Programmiertechnik für die Mechatronik

Autor(en): **Maier, G. E.**

Objekttyp: **Article**

Zeitschrift: **Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association Suisse des Electriciens, de l'Association des Entreprises électriques suisses**

Band (Jahr): **80 (1989)**

Heft 1

PDF erstellt am: **04.06.2024**

Persistenter Link: https://doi.org/10.5169/seals-903621

Nutzungsbedingungen

Die ETH-Bibliothek ist Anbieterin der digitalisierten Zeitschriften. Sie besitzt keine Urheberrechte an den Inhalten der Zeitschriften. Die Rechte liegen in der Regel bei den Herausgebern.
Die auf der Plattform e-periodica veröffentlichten Dokumente stehen für nicht-kommerzielle Zwecke in Lehre und Forschung sowie für die private Nutzung frei zur Verfügung. Einzelne Dateien oder Ausdrucke aus diesem Angebot können zusammen mit diesen Nutzungsbedingungen und den korrekten Herkunftsbezeichnungen weitergegeben werden.
Das Veröffentlichen von Bildern in Print- und Online-Publikationen ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Die systematische Speicherung von Teilen des elektronischen Angebots auf anderen Servern bedarf ebenfalls des schriftlichen Einverständnisses der Rechteinhaber.

Haftungsausschluss

Alle Angaben erfolgen ohne Gewähr für Vollständigkeit oder Richtigkeit. Es wird keine Haftung übernommen für Schäden durch die Verwendung von Informationen aus diesem Online-Angebot oder durch das Fehlen von Informationen. Dies gilt auch für Inhalte Dritter, die über dieses Angebot zugänglich sind.

Ein Dienst der *ETH-Bibliothek*
ETH Zürich, Rämistrasse 101, 8092 Zürich, Schweiz, www.library.ethz.ch

http://www.e-periodica.ch

Programmiertechnik für die Mechatronik

G. E. Maier

Ausgehend von den Anforderungen, welche an die Software für Mechatroniksysteme gestellt werden, wird der heutige Stand von Werkzeugen (Programmiersprachen, Echtzeitbetriebssysteme) und Lösungskonzepten aufgezeigt. Neben CASE-Werkzeugen, Exception-Behandlung und vollgraphischen Bedienungsschnittstellen wird ein Forschungsprojekt beschrieben, das zum Ziele hat, durch anwendernahe graphische Programmierung, den Programmierer möglichst weitgehend von Informatikspezialkenntnissen zu entlasten.

En partant des exigences posées aux logiciels pour systèmes mécatroniques, on montre l'état actuel des outils (langages de programmation, systèmes d'exploitation en temps réel) et des concepts de solutions. A côté des outils CASE, du traitement exceptionnel et des interfaces de conduite entièrement graphiques, un projet de recherche est décrit qui a pour but de décharger le programmeur dans une large mesure de connaissances spéciales en informatique par une programmation graphique bien applicable.

Adresse des Autors

Dr. *Georg E. Maier*, Asea Brown Boveri AG, Forschungszentrum, Abt. Informatik CRBC 2, 5405 Baden-Dättwil.

Ein wesentlicher Faktor für den Erfolg von Mechatroniksystemen ist ihre Flexibilität. Durch Austausch von Programmen kann die Funktionalität eines Systems ohne Anpassung der Hardware innerhalb eines weiten Rahmens verändert werden. Die Software wird mehr und mehr zum entscheidenden Faktor, der die Geschwindigkeit der technischen Entwicklung, z.B. auf dem Gebiet der Robotik, weitgehend bestimmt und begrenzt.

Das Erstellen von Software ist jedoch alles andere als einfach, und Kostenüberschreitungen, mangelhafte Qualität, fehlende Dokumentation und Probleme bei Erweiterungen sind fast an der Tagesordnung. Im Gegensatz zur raschen Entwicklung der Hardwareleistung steigt die Programmierproduktivität nur langsam an. Obwohl Programme fast ohne Aufwand vervielfältigt werden können und im Betrieb kein Verschleiss zu verzeichnen ist, besteht ein wachsender Bedarf nach neuer Software, der nur zu einem kleinen Teil durch Wiederverwendung gestillt werden kann.

