglichkeiten der UML-Editoren

Im Folgenden werden potenzielle Anwendungsgebiete der UML-Editoren aufgezeigt. Dabei liegt der Schwerpunkt auf deren Einsatz bei der Entwicklung von IEC 61131-3 basierten Bibliotheken, dem Hauptanwendungsgebiet für einen Systemanbieter wie ELAU. Ein Rückblick in die Entwicklungsgeschichte zeigt, dass neben den Technologiefunktionen auch Systemfunktionen einen großen Anteil an der Entwicklung ausmachen. Daher sollen im Weiteren verschiedene Beispiele zeigen, wie die UML-Editoren bei der Bibliotheksentwicklung zum Einsatz kommen können.

UML-Editoren sind auch für Maschinenbauer von Nutzen. Besonders im Serienmaschinenbau und für modulare Maschinen werden häufig Bibliotheken generiert, die ebenfalls Technologiefunktionen bzw. Maschinenmodule enthalten. In solchen Fällen verfolgt der Maschinenbauer ähnliche Ziele wie ein Systemanbieter und benötigt daher vergleichbare Werkzeuge. Aber auch bei der Realisierung des Maschinenprogramms, dem vorrangigen Ziel eines Maschinenbauers, lassen sich Anwendungsgebiete für UML-Editoren finden. Für die Darstellung diesbezüglicher Aspekte sei jedoch auf die Beiträge anderer Projektteilnehmer verwiesen.

5.2.1 Einsatz von Klassendiagrammen

Klassendiagramme bieten sich an, um das komplexe Zusammenspiel vieler Instanzen zu strukturieren und Beziehungen zwischen diesen aufzuzeigen. So werden aktuell bei ELAU Funktionsbausteine für komplexe Mechaniken wie Roboter realisiert. Verschiedene Verwaltungsinstanzen halten Roboterdaten vor, auf die strukturiert zugegriffen werden muss. Abhängigkeiten zwischen den verschiedenen Komponenten lassen sich mit Hilfe von Klassendiagrammen bereits in der Analysephase in einer ersten Übersicht darstellen. Im weiteren Verlauf der Entwicklung bieten diese Analysediagramme dann Hilfestellung beim Design. Durch die volle Synchronität zwischen Diagramm und Code werden die Rümpfe neu modellierter Elemente ohne Programmier-Aufwand automatisch erstellt.

Da bei ELAU Bibliotheken vorwärtskompatibel weiterentwickelt werden, entstanden im Laufe der Jahre für das PacDrive™ –System einige Versionen derselben Funktionsbausteine. Über Vererbungshierarchien lassen sich solche Weiterentwicklungen künftig realisieren. Sie sind dann in Klassendiagrammen darstell- und dokumentierbar. Durch die objektorientierten Programmiermöglichkeiten hinsichtlich Vererbung und Interfaces ist es künftig möglich, konsequent zwischen Daten und Verhalten zu trennen und beides getrennt zu vererben bzw. zu implementieren. So ließen sich z.B. Varianten von Bausteinen für Mechaniken abbilden, im Falle von Robotern z. B. 2-Achs-Delta- oder 3-

Achs-Delta-Versionen. Mit dem Klassendiagramm lassen sich solche komplexen Varianten effizient entwerfen und realisieren.

Außer für die Realisierung von Technologiefunktionen kommen die UML-Editoren auch bei der Realisierung von System-Bibliotheken zum Einsatz. Hier werden in erster Linie die objektorientierten Erweiterungen der Sprache genutzt. Für klassische Informatiker, die an Systembibliotheken arbeiten, ist die Verwendung des UML-Klassendiagramm-Editors für Designzwecke selbstverständlich und Teil des üblichen Vorgehens im Rahmen der Entwicklungsrichtlinien. Werden mit ihm Vererbungs- und Implementierungsbeziehungen sowie Klassen- und Interface-Hierarchien modelliert, ist bereits ein Teil der Systemdokumentation realisiert. Die Package-Elemente in den Klassendiagrammen lassen sich nutzen, um in Dokumentations-Diagrammen die Abhängigkeiten von Bibliotheken zu modellieren. Mit der konsequenten Nutzung der Editoren für die Dokumentation geht auch eine homogene Vermittlung technischer Inhalte einher.

Im Rahmen der Bibliotheksentwicklung dienen die Online-Ansichten von Klassendiagrammen primär zu Debugging-Zwecken. Die Diagramme befinden sich nach dem Erzeugen einer Bibliothek für den Kunden – den Benutzer – nicht mehr im Zugriff, daher steht auch keine Online-Ansicht der Bibliothek im Maschinenprogramm zur Verfügung.

5.2.2 Einsatz von StateCharts

Bisher wurde das Pattern der Statemachine bzw. endlicher terminierender Automaten bei ELAU in fast allen Funktionsbausteinen der Bibliotheken eingesetzt. Dieses Pattern erlaubt es, „unabhängig“ von der zyklischen Ausführung genau den Codeblock auszuführen, der im aktuellen Kontext notwendig ist. Der Kontext wird dabei z.B. von Signalen bzw. Eingangsparametern des Bausteins definiert.

In einer Analysephase wird in der Regel ein erster Entwurf für die Statemachine eines Funktionsbausteins in einem Modellierungstool entworfen. Dabei kommen entweder proprietäre Modellierungssprachen zum Einsatz oder UML-StateCharts. Die modellierten Statemachines werden nach IEC 61131-3 Code überführt, in der Regel in Form von Structured Text (siehe Abb 2.1).

```
iState : INT;
...
...
CASE iState OF
1:
 do();
 IF xDing1 THEN iState := 20; END_IF
20:
 do();
 if xDing2 THEN iState := 30; END_IF
 ...
30: (* kommen auch verschachtelt vor: iSubState30 *)
 CASE iSubState30 OF
 ...
 99:
  iState := 40;
 END_CASE (* iSubState30 *)
```

Abb. 5.1: Codebeispiel: prinzipielles Basispattern für eine Statemachine in der Programmiersprache Structured Text

Das Beispiel zeigt die Realisierung von komplexen Konstrukten, z.B. von verschachtelten Statemachines. Auch die Ausführung mehrerer States pro Zyklus wird bisher schon genutzt. Im Beispiel würde hierfür entweder Extracode nach dem letzten END_CASE benötigt oder eine WHILE-Schleife um die ganze Statemachine. Hierin begründet sich unter anderem der Verzicht auf Ablaufsprache an dieser Stelle. Vordergründig läge dies nahe, weil hiermit eine grafische Programmierung der Statemachine erfolgen könnte. Wie bereits zuvor an anderer Stelle [3] detaillierter ausgeführt wurde, gibt es eine ganze Reihe von Gründen, die gegen die Nutzung von Ablaufsprache sprechen, um die Statemachine eines Funktionsbausteines zu realisieren. Stichpunktartig seien folgende Gründe genannt:

- Keine grafische Repräsentation hierarchischer Automaten
- Restriktion hinsichtlich Ausführungsgeschwindigkeit: 1 Zustand pro Zyklus

- Schwächen in der Benutzbarkeit: fehlende Modellierungselemente, Pseudozustände und Sprünge sind zur Realisierung notwendig

Daher ist die Nutzung des UML-StateChart-Editors zur Implementierung der Statemachines der Funktionsbausteine nur konsequent. Zum einen stehen alle notwendigen Modellierungselemente zur Verfügung, mehr als z.B. Ablaufsprache anbietet. Zum anderen lassen sich die Restriktionen von Ablaufsprache hinsichtlich der Ausführungsgeschwindigkeit vermeiden. Außerdem überbrückt der UML-StateChart-Editor der Universität Kassel den bisher existierenden Toolbruch. Im Programmiersystem kann die Statemachine entworfen werden und der Entwurf wird 1:1 für die Implementierung verwendet. Letztlich IST der Entwurf die Implementierung. Aus dieser Perspektive wird die Statemachine in ihrer „nativen" Sprache realisiert. Damit entfallen jegliche Differenzen zwischen Spezifikation, Implementierung und Dokumentation des dynamischen Verhaltens eines Funktionsbausteins!

Ohne Debugging-Fähigkeiten wäre der StateChart Editor nur begrenzt zur Realisierung von komplexen Technologiefunktionen einsetzbar. Auch wenn der Endkunde die erstellten Diagramme (den „Code") bei der Nutzung der Bibliothek nicht mehr sieht und aus seiner Sicht Debugging-Möglichkeiten für ELAU-Bibliotheken gar nicht zur Verfügung stehen, so sind diese doch wesentliche Voraussetzungen für eine effiziente und fehlerminimierte Implementierung der Funktionsbausteine und Bibliotheken.

Anhand der Online-Ansicht - z.B. der Werte der Bedingungen - kann der Entwickler schließen, warum sein in StateChart realisierter Algorithmus im aktuellen Kontext fehlgeschlagen ist. Durch das Setzen von Breakpoints und das Durch-Steppen des Diagramms, kann der Programmierer einzelne Schritte des Ablaufs nachvollziehen und so erste Entwicklertests auf dem Algorithmus ausführen. Das gezielte Setzen (Forcen) von Werten erleichtert darüber hinaus gezielte Tests einzelner Zweige.

Besonders für die simulierte Ausführung des Bausteins in CoDeSys oder für einen Lauf mit einer Testmechanik im Labor eignet sich die trace-artige Aufzeichnung der durchlaufenen Zustände. Damit ist es möglich, mit nur minimaler Beeinflussung der Laufzeit, das Echtzeitverhalten des Funktionsbausteins bzw. des in StateChart realisierten Algorithmus aufzuzeichnen und später nachzuvollziehen. Alle drei Aspekte zusammen - intuitive Modellierungsmöglichkeiten, effiziente Debugging-Möglichkeiten und direkt verfügbare Online-Ansichten - machen den StateChart-Editor zu einem mächtigen Werkzeug, das künftig bei der Realisierung vieler Technologiefunktionen zum Einsatz kommen wird.

Neben einigen weiteren nützlichen Neuerungen des StateChart-Editors, z.B. Exception-Transitionen oder BinaryChoices bieten primär die <<InCycle>>-States interessante Anwendungsmöglichkeiten. Dies soll an einer nicht geschäftskritischen Diagnosefunktionalität gezeigt werden: Eine Funktion soll Diagnoseinformationen auf einen Datenträger schreiben. Um das Restsystem nicht zu beeinflussen, wird diese Funktion in einer eigenen niederprioren Task zyklisch ausgeführt. Hauptziel ist es, so viele Diagnoseinformationen wie möglich pro Zyklus zu schreiben. Dabei sind zwei Randbedingungen kritisch:

- Die Zykluszeit darf nicht überschritten werden.

- Der Datenträger hat nur eine begrenzte Größe.

```
xWriteError := FALSE;
TPL_FC_CanContinueWriting(iq_stCanContinueWriting              :=
iq_stComStructure.stCanContinueWriting);
(* loop as long as we can continue to write *)
WHILE(iq_stComStructure.stCanContinueWriting.xCanContinue) DO
      CASE diCmd1ExecState OF
            1: (* start writing *)
                  ...
            2:
                  ...
            7 :
                  ...
            diCmd1ExecState := 0;
           EXIT; (* exit the while loop *)
      END_CASE
      ...
      (* check again whether we can continue writing *)
      TPL_FC_CanContinueWriting(iq_stCanContinueWriting       :=
iq_stComStructure.stCanContinueWriting);
END_WHILE
```

Abb. 5.2: Codebeispiel: unvollständiger Auszug aus einer ELAU Diagnosefunktion, die Daten auf einen Datenträger schreibt

Nach jedem Schritt in der StateMachine prüft der Algorithmus, ob die Randbedingungen weiterhin unkritisch sind. Wenn ja, wird der nächste Schritt in der Maschine ausgeführt - solange, bis alle Informationen geschrieben sind. Wird eine der Randbedingungen kritisch, pausiert der Algorithmus den Schreibvorgang, bis eine Fortsetzung möglich ist. Im Idealfall finden alle Schreibvorgänge innerhalb eines Zyklus statt. Daher bietet sich eine Realisierung des Algorithmus mit <<InCycle>>-States an.

Abb. 5.3 zeigt beispielhaft eine Realisierungsmöglichkeit der Diagnosefunktion mit dem StateChart-Editor. Die eigentliche Schreib-Logik wird dabei jeweils in den Entry-Aktionen der States realisiert, weil das Schreiben nur einmal ausgeführt werden darf. Würde die Do-

Aktion des States dafür genutzt, würde über mehrere Zyklen hinweg geschrieben werden, wenn die Randbedingungen ein Weiterschalten zum nächsten Zustand nicht zulassen. Das im Bild gezeigte Guard-Flag sollte bei einer echten Realisierung natürlich ein Property oder eine Methode sein, so dass auch eine Prüfung der Randbedingungen erfolgen kann.

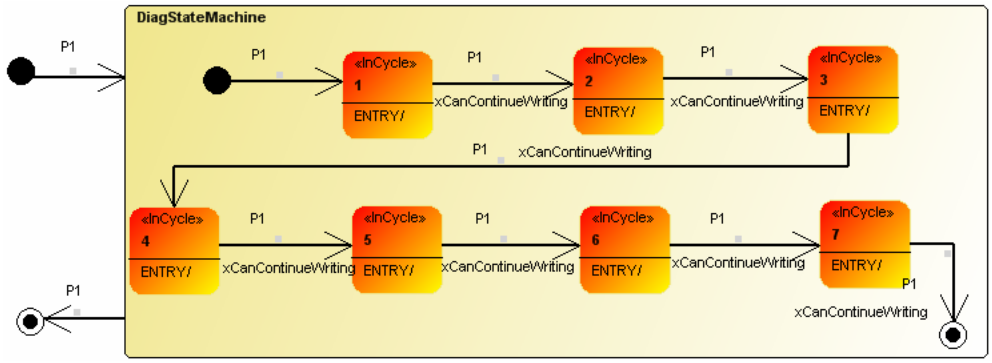


Abb. 5.3: Mögliche Realisierung der Diagnosefunktion mit <<InCycle>>-States

5.2.3 Einsatz von Aktivitätendiagrammen

Aus Sicht eines Systemanbieters ist es gut vorstellbar, Aktivitätendiagramme zu Analyse-Zwecken einzusetzen. Diese wären ein gutes Hilfsmittel, um komplexe Kundenanforderungen zu verstehen und zu dokumentieren. Hauptziel wäre dabei, den Kontext eines Bausteins zu ermitteln - sprich - in welchem Rahmen und unter welchen Bedingungen er innerhalb der Maschine eingesetzt wird.

Beim Verpackungsmaschinenbauer ist es ebenfalls vorstellbar, in einer frühen Analyse-Phase Aktivitätendiagramme einzusetzen. Dabei könnte vor allem die Kommunikation zwischen den Gewerken verbessert werden, z.B. zwischen der Verfahrenstechnik und der Softwareentwicklung. Dadurch könnte im späteren Maschinenprojekt auf bereits existierende erste Analysen zurückgegriffen und anhand dieser Diagramme das Design verfeinert werden.

Da Aktivitätendiagramme Code erzeugen und ausführbar sind, können Teile der Design-Diagramme im Projekt genutzt werden. Aufgrund des Petri-Netz-artigen Charakters des Aktivitätendiagrammes eignet es sich vor allem zur Prozessautomatisierung auf einer funktional hohen Ebene, z.B. bei der Werkstückbearbeitung. Dabei durchläuft das Werkstück im Bearbeitungsprozess Schritt für Schritt verschiedene Stationen. Diese Schrittfolge mit einer jeweils relativ „langen“ Verweildauer in einer Station lässt sich fast intuitiv mit dem Aktivitätendiagramm modellieren. Die Details der einzelnen Prozessschritte werden dann i.d.R. wieder in einer klassischen Sprache (strukturierter Text…) oder evtl. in StateChart implementiert. Hierbei ergibt sich der Vorteil, dass durch die Online-Ansicht des Diagramms bereits eine „Visualisierung“ des Ablaufes mitgeliefert

wird. Der Software-Entwickler müsste keine separate Inbetriebnahme-Visualisierung entwerfen oder könnte sich dabei zumindest an den existierenden Aktivitätendiagrammen orientieren.

Damit wird auch deutlich, dass diese Diagrammart zur „Ausführung“ von Code und zur Visualisierung des Zustandes (Online-Ansicht) für Verpackungsmaschinen wenig geeignet ist. In Verpackungsmaschinen laufen die „interessanten“ Prozesse synchron und mit hoher Geschwindigkeit ab, wobei der Produktstrom kontinuierlich manipuliert wird. Prinzipiell eignen sich Aktivitätendiagramme gut für Batch- oder Einzelstückprozesse. Dort kommen die oben erwähnten Vorteile voll zum Tragen.

5.3 Einsatz von StateCharts für PackML

In den letzten Jahren entstanden in verschiedenen Bereichen der Verpackungs- und Konsumgüterindustrie Standards und Richtlinien, die sich auf die Definition der Betriebsarten und Zustände der Maschine fokussieren. Erwähnenswert sind hier vor allem der Weihenstephaner Standard [4] und die PackML-Richtlinie [5]. Der Weihenstephaner Standard orientiert sich im Schwerpunkt an den Anforderungen der deutschen Getränkeindustrie. PackML verfolgt dagegen einen breiteren Ansatz für die Verpackungsindustrie allgemein und wird vor allem von global agierenden Unternehmen nach vorne getrieben.

Beide Standards verfolgen ähnliche, teilweise sich überlappende und identische Ziele. Im Mittelpunkt steht immer die volle Integration der Verpackungsmaschine in übergeordnete Produktionsdatenerfassungs- und Steuerungssysteme. Dazu werden produktionsrelevante Kennzahlen definiert und die Art und Weise, wie diese ermittelt werden sollen. In engem Zusammenhang damit steht die Definition der Betriebsarten einer Maschine sowie der Zustände dieser Betriebsarten. Für den Konsumgüterproduzenten entsteht so der Vorteil, dass er lediglich auf die Verwendung des Standards - z.B. PackML - bei einer Maschine achten muss, und dann eine ihm bekannte, standardisierte Maschinensteuerung sowie ein einheitliches Look&Feel erwarten kann.

Für den Maschinenbauer ergeben sich ebenfalls Vorteile. So können Teile des Maschinenprogramms als wieder verwendbare Bausteine oder Bibliothek ausgelegt werden. Für neue Maschinen bzw. Module ist die Realisierung der Baustein-Betriebsarten damit ebenfalls vorgezeichnet. Dadurch steigt insgesamt die Realisierungsgeschwindigkeit der Maschine. Das Engineering wird vereinfacht und ebenso die Wartung, da Teile der Software auf einem ähnlichen Muster basieren. Realisiert ein Maschinenbauer eine Maschine gemäß PackML, kann er außerdem unterschiedlichen Kunden eine standardisierte Schnittstelle zur Produktionsdatenerfassung anbieten. Diese Schnittstelle muss so nur einmal definiert und realisiert werden und nicht individuell in jedem Kundenprojekt neu „erfunden“ werden.

Nach dem Zusammenschluss von OMAC und ISA im Jahr 2005 erfolgte eine Harmonisierung der jeweiligen Standards, u.a. auch der PackML-Richtlinie der OMAC und der ISA-88 Standards [6]. Als aktuellstes Ergebnis dieses Zusammenschlusses kann der gemeinsame Technical Report - ISA-TR88.00.02 [7] - beider Arbeitsgruppen gelten. Er zeigt die Anwendung von PackML bei der Realisierung diskreter Maschinen, z.B. von Verpackungsmaschinen. Im Folgenden soll demonstriert werden, wie sich mit UML-StateCharts die PackML-Definitionen realisieren lassen.