Ziel dieses Artikels ist, die Anforderungen an die Programmierung von Mechatroniksystemen aufzuzeigen, den heutigen Stand der angewandten Programmiertechniken zu beschreiben und auf einige Trends in Entwicklung und Forschung einzugehen. Aspekte der Rechnerkommunikation, der künstlichen Intelligenz und Datenbanken können aus Platzgründen nicht berücksichtigt werden.

Anforderungsanalyse

Funktionalität

In der Mechatronik übernimmt die Software je nach Grösse des Systems eine oder mehrere der folgenden Aufgaben, die oft mit den Funktionen des Gesamtsystems zusammenfallen oder sehr eng mit diesen verknüpft sind:

- Erfassen (Ablesen von Sensoren), Aufbereiten (Filtern), Speichern (in einer Datenbank), Verarbeiten (Analyse, Statistik usw.) sowie Ausgabe (Ansteuern von Stellgliedern) von Daten,
- Überwachung, Alarmierung und Protokollierung,
- quasikontinuierliche und ereignisorientierte Steuerungen (Open-Loop) und Regelungen (Closed-Loop),
- Modellierung wichtiger Aspekte des Systems oder seiner Umwelt als Grundlage für die Erfüllung anderer Aufgaben,
- Bedienungsschnittstelle für den Betrieb (Darstellung von Daten, z.B. des Anlagezustands, Bedienung der Anlage, Unterstützung des Bedieners),
- Bedienungsschnittstelle für den Unterhalt (Engineering: On-line-Planung, -Konfiguration, -Simulation und -Programmierung der Anlage[1]).

Hardwarenahe Programmierung

Bei der Datenerfassung und -ausgabe – z.B. beim Ablesen von Sensoren und beim Ansteuern von Stellgliedern – wird auf tiefster Stufe direkt auf Geräteregister zugegriffen, die auf absoluten Adressen im Hauptspeicher oder in einem speziellen Input-Output-Bereich liegen. Dabei müssen Daten auf der Stufe einzelner Bit manipuliert werden können (z.B. Konversion eines 12-Bit-A/D-Wandlerausganges in eine interne 16-Bit-Darstellung und umgekehrt). Weiter müssen spezifische

[1] Beim On-line-Engineering kann das System während des Betriebes verändert werden. On-line-Engineering ist also eine Funktion (Tätigkeit) des Systems, während Off-line-Engineering eine Aufgabe der Umgebung ist, in der das System entwickelt wird.

Timing-Anforderungen eingehalten und Interrupts verarbeitet werden können.

Echtzeit

Im Gegensatz zur kommerziellen Datenverarbeitung muss die Software in der Mechatronik den zeitlichen Anforderungen der betreffenden Anwendung genügen. Harte, d.h. unbedingt einzuhaltende Grenzen sind die Abtastzeiten von quasikontinuierlichen Regelungen und Steuerungen sowie die maximale Reaktionszeit bei der Verarbeitung externer Ereignisse. Eine weitere, harte Grenze für die minimale Rechenleistung kann sich aus der Menge der Daten ergeben, die über einen längeren Zeitraum erfasst, aufbereitet, gespeichert und verarbeitet werden müssen.

Weniger absolut, aber immer noch strenger als in der kommerziellen EDV sind die zeitlichen Anforderungen, die an die Mensch-Maschine-Kommunikation gestellt werden. Die maximalen Antwortzeiten – z.B. die Zeit, die benötigt wird, um auf Abruf den Zustand einer Anlage graphisch darzustellen – müssen typisch im Bereich von 1 bis 3 Sekunden liegen.

Parallelität

Bereits in kleinen Systemen muss die Software nicht nur eine, sondern viele, meist sehr unterschiedliche Aufgaben erfüllen. Mit einem sequentiellen Programm ist dies kaum möglich. Es muss deshalb mit *Multitasking* (parallele Prozesse) gearbeitet werden können.