Ein zentrales Element des ISA-TR88 ist der Machine State. Ein Machine State definiert dabei den kompletten aktuellen Zustand der Maschine. In einem Machine State werden weiterhin die für diesen Zustand typischen Operationen auf der Maschine ausgeführt. Der Standard definiert genau 17 Machine States und die zwischen diesen erlaubten Übergänge. Die Übergänge von einem Zustand zum anderen werden über State Commands ausgelöst. Weiterhin existieren 3 Arten von States. Je nach Art des States können die Kommandos aus dem Programm selbst heraus abgesetzt werden (z.B. StateComplete - SC) oder Kommandos resultieren aus Signalen, ausgelöst z. B. durch Bediener-Interaktionen.

In ISA-TR88 ist ein Base State Model definiert, das die Anordnung der Machine States, deren Übergänge und der State Commands zeigt (siehe Abb. 5.4). Damit dieses Modell an individuelle Maschinen adaptiert werden kann, lässt der Standard ein Eliminieren beliebiger States zu. Weiterhin definiert der Standard Machine Control Modes (Maschinen-Betriebsarten), von denen eine Maschine beliebig viele implementieren kann. Jede Betriebsart wird dann durch ein zugeschnittenes State Model realisiert.

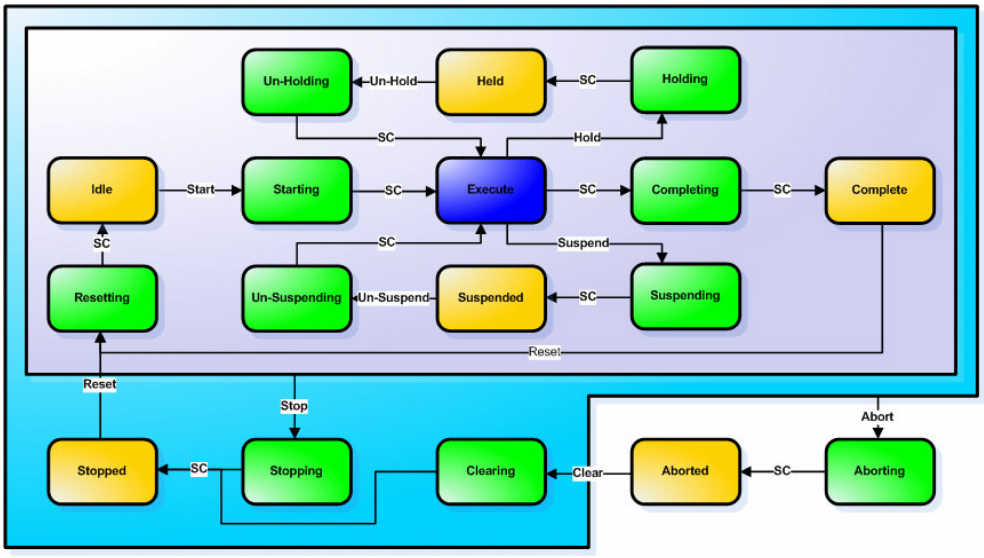


Abb. 5.4: Das Machine State Model aus der PackML Richtlinie - eine inhaltlich identische Darstellung zum Base State Model in ISA-TR88 [7]

Aufgrund der großen Bedeutung der PackML-Richtlinie, besonders für Endkunden in den USA, bietet ELAU seit 2002 bibliotheksbasierte Lösungen für die einfache Nutzung von PackML an. Da der Standard im Laufe der Zeit weiterentwickelt wurde und auch die Anforderung der Kunden mitgewachsen sind, hat auch ELAU seine Lösungen in diesem Bereich kontinuierlich weiterentwickelt. Die letzte Entwicklung auf diesem Gebiet fand in der vor kurzem frei gegebenen Bibliothek „PD_PackML3 V20.00“ ihren Niederschlag. Dabei wurde das Base State Model als Visualisierung für CoDeSys 2.3 implementiert. Während die Visualisierung lediglich eine Anzeige des aktuellen Machine States darstellt, erfolgt die eigentliche Ausführung und Steuerung über das darunter liegende IEC-Programm. Dies kann zum Beispiel mit Ablaufsprache realisiert werden. Das State Model kann der Kunde dann als Vorlage für seine Entwicklung verwenden und an seine Maschine anpassen.

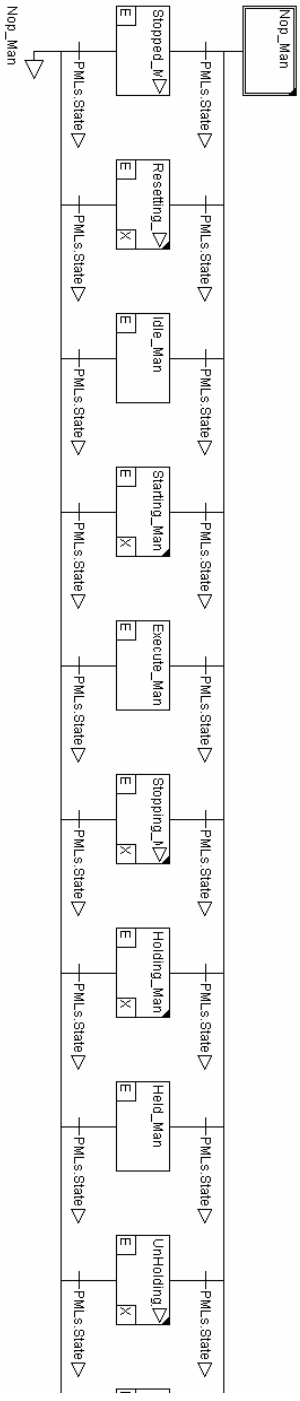


Abb. 5.5: Der Screenshot zeigt ausschnittsweise eine mögliche Realisierung des Base State Models aus ISA-TR88 / PackML für ein CoDeSys V2.3 System

Damit wird augenfällig, welche Nachteile die aktuelle Variante der Realisierung birgt: Die in ISA-TR88 / PackML definierte Darstellung der Maschinenzustände orientiert sich stark an StateCharts. Die bisher verwendeten Implementierungssprachen, egal ob Strukturierter Text oder Ablaufsprache, entfernen sich wieder von der definierten Darstellung. Erst der Endkunde bzw. der Maschinenbediener bekommt mit der PackML-Visualisierung wieder das in den Richtlinien spezifizierte Interface zu sehen.

Mit CoDeSys V3 und dem UML-Editor-PlugIn steht künftig eine noch effektivere Realisierungsmöglichkeit für die PackML State Models zur Verfügung. Da bei der Spezifikation von PackML UML StateCharts Berücksichtigung fanden und die Ähnlichkeit schon rein visuell offensichtlich ist, können die PackML State Models künftig auf einfache Weise in Form von StateCharts realisiert werden. Neben den bereits erwähnten Vorteilen - kein Tool-Bruch mehr, keine Differenz zwischen Spezifikation, Implementierung und Dokumentation sowie die Implementierung der State Models in ihrer „ursprünglichen“ Sprache - lassen sich noch weitere Vorteile anführen. Mit StateCharts sind PackML State Models intuitiv und fast 1:1 realisierbar. Der Programmierer muss für die State Models keine Inbetriebnahme-Visualisierung programmieren - in den meisten Fällen dürfte die Online-Ansicht seiner State Model Implementierung ausreichend sein.

5.4 Zusammenfassung

Wie in den Kapiteln 2.2 und 2.3 ausgeführt, bieten die neuen UML-Editoren sowohl für Systemanbieter als auch für Maschinenbauer konkrete Vorteile bei der Software-Entwicklung bzw. beim Engineering. Beim Systemanbieter entfaltet sich der Nutzen in mehreren Bereichen:

Klassendiagramme bieten sich an, um bei der Entwicklung bzw. Weiterentwicklung von Technologiebausteinen das komplexe Zusammenspiel vieler Instanzen zu strukturieren

und Beziehungen zwischen diesen aufzuzeigen. Daraus ergeben sich verbesserte Möglichkeiten, neue Bausteine oder Varianten über Vererbungshierarchien mit reduziertem Aufwand zu realisieren.

Der UML-StateChart-Editor eignet sich zur Implementierung der Statemachines der Funktionsbausteine. Er stellt alle hierzu notwendigen Modellierungselemente zur Verfügung und vermeidet Nachteile, die beispielsweise bei der bisher hierzu verwendeten Ablaufsprache auftreten.

Aktivitätendiagramme sind beim Systemanbieter für Analyse-Zwecke einsetzbar. Sie bieten sich z. B. als Hilfsmittel an, um komplexe Kundenanforderungen zu verstehen und zu dokumentieren. Im Fokus steht dabei die Möglichkeit zur Ermittlung der Umstände unter denen Funktionsbausteine verwendet werden. Aktivitätendiagramme sind darüber hinaus auch für den Maschinenbau interessant, allerdings weniger für den Verpackungsmaschinenbau.

Einen großen Vorteil stellen für Systemanbieter und Maschinenbauer gleichermaßen die besseren Implementierungsbedingungen für das PackML-State Model dar. PackML State Models sind künftig auf einfache Weise in Form von StateCharts realisierbar. Dieses Vorgehen vermeidet dabei die bisherigen Nachteile wie Tool-Bruch, Differenzen zwischen Spezifikation, Implementierung und Dokumentation. Inkonsistenzen zwischen Visualisierung und dem ausführenden IEC-Programm, die bisher eine Inbetriebnahme-Visualisierung erforderten, werden eliminiert.

5.5 Ausblick

Vor einem Jahr stand an dieser Stelle die Formulierung eines Anforderungskatalogs für die zu entwickelnden UML-Editoren im Fokus [3]. Heute darf behauptet werden, dass im Prinzip alle zentralen Punkte dieses Katalogs erfüllt sind. Partielle ´Problemzonen´ im Umgang mit den Editoren resultieren aus deren Prototypenstatus, rütteln jedoch nicht an der grundsätzlichen Qualität der realisierten Lösungsansätze.

Tab. 5.1: Während des Automation Symposium 2008 vorgestellte zentrale Anforderungen an die zu entwickelnden UML-Editoren [3]

Übersicht über zentrale Punkte aus dem Anforderungskatalog:
UML Modellierung in die Programmierumgebung integriert
Bijektive Abbildung zwischen Modell und Code (Reverse Engineering)
Modellierung von FunctionBlocks in Klassendiagrammen
Modellierung von Statecharts ohne die SFC- Restriktionen
Debug-fähige Statecharts
Sichten-Konzepte in allen Diagrammen, um Elemente verbergen zu können
Intuitive Bedienbarkeit aller Diagrammeditoren

Die Überführung der heute vorhandenen Editoren in marktfähige Produkte bedarf daher unbedingt einer Fortsetzung des Projektes. Über die Zeitfrage hinaus wird dann auch die Praxis zeigen, ob die Modellierungselemente der UML-Editoren in der heutigen Form ausreichen. Denkbar wäre zum Beispiel ein Bedarf von Historien in Hierarchischen CompositeStates oder anderer Elemente. Letztendlich ist jedoch immer eine Betrachtung notwendig, ob sich der Aufwand für die Entwicklung neuer Elemente auch durch einen entsprechend breiten Bedarf rechtfertigen lässt. Aus diesem Blickwinkel stellen die heute verfügbaren UML-Editoren ohne Frage ein sinnvolles Paket an Funktionalitäten zur Verfügung.

5.6 Referenzen

[1] Unified Modelling Language: Superstructure, version 2.1.1, 05.02.2007, by OMG

[2] Vogel-Heuser B.: Forschungsprojekt: Steigerung der Effizienz und Qualität im Software-Engineering der Automatisierungstechnik für die Domäne des Maschinen- und Anlagenbaus, im Juni 2006

[3] Diehm S.: Anforderungen an die Modellierung von Funktionsbausteinen mit UML aus Sicht eines Systemanbieters, in: Automation & Embedded Systems, Oldenburg Verlag, München, 2008

[4] Langowski, Kather A., Voigt T.: Weihenstephaner Standards für die Betriebsdatenerfassung bei Getränkeabfüllanlagen, Teil 1 – 4, TU München, Wissenschaftszentrum Weihenstephan, Freising, 2005

[5] Packaging Machine Language V3.0 Mode & States Definition Document, Juni 2006, by OMAC

[6] The Instrumentation, Systems and Automation Society: ISA-88.00.01 Batch Control Part 1: Models and Terminology; ISA-88.00.02 Batch Control Part 2: Data Structures and Guidelines for Languages

[7] The Instrumentation, Systems and Automation Society, ISA-TR88.00.02 – Machine and Unit States: An Implementation Example of ISA-88, 1. August 2008

6 Modularität und Wiederverwendung im Engineering des Maschinen- und Anlagenbaus

Anforderungen an Programmierverfahren, UML und Tools

Dr.-Ing. Oliver Frager, teamtechnik GmbH

Dipl.-Ing.(FH) Walter Nehr, teamtechnik GmbH

Kurzfassung:

Dieser Aufsatz beschreibt Kriterien einer zweckmäßigen Modularität im Maschinen- und Anlagenbau, die helfen kann, zu einer Steigerung der Wiederverwendbarkeit von Modulen und somit zur Zielsetzung einer Produktivitätssteigerung beizutragen. Dies wird am Beispiel der modularen Anlagenplattform TEAMOS verdeutlicht. Es wird dargelegt, wie im Bereich der Steuerungs-Software durch Einsatz von objektorientierten Programmierverfahren Voraussetzungen geschaffen werden können, die Wiederverwendbarkeit von Software-Modulen zu steigern. Aus der Praxis des Maschinen- und Anlagenbaus werden konkrete Anforderungen an fortschrittliche, objektorientierte Steuerungs-Programmierverfahren, an die heutige gängige, vereinheitlichte Modellierungssprache Unified Modelling Language (UML) sowie an Programmier-Tools gestellt, um die genannte Zielsetzung erreichen zu können.

Abb. 6.1: TEAMOS-Anlage

6.2 Heutige Situation der Modularität im Anlagenbau

An der Entwicklung und Fertigung von Produktsanlagen sind unterschiedliche Disziplinen, wie Maschinenbau, Elektrotechnik und Automatisierungs-Software, beteiligt.

Jede dieser Disziplinen schafft und nutzt dabei ihre eigenen Module. Typische Module in der Disziplin Maschinenbau oder Verfahrenstechnik sind dabei beispielsweise Ventile, Pumpen und Motoren. Die Disziplinen Elektrotechnik und Automatisierungs-Hardware verwenden andere Module wie beispielsweise Sensoren und Aktoren bzw. SPS- und I/O-Baugruppen. Die Disziplin Automatisierungs-Software wiederum verwendet für die Informationsverarbeitung in der Steuerungssoftware modulare Funktionsbausteine. Für die Dateneingabe und Datendarstellung in der Bediensoftware werden graphische Bausteine verwendet.

6.3 Anforderungen an das Engineering

Eine generelle Zielsetzung für das Engineering von Produktionsanlagen lautet Steigerung der Wettbewerbsfähigkeit durch Produktivitätssteigerung.

Die Aufgabenstellungen, die sich hierbei ergeben, sind:

- Qualität sichern
- Erstellungszeit verkürzen bzw. kürzere Projektlaufzeiten
- Kosten reduzieren

Neben der Verkürzung der Projektlaufzeiten und der Reduzierung der Kosten steht auch der Wunsch nach einer besseren Kalkulierbarkeit dieser Größen im Mittelpunkt des Interesses, insbesondere im Sonderanlagenbau.

Zur näheren Betrachtung der Aufgabe 'Qualität sichern' wurden folgende wesentliche Aspekte der Qualitätssicherung in [bertrand-meyer88] formuliert:

- **Korrektheit**
  Korrektheit ist die Fähigkeit von Produkten, ihre Aufgabe exakt zu erfüllen, wie sie durch Anforderungen und Spezifikationen definiert ist.
- **Robustheit**
  Robustheit heißt die Fähigkeit von Systemen, auch unter außergewöhnlichen Bedingungen zu funktionieren.
- **Erweiterbarkeit**
  Erweiterbarkeit bezeichnet die Leichtigkeit, mit der Produkte an Spezifikationsänderungen angepasst werden können.
- **Wiederverwendbarkeit**
  Die Wiederverwendbarkeit von Produkten ist die Eigenschaft, ganz oder teilweise für neue Anwendungen wiederverwendet werden zu können.

- **Kompatibilität**
  Kompatibilität ist das Maß der Leichtigkeit, mit der Produkte mit anderen verbunden werden können.
- **Effizienz**
  Effizienz ist ökonomischer Nutzen von Ressourcen.
- **Portabilität**
  Portabilität ist das Maß der Leichtigkeit, mit der Produkte auf verschiedene Umgebungen übertragen werden können.
- **Verifizierbarkeit**
  Verifizierbarkeit ist das Maß der Leichtigkeit, mit der Abnahmeprozeduren und Prozeduren zur Fehlererkennung und -verfolgung während der Validierungs- und Betriebsphase erzeugt werden können.
- **Integrität**
  Integrität ist die Fähigkeit von Systemen, ihre verschiedenen Komponenten (z.B. Daten, Dokumente) gegen unberechtigte Zugriffe und Veränderungen zu schützen.
- **Benutzerfreundlichkeit**
  Benutzerfreundlichkeit ist die Leichtigkeit, mit der die Benutzung von Systemen, ihre Bedienung, das Bereitstellen von Eingabedaten, die Auswertung der Ergebnisse und das Wiederaufsetzen nach Benutzerfehlern erlernt werden kann.

6.3.1 Herausforderung für das Engineering

Die Herausforderung für das Engineering, eine Produktivitätssteigerung zu erzielen, erfordert eine ganzheitliche Betrachtung unter folgenden Gesichtspunkten.

Die organisatorische Herausforderung besteht in der Optimierung der Projektabwicklung.

Die technische Herausforderung besteht in der Definition einer flexibel wiederverwendbaren Systemarchitektur.

Die Kennzeichen einer optimierten Projektabwicklung sind dabei Durchgängigkeit in der Projektabwicklung und Disziplin-Integration.

Die Durchgängigkeit zielt in die Richtung, den Projektabwicklungsprozess durchgängig über seine einzelnen Phasen zu gestalten. Daher sollten z.B. Methodik (objektorientiert) und Notation (UML) phasenübergreifend eingesetzt werden bei der Anforderungsanalyse (use-case-Modell), dem System-Design (konzeptionelles / architektonisches System-Modell), dem SW-Design (Implementierungsmodell), der Software-Realisierung und dem Software-Test

Die Disziplin-Integration zielt in Richtung einer verbesserten Integration der verschiedenen Tätigkeiten innerhalb einer Phase. Als Beispiel kann hier genannt werden, dass sowohl in der Disziplin Datentechnik als auch in der Disziplin Steuerungstechnik mit UML gearbeitet werden soll.

Die flexible Wiederverwendbarkeit wird durch eine **zweckmäßige modulare Systemarchitektur** erreicht, die gekennzeichnet ist durch:

- **Zerlegbarkeit**
  Ein Problem wird in verschiedene Teilprobleme zerlegt, deren Lösungen jeweils getrennt gesucht werden können. Hierdurch wird die anfängliche Komplexität reduziert.
- **Kombinierbarkeit und Erweiterbarkeit**
  Es muss sicherstellt sein, dass vorhandene Module bei der Entwicklung neuer Systeme wiederverwendet, bei Bedarf erweitert und bei Bedarf zu höherwertigen Funktionen zusammengefügt werden können.
- **Autonomie**
  Autonomie bedeutet:
  - möglichst wenige Schnittstellen, d. h., ein Modul sollte mit möglichst wenig anderen kommunizieren
  - möglichst lose Kopplung bzw. schmale Schnittstellen, d. h., Module sollten so wenig Information wie möglich untereinander austauschen
  - explizite Schnittstellen, d. h., die Außenbeziehung muss deutlich gekennzeichnet sein
  - Isolation von Fehlern und Störungen, d. h., deren Auswirkungen auf das Modul beschränken und deren Ausbreitung begrenzen
  - gute Testbarkeit

Die Zerlegung einer Aufgabenstellung in Teilaufgaben, deren Teil-Lösungen nicht zur Lösung einer möglichst großen Anzahl weiterer Aufgabenstellungen in einem anderen Kontext beitragen können, ist beispielsweise nicht zielführend aus dem Blickwinkel der Kombinierbarkeit einer zweckmäßigen Modularität.