Software-Engineering

Software-Engineering ist die Ingenieurdisziplin, die sich mit der Spezifikation, dem Entwurf, der Realisierung, dem Test, der Wartung und der Erweiterung von Software befasst. Viele Mechatroniksysteme müssen während ihrer Lebensdauer immer wieder an die Entwicklung oder an spezielle Aufgaben angepasst werden. Die Softwarequalität und die vorhandene Dokumentation bestimmen dann weitgehend, wie leicht Korrekturen, Änderungen und Erweiterungen realisiert werden können und ob die Programme in weiteren Projekten wiederverwendet oder neu entwickelt werden müssen. Angesichts des wachsenden Anteils der Softwarekosten an den Gesamtkosten ist die Wiederverwendbarkeit von Programmteilen von hoher wirtschaftlicher Bedeutung. Neben den Methoden und Werkzeugen, die direkt der Programmerstellung dienen, fallen auch Hilfsmittel für das Projektmanagement und die Verwaltung verschiedener Versionen und Varianten ins Gebiet des Software-Engineerings.

Heutiger Stand der Programmiertechnik

Die heutige Situation stellt hohe Anforderungen an den Software-Entwickler. Die Produktion effizienter, qualitativ guter und wiederverwendbarer Software setzt Kenntnisse und Erfahrungen auf vielen Gebieten voraus, wie Programmiersprachen, Standardwerkzeuge (Editor, Compiler, Linker usw.), Programmierschnittstellen von Bibliotheken, parallele Prozesse und Synchronisation, Fehlerbehandlung, hardwarenahe Programmierung und Rechnerkommunikation. Dieses Wissen hat in der Vergangenheit bei vielen Projekten gefehlt. Kostenüberschreitungen, fehlende Dokumentation, mangelhafte Leistung und Probleme bei Erweiterungen und Wiederverwendung waren immer wieder festzustellen. Die Entwicklung von Echtzeitsoftware gilt deshalb auch heute noch als schwierig und risikoreich.

Wie immer, finden sich auch hier Ausnahmen, welche die Regel bestätigen. So stehen auf dem Gebiet der speicherprogrammierbaren Steuerungen heute graphische, PC-basierte Programmier- und Testwerkzeuge zur Verfügung, mit deren Hilfe der Anwender ein Programm ausschliesslich auf der Funktionsplanebene (Graphik) manipulieren kann. Im Vergleich zur konventionellen Programmiertechnik werden wesentlich weniger Spezialkenntnisse benötigt. Im folgenden werden einige heute weitverbreitete Programmierwerkzeuge und -methoden näher beschrieben.

Programmiersprachen

Während der letzten Jahre haben Hochsprachen mit Echtzeitunterstützung wie Pascal und Modula-2 sowie neuerdings auch Ada und C die Assemblersprachen in Echtzeitsystemen weitgehend verdrängt. Die Vorteile dieser Entwicklung sind Einfachheit, Maschinenunabhängigkeit, Portabilität, verbesserte Lesbarkeit, Unterstützung für strukturierte Programmierung und Typenchecks durch den Compiler. Neben diesen gemeinsamen Vorteilen hat jede dieser Sprachen auch ihre Eigenheiten:

- Die ursprüngliche Definition von *Pascal* [1] enthält keine Konstrukte für Multitasking, Synchronisation, hardwarenahe Programmierung und Modularisierung (Aufteilung eines Programmes in Module mit genau definierten Schnittstellen). Es entstanden aber eine ganze Reihe von Pascal-Dialekten, die alle oder einen Teil dieser Aufgaben unterstützen. Diese Erweiterungen sind jedoch nicht standardisiert und Programme sind nicht mehr portabel, d.h. sie können nicht mehr ohne Änderung auf ein anderes System gebracht werden.
- *Modula-2* [2] unterstützt die Modularisierung und die hardwarenahe Programmierung sehr gut. Weiter erlaubt das Modulkonzept, Erweiterungen wie Multitasking und Synchronisation in Modula-2 zu kodieren. Portabilität ist leicht erreichbar, indem die in Modula-2 kodierten Erweiterungen portiert werden.
- Die Sprache des amerikanischen Verteidigungsministeriums *Ada* [3] unterstützt Multitasking, Synchronisation, Modularisierung und hardwarenahe Programmierung. Sie wurde speziell im Hinblick auf die Portabilität von Anwendungen in der Mechatronik entwickelt. Im Vergleich mit Modula-2 ist Ada deutlich komplexer. Die Ada-Compiler setzen deshalb leistungsfähigere Rechner voraus und sind wesentlich teurer.
- Die Sprache *C* [4] verdankt ihre Beliebtheit und Verbreitung dem herstellerunabhängigen Betriebssystem Unix, das sich mehr und mehr zum Standard für Arbeitsstationen entwikkelt. Für die meisten neuen Prozessoren ist heute zuerst ein C-Compiler verfügbar. C ist gut geeignet für hardwarenahe Programmierung und unterstützt bis zu einem gewissen Masse die Modularisierung. Nachteile von C sind die schlechte Lesbarkeit und die mangelhafte Unterstützung von Datentypen. Besonders gravierend sind diese Nachteile unserer Meinung nach für Ingenieure, die nur gelegentlich programmieren, und für Systeme, die noch nach Jahren erweitert werden müssen, wenn der ursprüngliche Entwickler längst nicht mehr zur Verfügung steht.

Um das Bild der in der Schweiz verwendeten Echtzeitsprachen abzurunden, seien hier noch *Portal* [5] und *Chill* [6] erwähnt. Ähnlich wie Ada unterstützen beide Multitasking, Synchronisation, Modularisierung und hardwarenahe Programmierung. Chill

wird vor allem in der Telekommunikation eingesetzt.

Multitasking und Echtzeit

Echtzeit-Betriebssysteme sind heute Bestandteil der eingesetzten Programmiersprache (Pascal-Dialekte, Ada, Portal, Chill) oder stehen als Bibliothek von Unterprogrammen zur Verfügung (Modula-2, C). Sie unterstützen Multitasking und Synchronisation, um die Zusammenarbeit paralleler Prozesse zu koordinieren.

Die Gliederung eines Programmes in parallele Prozesse ist eine Aufgabe der Entwurfsphase, für die nur wenige allgemeingültige Regeln angegeben werden können:

- Das Echtzeitverhalten kann optimiert werden, indem zeitkritische Aufgaben von unkritischen getrennt werden. In einem System mit Verdrängung (Preemption) kann mit unterschiedlichen Prozessprioritäten erreicht werden, dass der Prozessor jederzeit der momentan wichtigsten Aufgabe zugewiesen wird: Die Verarbeitung eines wichtigen Ereignisses mit einer maximal tolerierbaren Reaktionszeit t_R muss eine Berechnung niedriger Priorität verdrängen, falls diese länger als t_R dauert.
- Unabhängige oder nur schwach miteinander gekoppelte Aufgaben werden mit Vorteil als parallele Prozesse bearbeitet, um auch eine Entkopplung der entsprechenden Programmteile zu erreichen.

Weiter basiert ein guter Entwurf vor allem auf Erfahrung. Neben der Randbedingung, dass der Overhead im Betriebssystem für Prozessumschaltungen und Synchronisation maximal 10 bis 20% der Rechenleistung betragen darf, muss auch die Korrektheit gewährleistet werden. Prozesse dürfen sich nicht gegenseitig blockieren (was vorkommen kann, wenn z.B. jeder wartet, bis ihm ein anderer eine Meldung sendet) und die Konsistenz von Daten darf nicht verletzt werden (Beispiel: zwei Prozesse modifizieren gleichzeitig dieselben Daten). Blosses Testen kann die Korrektheit von Echtzeitprogrammen nicht garantieren, da die Wahrscheinlichkeit des Auftretens eines vorhandenen Fehlers beliebig klein sein kann. Es sind deshalb auch formale Überlegungen nötig.

Die *Synchronisation*, das Gewährleisten zeitlicher Einschränkungen bei der Ausführung von Prozessen, gliedert man in gegenseitigen Ausschluss (Mutual Exclusion) und gegenseitiges Anstossen (Cross Stimulation). Zwischen den beiden Konzepten besteht ein fliessender Übergang. Grundsätzlich kann jede Art von Synchronisation sowohl mit gegenseitigem Ausschluss beim Zugriff auf globale Daten als auch durch Austauschen von Meldungen realisiert werden.