Andererseits ist ein Überfrachten eines Moduls mit flexibler Parametrierbarkeit, um eine Wiederverwendbarkeit mit hohem funktionalem Abdeckungsgrad für möglichst viele Modul-Varianten zu erreichen, nicht zielführend aus dem Blickwinkel der Autonomie, und der dabei geforderten schmalen Schnittstelle und guten Testbarkeit. Darüber hinaus ist es in der Regel nur begrenzt vorhersehbar, welche Modul-Varianten zukünftig benötigt werden.

Die grundsätzliche Forderung nach einfacher, kostengünstiger Realisierbarkeit und Anwendbarkeit muss ebenfalls Berücksichtigung finden.

Diese teilweise widersprüchlichen Anforderungen müssen bei der Entwicklung der optimalen Systemarchitektur gegeneinander abgewogen werden.

Eine zweckmäßige modulare Systemarchitektur erreicht dann die genannten Zielsetzungen

- Qualität sichern durch Einsatz erprobter Module
- Kosten mindern und Erstellungszeit kürzen durch Einsatz vorhandener Module
- Optimalen Abdeckungsgrad wählbar gestalten durch anpassbare und erweiterbare Module

6.4 Stand der Technik bei teamtechnik

Die technische Herausforderung einer zweckmäßigen modularen Systemarchitektur wurde bei teamtechnik frühzeitig erkannt. Dabei entstand als Systemplattform für den Maschinen- und Anlagenbau die prozessmodulare Plattform TEAMOS (siehe Abb. 6.2).

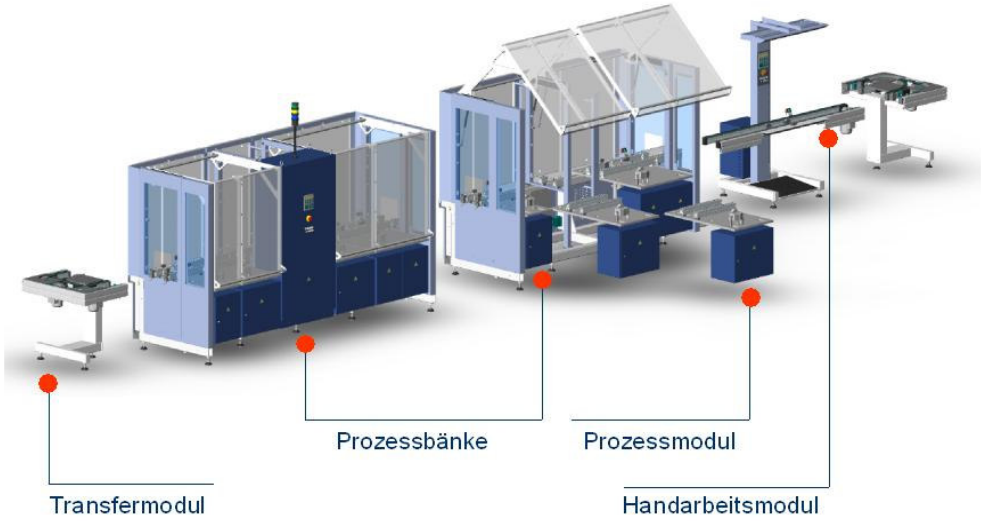


Abb. 6.2: Prozessmodulare Plattform TEAMOS

6.4.1 Erzielung der technischen Anforderungen bei TEAMOS

Die technische Herausforderung der Wiederverwendbarkeit wird durch einen hohen Grad der **Standardisierung** erreicht.

Beispiele hierfür sind

- standardisierte Anlagenstruktur mit Prozessmodulen
- standardisierte Bedienung
- standardisiertes Fehler- / Störungs- / Meldungsmanagement
- standardisierte Schnittstellen zu
  - Betriebsdatenmanagement (BDE)
  - Produktdatenmanagement (PDM)
  - Fertigungssteuerung / Anlagensteuerung (MES / SCADA)
  - Produktionsplanung (PPS)
- Abstraktion von gemeinsamen Eigenschaften, d. h. standardisierte Zustände und standardisiertes Verhalten
  Beispiele für standardisierte Zustände sind:
  - Standardisierte Betriebsarten ('Einricht-Betrieb', 'Schritt-Betrieb', 'Automatik-Betrieb', usw.) für alle Anlagenteile
  - Standardisierte Zustände ('Grundstellung', 'Arbeitsstellung') für alle Zylinderarten
  - Standardisierte Zustände ('offen', 'geschlossen mit Teil', 'geschlossen ohne Teil') für alle Greiferarten

Ein Beispiel für standardisiertes Verhalten sind standardisierte Basisabläufe, z. B. ein standardisierter Werkstückträgerdurchlauf in allen Prozessmodulen

Die technische Herausforderung der Flexibilität wird durch **Parametrierbarkeit**, **Skalierbarkeit und individuelle konstruktive Anpassungen im Kundenprojekt** erreicht.

Beispiele hierfür sind:

- Der Zylinder-Ansteuerungs-Baustein ist ein parametrierbarer SW-Bausteine und besitzt eine Parameterauswahl 'simultan in Grundstellung fahren' oder 'über eine Schrittkette gesteuert in Grundstellung fahren'.
- Die TEAMOS-Prozessbank ist eine skalierbare Software-Baugruppe mit einer wählbaren Anzahl der Schutzbereiche.

6.4.2 Das Prozessmodul

Auf neutralen TEAMOS-Prozessträgern werden Prozesse geplant und konstruiert (siehe Abb. 6.3).

Der Prozessträger enthält viele Basis- Eigenschaften und -Funktionalitäten (mechanisch, elektrisch, in Software). Diese werden vom Prozessträger an das Prozessmodul vererbt.

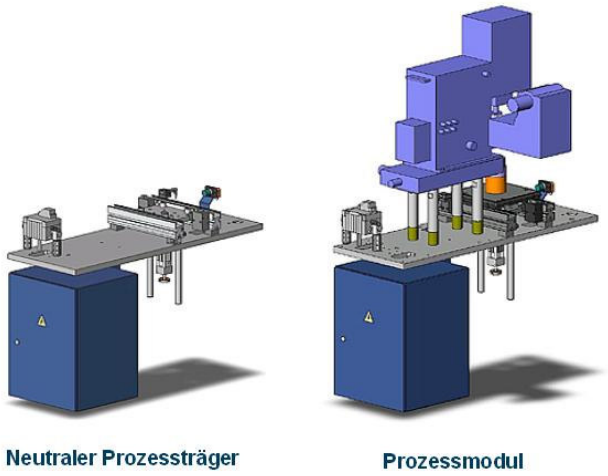


Abb. 6.3: Neutraler Prozessmodulträger und Prozessmodul

Standardisierte Prozessmodule

Für wiederkehrende Aufgabenstellungen existieren standardisierte Prozessmodule als mechatronische Einheiten, mit vorhandenen standardisierten mechanischen Konstruktionen, mit vorhandenen standardisierten elektrischen Konstruktionen und mit vorhandenen Standard-Software-Modulen.

Die flexible Wiederverwendbarkeit von erprobten Prozessmodulen ermöglicht eine schnelle projektspezifische (produkt-, anlagen- und kundenspezifische) Anpassung sowie kurze Inbetriebnahme- und Lieferzeiten.

Abb. 6.4 zeigt typische wiederverwendbare Prozessmodule.

Abb. 6.4: : Flexible wiederverwendbare Prozessmodule

Der teamtechnik-Prozesspool

Der teamtechnik-Prozesspool besteht aus einer Datenbank mit standardisierten und kundenspezifischen Prozessen.

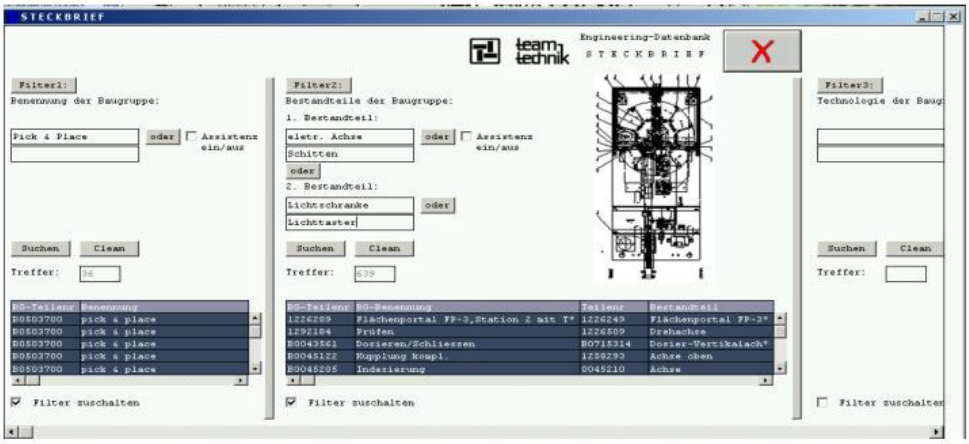


Abb. 6.5 zeigt einen Screenshot der Datenbank-Benutzeroberfläche.
Abb. 6.6 illustriert die Prozesspool-Philosophie 'Konfigurieren statt Konstruieren'.

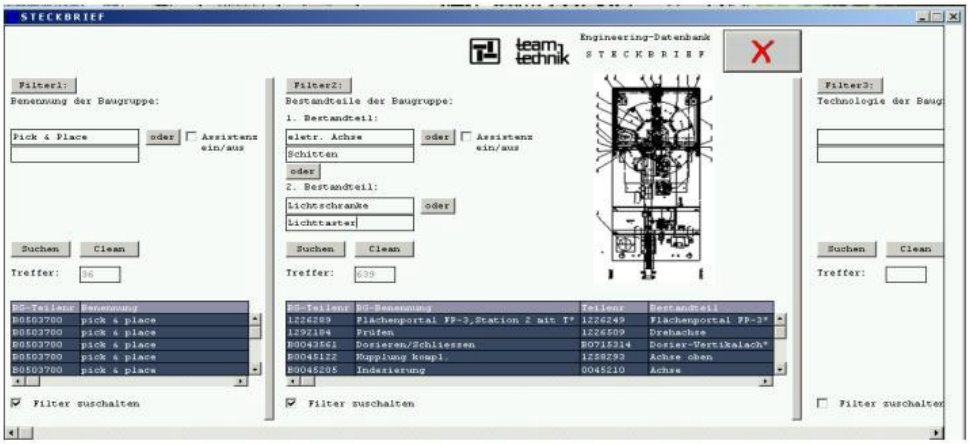


Abb. 6.5: Bildschirmmaske zur Auswahl von Prozessmodulen aus dem teamtechnik Prozesspool

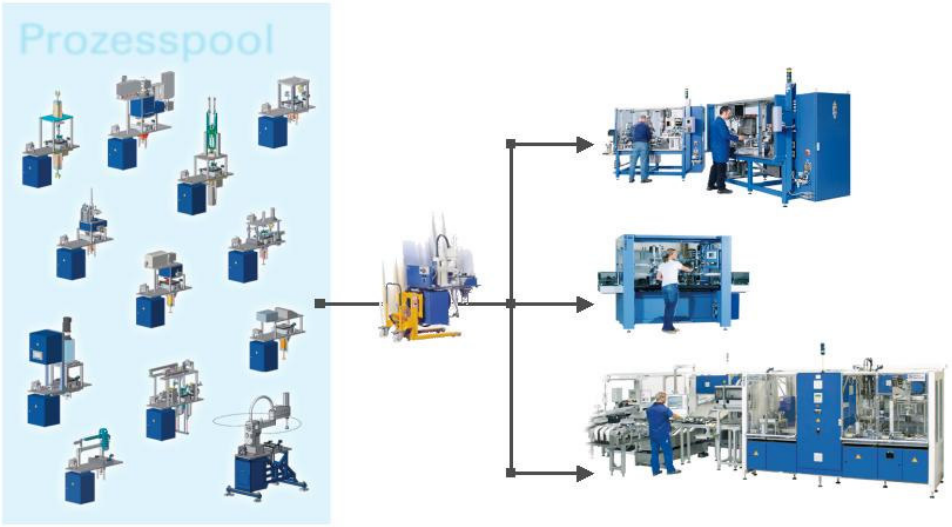


Abb. 6.6: Konfigurieren statt Konstruieren

6.5 Objektorientiertheit und IEC 61131-3

Tab. 6.1 zeigt einen Vergleich der klassischen SPS-Programmierung (IEC61131-3) mit der objektorientierten Programmierung (OOP).

Tab. 6.1: Vergleich klassischer SPS-Programmierung mit objektorientierter Programmierung

Methode Beispiel	IEC61131-3 (ST)	IEC61131-3 (AS / S7Graph)	3S-erweiterte IEC61131 (ST)	OOP (z.B. C++)
Abstraktion	POE	POE	POE	Class
Instanziierungsgranularität	FB	FB	FB	Class
Datenkapselung	Nein	Nein	Nein	Ja
Vererbung	Nein	Nein	Ja	Ja
Polymorphismus	Nein	Nein	Ja	Ja
Dynamische Instanziierung	Nein	Nein	Nein	Ja

Gemeinsamkeit von IEC61131-3 und OOP

- Abstraktion und Instanziierung
  Die Trennung von Definition (class bzw. FB) und Verwendung (Instanz) ist sowohl bei den IEC 61131-3 Sprachen, als auch bei objektorientierten Sprachen möglich.

Unterschiede zwischen IEC61131-3 und OOP

- **Datenkapselung**
  ist bei IEC61131-3 durch entsprechende Programmier-Vereinbarungen lösbar.
- **Vererbung**
  kann bei IEC61131-3 teilweise durch andere Methoden (z. B. Delegation) nachgebildet werden; dies muss jedoch bereits beim Design der Basis-Funktionalität berücksichtigt werden.
- **Polymorphismus**
  Das dynamische Binden von Funktionen ist in der IEC 61131-3 nicht möglich und nicht nachbildbar.
- **Dynamische Instanziierung** wird in der Steuerungstechnik in der Regel nicht benötigt.

Objektorientiertheit betrifft jedoch nicht nur die Begriffe und Konzepte wie Klassen und deren Instanziierung in Objekten, Attribute, Methoden, Abstraktion, Datenkapselung, Vererbung, Polymorphismus, sondern das Grundprinzip der Objektorientiertheit umfasst auch die folgenden Aspekte:

- Ein System besteht aus Objekten, die eigenverantwortlich das tun, was sie auf Grund ihrer Informationen (eigene Zustände, Signale aus der Umwelt) selbständig tun können
- Objekte können Teile ihrer Arbeit an andere Objekte delegieren
- Objekte können bei Bedarf mit anderen Objekten zusammenarbeiten
- Objekte sind letztlich Modelle der realen Welt

Somit erfüllt Objektorientiertheit die Anforderungen an eine zweckmäßige Modularität

- natürliche Struktur
- Eigenverantwortlichkeit (vgl. modulare Autonomie)
- Zusammenarbeit und Delegation (vgl. modulare Kombinierbarkeit, modulare Zerlegbarkeit)

womit die technische Herausforderung 'flexible Wiederverwendbarkeit' erreicht wird.

Die objektorientierte Methodik erfordert dabei

- das Aufspüren von Objekten, deren Gemeinsamkeiten und deren Beziehungen untereinander
- die formale Spezifikation der Objektmerkmale (Attribute) und der Operationen (Methoden), die für das Objekt verfügbar sind, basierend auf der Theorie der abstrakten Datentypen
- das Aufdecken / Verhindern versteckter Schnittstellen

6.6 Anforderungen an UML und an UML-Tools

6.6.1 Was ist UML?

UML (Unified Modelling Language) ist eine **vereinheitlichte Sprache für die graphische Modellierung** von Systemen.

Der Vorteil gegenüber proprietären graphischen Beschreibungsmethoden (z.B. mit Hilfe von Microsoft Visio) ist, dass UML projektphasenübergreifend, disziplinübergreifend und firmenübergreifend eingesetzt werden kann.

Durch die vereinheitlichte Sprache wird der Informationsaustausch zwischen Projektphasen, Disziplinen und Firmen vereinfacht. Dadurch unterstützt UML die organisatorischen Herausforderungen 'Durchgängigkeit im Projektabwicklungsprozess' und 'Disziplin-Integration'.

Nachteile gegenüber proprietären Beschreibungsmethoden sind, dass sich spezielle Modelleigenschaften einfacher mit einem spezialisierten Modellierungswerkzeug beschreiben lassen und ein zusätzlicher Aufwand für die Einführung und für das Erlernen einer neuen Modellierungssprache entsteht.

Der Einsatz von Standards, wie UML, führt generell zu weniger Freiheitgraden und

weniger Gestaltungsmöglichkeiten für den Anwender der Standards. Je nach Blickwinkel kann dies als Nachteil oder als Vorteil betrachtet werden. Erfahrene Entwickler werden möglicherweise in ihren Ausdrucksmöglichkeiten und in ihrer Produktivität eingeschränkt. Auf der anderen Seite wird für eine gute Produktivität eines Team eine allgemein verständliche und präzise Ausdrucksform benötigt.

6.6.2 Strukturdiagramme

Zur heutigen Situation in Bezug auf Strukturdiagramme im Maschinen- und Anlagenbau kann gesagt werden, dass keine firmenübergreifend standardisierte Darstellungsart hierfür existiert. Teilweise wird die Modul-Struktur firmenspezifisch „semiformal“ spezifiziert, z.B. mit Hilfe von Zeichenprogrammen. Diese Diagramme stellen in der Regel nicht Architekturmodelle dar, sondern beschreiben die Struktur konkreter Anlagen, sie entsprechen daher eher einem UML-Objektdiagramm, jedoch erweitert um Echtzeit-Angaben (z. B. Tasks).

UML bietet die Strukturdiagramm-Arten Klassendiagramm (vgl. Abb. 6.7) und Objektdiagramm (vgl. Abb. 6.8)

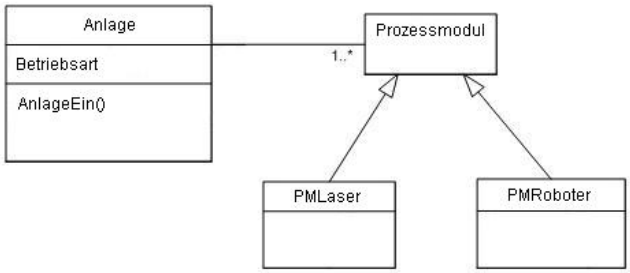


Abb. 6.7: Klassendiagramm

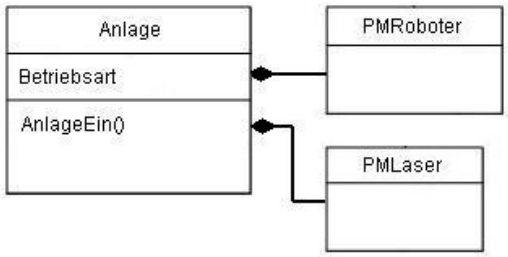


Abb. 6.8: Objektdiagramm

Bewertung der UML-Strukturdiagramm-Arten

- Klassendiagramm
  - Anforderungen sind im UML-Standard vollständig abgedeckt.
- Objektdiagramm
  - Anforderung: Objektdiagramm erweitern, um das Echtzeitverhalten definieren zu können für die Zuordnung zu einer Task.

6.6.3 Verhaltensdiagramme

Verhaltensdiagramme im Maschinen- und Anlagenbau

Zur Beschreibung des Maschinen- und Anlagenverhaltens werden häufig die Diagramm-Arten Weg-Zeit-Diagramm, Funktionsplan und Zustandsgraph verwendet.

Weg-Zeit-Diagramm

Weg-Zeit-Diagramme werden von den Maschinenbau-Konstrukteuren dazu verwendet, Aussagen über das Gesamt-Zeitverhalten einer Anlage und über die zu erwarteten Taktzeiten zu erhalten. Weg-Zeit-Diagramme helfen dabei, frühzeitig mögliche Performance-Probleme der Anlage aufzudecken. Abb. 6.9zeigt ein Weg-Zeit-Diagramm, das mit [comos] erstellt wurde.

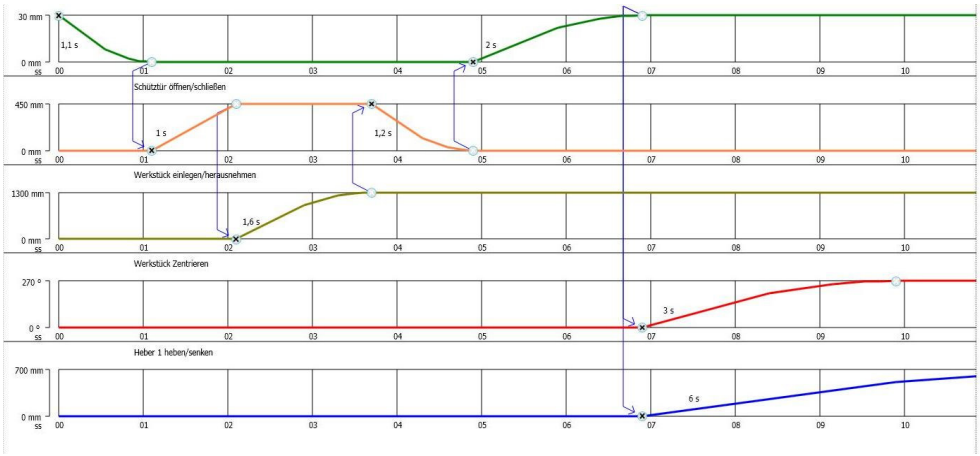


Abb. 6.9: Weg-Zeit-Diagramm [comos]

Funktionspläne

Funktionspläne (nach DIN EN 40719-6 bzw. DIN EN 60848) beschreiben die Feinabläufe in einer Anlage; und da sie eine disziplin-übergreifende Darstellungsform besitzen, werden sie häufig als Grundlage für inter-disziplinäre Diskussionen verwendet. Im Gegensatz zu den Weg-Zeit-Diagrammen, die lediglich ein einziges, typisches, dynamisches Verhalten der Anlage beschreiben, definieren Funktionspläne das systemische Verhalten. Jeder Durchlauf-Variante lässt sich ein Weg-Zeit-Diagramm zuordnen.

Abb. 6.10 zeigt einen Funktionsplan, der mit Hilfe des Funktionsplan-Editors der teamtechnik GmbH erstellt wurde.

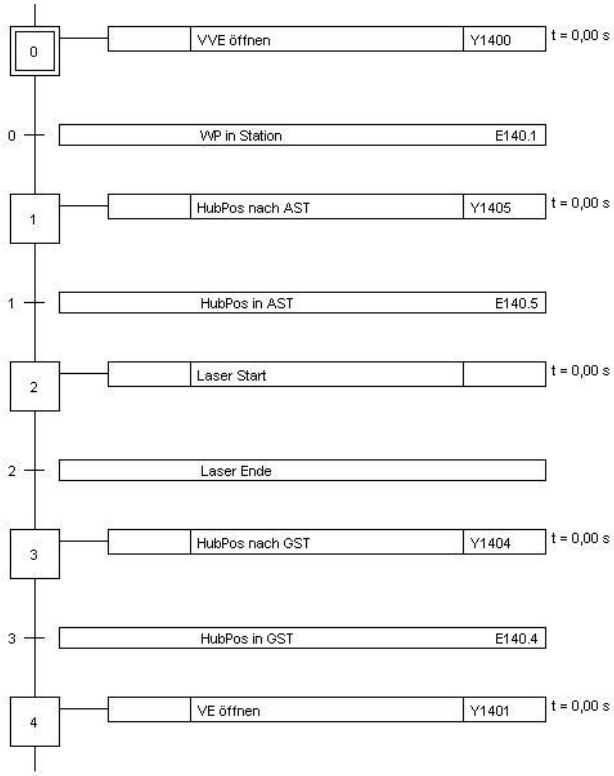


Abb. 6.10: Funktionplan

Zustandsgraph

Zustandsgraphen (vgl. Abb. 6.11) beschreiben Zustände und Zustandsübergänge.

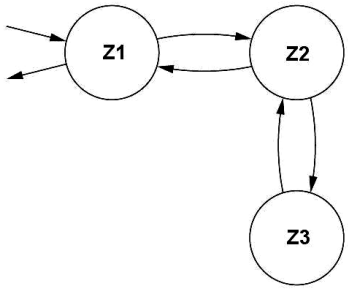


Abb. 6.11: Zustandsgraph

Verhaltensdiagramme in UML

UML bietet die Verhaltensdiagramm-Arten Sequenzdiagramm, Zeit-Diagramm, Aktivitätsdiagramm und Zustandsdiagramm.

Das Sequenzdiagramm (vgl. Abb. 6.12 [wikipedia]) beschreibt die Interaktion zwischen Objekten als funktionale Abfolge.

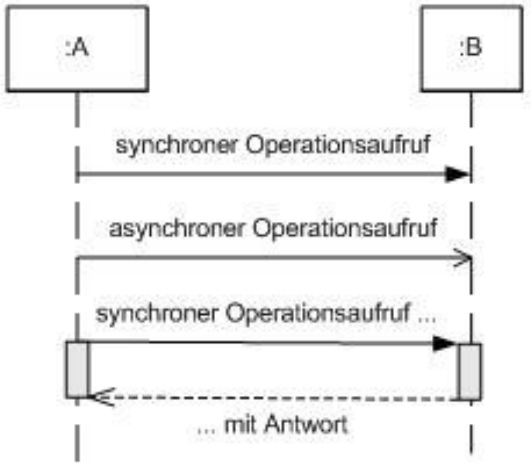


Abb. 6.12: Sequenzdiagramm

Das Zeit-Diagramm (vgl. Abb. 6.13 [wikipedia]) beschreibt die Interaktion zwischen Objekten als zeitlichen Ablauf.

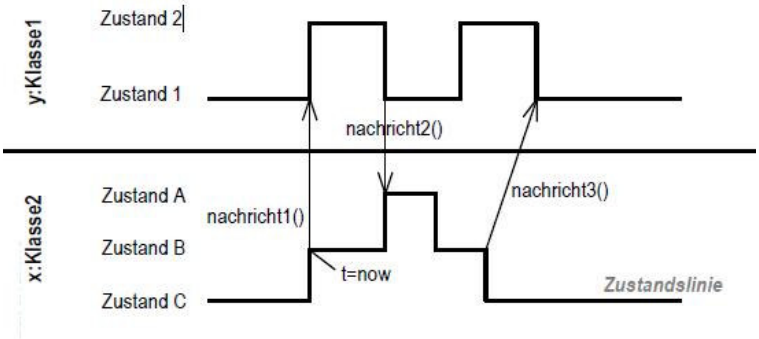


Abb. 6.13: Zeit-Diagramm

Das Aktivitätsdiagramm (vgl. Abb. 6.14 [martin-fowler04]) beschreibt das prozedurale und parallele Verhalten von Objekten.

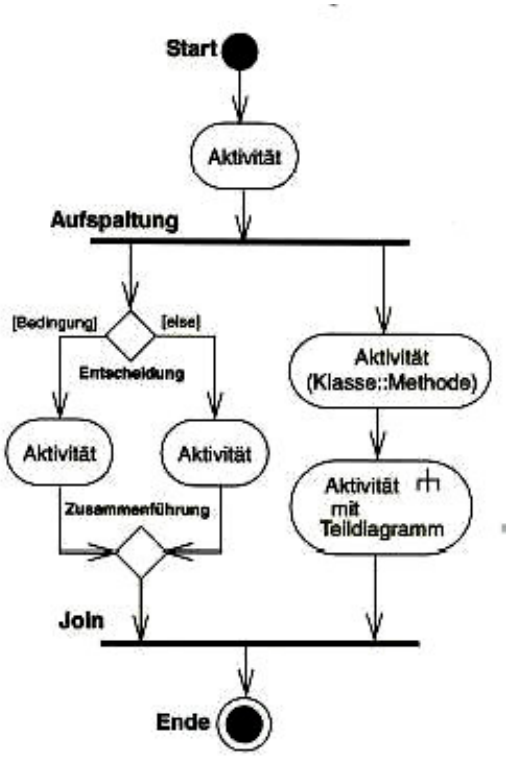


Abb. 6.14: Aktivitätsdiagramm

Das Zustandsdiagramm (vgl. Abb. 15 [martin-fowler04]) beschreibt Zustände und Zustandsübergänge eines Objektes.

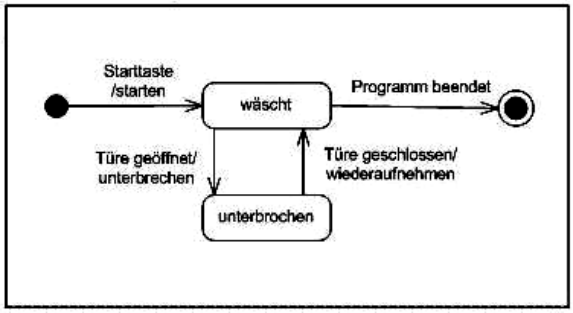


Abb. 15: Zustandsdiagramm

Bewertung der UML-Verhaltensdiagramm-Arten

Die Ausdrucksmöglichkeiten eines Weg-Zeit-Diagramms sind in UML in keiner Interaktionsdiagramm-Art verfügbar. Daher besteht die Anforderung, das Sequenzdiagramm oder das Zeit-Diagramm in Richtung Weg-Zeit-Diagramm zu erweitern.

Das Aktivitätsdiagramm sollte dahingehend erweitert werden, zusätzlich Interaktionen mit der Umwelt (E/A) und Echtzeitverhalten (Wartezeiten) angeben zu können.

Die Anforderungen an das Zustandsdiagramm werden im UML-Standard vollständig abgedeckt.

Weitere Anforderungen und Wünsche an UML-Tools

Die Eigenschaften eines Objektes werden in UML einerseits in den Aktivitätsdiagrammen als prozedurale und parallele Abfolge von Aktivitäten beschrieben und anderseits als Abfolge von Interaktionen mit anderen Objekten. Zwischen diesen Darstellungsarten sollte umgeschaltet werden können, wobei jede Durchlauf-Variante im Aktivitätsdiagramms ein zugehöriges Interaktionsdiagramm - und mit der weiter oben genannten Ergänzung, ein zugehöriges Weg-Zeit-Diagramm - erzeugt.

Zur Erhöhung der Durchgängigkeit in der Projektabwicklung besteht der Wunsch einer Inbetriebnahme auf Modellebene. Dazu muss ein UML-Tool online-fähig und inbetriebnahme-tauglich sein, z. B. mit Beobachtungs- und Debugging-Funktionen (Forcen, Breakpoints, Trace, etc.).

Weiterhin müssen Investitionen geschützt werden. Dazu muss die Fähigkeit zur schrittweisen Migration existieren und bestehende SPS-Programmteile müssen einbindbar sein.

Für die Dokumentation wäre es wünschenswert, wenn mehrere Diagramm-Arten in einem Diagramm dargestellt werden könnten, um eine kompakte Darstellung in Form einer Übersicht zu ermöglichen. In dieser Übersicht könnte dann die Struktur, das systemische Verhalten und ein oder mehrere typische Interaktionsverhalten dargestellt werden.

Weiterhin sollte die Fähigkeit, Code aus dem Modell zu generieren, gegeben sein. Die Code-Generierung sollte für marktgängige SPS-Systeme wählbar sein.

Zur Integration mit anderen Tools wird ein offenes Interface benötigt. Beispiele für Tools, für die eine Integration wünschenswert wäre, sind:

- Tools für das Versions-Management
- Tools für das Konfigurations-Management (Varianten-Management)
- Tools für das Modul-Management (Anzeigen, Suchen über Modul-Eigenschaften, Kombinieren, Parametrieren). Tools für das Modul-Management sind derzeit am Markt in Entwicklung, z. B. [eec] und [comos]. Grundlage für einen sinnvollen Einsatz dieser Tools ist jedoch eine zielführende Modularität. Daraus ergibt sich eine Wechselwirkung: Einerseits werden Tools für das Management der modularen Systemarchitektur benötigt, andererseits wird eine zielführende Modularität der Systemarchitektur von den Tools gefordert.

- Tools für andere Projektphasen (z.B. Tool für das Anforderungsmanagement)
- Entwicklungstools weiterer SPS-Hersteller

Eine weitere Forderung ist die Multi-User-Fähigkeit in der Designphase und im Online-Betrieb.

6.7 Weitere Anforderungen an Programmierverfahren

Eine besondere Herausforderung für die Steigerung der Wiederverwendung von Modulen ist durch die Anforderung gegeben, dass oft feingranulare Anpassungen der Modulimplementierung bei der Anwendung im konkreten Einzelfall erforderlich sind. Im Maschinen- und Anlagenbau erwächst diese Anforderung etwa aus der Tatsache, dass unterschiedlichste Werkstücke mit den prinzipiell selben Prozessmodulen (PM), z. B. PM Schrauben, PM Kleben, PM Einpressen, etc bearbeitet werden sollen. Auch im Prinzip vergleichbare Werkstücke, wie beispielsweise Einspritzventile, die in unterschiedlichen Produktionsanlagen montiert und geprüft werden sollen, können in ihrem Detailaufbau so unterschiedlich gestaltet sein, dass mehr oder weniger große Anpassungen der Prozessmodule erforderlich sind, die mitunter so feingranulare Änderungen erfordern, dass bei der heute üblichen SPS-Programmierung nach IEC 61131-3 beispielsweise Anpassungen von einzelnen Codezeilen in vorhandenen Software-Bausteinen erforderlich werden.

Die Varianten-Anpassungen, die hierbei erforderlich werden, sind häufig feingranularer, als die Ausprägung von Modulvarianten, die mit den in objektorientierten Programmierverfahren üblichen Klassen-Ableitungshierarchien praktikabel abgebildet werden können.

Erschwerend kommt hinzu, dass diese notwendigen werkstück-spezifischen Anpassungen nicht im voraus bei der ursprünglichen Erstellung der Standard-Prozessmodule und der zugehörigen Software bekannt sind, da zu diesem Zeitpunkt noch gar nicht in vollem Umfang bekannt sein kann, welches Werkstückspektrum später mit diesem Prozessmodul gefertigt werden soll. Selbst wenn eine große Anzahl von Modulvarianten frühzeitig bekannt sein sollte, ist es oft nicht sinnvoll, viele mögliche Detail-Implementierungs-Varianten gemeinsam in entsprechend parametrierbare Modulvorlagen zu packen. Dies würde zu Modulen führen, die zu groß, zu komplex und somit in der Praxis kaum noch beherrschbar wären.

Eine weitere Besonderheit ist beispielsweise im Maschinen- und Anlagenbau für die Automotive-Branche, aber auch in anderen Branchen, dass die zu bearbeitenden Werkstücke oft mehrmals ein "Last-Minute Re-Design" erfahren, wenn die Produktionsanlage und die zugehörigen Prozessmodule, auf der das Werkstück gefertigt werden soll, schon fertig konstruiert und einschließlich vorbereiteter Software inbetriebnahmebereit aufgebaut sind. Dass die durch derartige "Last-Minute Re-Designs"

des Werkstücks erforderlichen Modulanpassungen in Konstruktion und Software sehr schnell und pragmatisch umgesetzt werden müssen, erklärt sich von selbst.

Moderne Steuerungs-Programmierverfahren und deren zugehörige Modellierungs- und Programmier-Tools müssen also auch diesen besonderen Anforderungen aus der Praxis gerecht werden, um bei den Praktikern Akzeptanz zu finden.

6.8 Schlussbetrachtung

Neben den genannten Möglichkeiten sollten weitere Möglichkeiten gesucht werden, die flexible Wiederverwendbarkeit von Modulen im Maschinen- und Anlagenbau steigern; etwa die Nutzung existierender offener Standards, z.B. die Nutzung von PLCopen, oder die Definition weiterer offener Standards.

Wünschenswert wäre beispielsweise die Definition eines steuerungstechnik-hersteller-unabhängigen Software-Architektur-Standards für SPS-Software (vgl. [OMAC]) jedoch zusätzlich mit zumindest auf der obersten Architekturebene branchenunabhängigen, genormten Funktionalitäten, Funktionspaketen (Betriebsartenverwaltung, Bedienung, Fehler-, Störungs- und Meldungshandling, SPS-SPS-Kommunikation, SPS-MES-Kommunikation), mit genormten Schnittstellen zwischen Funktionen und mit Varianten von Funktionen.

Die Einführung von Objektorientiertheit für die SPS-Programmierung und UML für die Modellierung in einem Maschinen- und Anlagenbau-Unternehmen erfordert die Betrachtung weiterer Aspekte und die Schaffung der entsprechenden Voraussetzungen: Qualifizierung der Mitarbeiter, Akzeptanz herbeiführen durch angemessene Einführungskonzepte für diese in der Steuerungstechnik neuen Methoden und ein Change-Management im Projektabwicklungsprozess.

Für die genannten Problemstellungen müssen, sofern sie noch nicht im aktuellen Forschungsprojekten [uml-tool] oder anderen Forschungsprojekten betrachtet wurden, praktikable Lösungen gefunden werden.

Abschließend bleibt festzustellen: Erst im praktischen Einsatz werden sich die Anforderungen an Objektorientiertheit und UML in vollem Umfang erkennen lassen.

6.9 Referenzen und weiterführende Literatur

[bertrand-meyer88] Bertrand Meyer, Objektorientierte Softwareentwicklung, Carl Hanser Verlag München Wien, 1990

[martin-fowler04] Martin Fowler, UML konzentriert, Addison-Wesley, 2004

[comos] Comos Industry Solutions GmbH, Schwelm (http://www.comos.de) Comos ME Designer

[eec]	EPLAN Software & Service GmbH & Co. KG, Monheim am Rhein (http://www.eplan.de) Eplan Engineering Center
[OMAC]	Organisation for Machine Automation and Control (http://www.omac.org)
[plc-open]	hersteller- und produkt-unabhängige Organisation, die sich mit der Definition von Standards im Bereich industrieller Steuerungen befasst (http://www.plcopen.org)
[uml-tool]	Universität Kassel, FB Informatik, Fachgebiet Eingebettete Systeme Forschungsprojekt „Automatische Codegenerierung aus der UML für die IEC 61131-3 in eingebetteten Automatisierungssystemen unter besonderer Berücksichtigung der Systemarchitektur“
[wikipedia]	freie Online-Enzyklopädie (http://www.wikipedia.de)

- Hans-Peter Wiendahl, Detlef Gerst, Lars Keunecke (Hrsg.), Variantenbeherrschung in der Montage, Springer Verlag, 2004, Kap. 5.1, Max Roßkopf, Hubert Reinisch Prozessmodulare Gestaltung von Produktionssystemen

- Beckhoff Industrie Elektronik (Hrsg.), PC-Control, Ausgabe 2 / 2004, Seite 16, Prof. Dr.-Ing. Birgit Vogel-Heuser, Daniel Witsch, UML für die Steuerungsprogrammierung

- atp international, Ausgabe 1 / 2004, Seite 23, Uwe Katzke, Birgit Vogel-Heuser und Katja Fischer Analysis and state of the art of modules in industrial automation, Oldenbourg Industrieverlag, München

6.10 Glossar

AS	- Ablaufsprache
FB	- Funktionsblock
MES	- Manufacturing Execution System
OO	- Objektorientierung
OOP	- Objektorientierte Programmierung
POE	- Programmorganisationseinheit
SCADA	- Supervisory Control and Data Acquisition
SPS	- Speicherprogrammierbare Steuerung
TEAMOS	- Prozessmodulare Anlagenplattform der Firma teamtechnik
UML	- Unified Modelling Language