Gegenseitiges Anstossen – um z.B. von einem anderen Prozess eine Dienstleistung anzufordern – kann mit einer *Mailbox* (Briefkasten) realisiert werden. Mit der Operation *Send(mb, message)* wird eine Meldung in den Briefkasten gelegt, und mit *Receive(mb, message)* kann der andere Prozess die Meldung abholen. Falls noch keine Meldung da ist, wird er in *Receive* verzögert, bis eine eintrifft.

Gegenseitiger Ausschluss – z.B. um einen Zugriff auf globale Daten zu schützen – kann ebenfalls mit einer Mailbox erreicht werden. Zu Beginn wird im Briefkasten eine Meldung abgelegt, die als Schlüssel dient. Jeder Prozess muss vor einem Zugriff mit *Receive(mb, key)* den Schlüssel holen und ihn nachher mit *Send(mb, key)* wieder zurücklegen. Im Konfliktfall wird der Prozess, welcher den Schlüssel verlangt, verzögert, bis dieser verfügbar ist.

Software-Engineering

Das Lebensphasenmodell (Spezifikation, Entwurf, Realisierung, Test und Wartung) ist allgemein bekannt, wie auch die Tatsache, dass Kosten für die Korrektur eines Fehlers um Faktoren zunehmen, je später der Fehler entdeckt wird. Die praktische Anwendung der Methoden des Software-Engineering stösst aber noch häufig auf Schwierigkeiten. Es gibt beispielsweise kaum Unterstützung für Iterationen, d.h. für nachträgliche Änderungen der Spezifikation oder des Entwurfs. Die Versuchung ist gross, nur den Code zu korrigieren. Dann aber sind die Spezifikations- und Entwurfsdokumente nicht mehr mit dem Code konsistent. Eine weitere Schwierigkeit liegt in der Ausbildung und Erfahrung. Ein Anfänger lernt zuerst an kleinen Beispielen zu programmieren, und erst nach und nach auch zu entwerfen und zu spezifizieren. Innerhalb eines Projekts ist die Reihenfolge jedoch umgekehrt. Man ist deshalb von bereits gemachten Erfahrungen abhängig. Die Methoden, welche heute in der Spezifikations- und Entwurfsphase eingesetzt werden, sind Zustandsdiagramme, Entscheidungstabellen, Petri-Netze, Struktogramme, Datenflussdiagramme und Pseudocode. Die damit erarbeiteten Resultate werden manuell in den Code umgesetzt. Als Werkzeuge stehen Texteditoren, Compiler und Linker, symbolische Debugger sowie Versions- und Variantenverwaltungssysteme zur Verfügung.

Tendenzen in der Entwicklung und Forschung

Nach diesem Überblick über die heute üblichen Programmiertechniken werden nun als Beispiele einige kurz- und mittelfristige Entwicklungen diskutiert.

CASE-Werkzeuge (kurzfristig)

Rechnergestützte Werkzeuge für *Computer-Aided-Software-Engineering* stehen vielerorts kurz vor der Einführung oder werden bereits versuchsweise eingesetzt. Sie unterstützen Spezifikations- und Entwurfsmethoden wie SA/SD (Structured Analysis, Structured Design). Weitentwickelte Pakete enthalten Erweiterungen für Echtzeitanwendungen [7]. Die Voraussetzung für CASE-Werkzeuge sind graphikfähige Arbeitsstationen. Die Investitionen pro Arbeitsplatz betragen heute etwa 15 bis 40 000 Franken für Hard- und Software.

CASE-Werkzeuge erlauben dem Anwender, sich auf abstrakter Stufe mit seinem Problem auseinanderzusetzen und nicht auf der Detailstufe einer Programmiersprache. Die erzeugten Dokumente sind lösungsneutraler und leichter verständlich als konventionelle Programme. Weitere Erleichterungen für den Anwender werden sich ergeben, sobald die Werkzeuge automatisch Code erzeugen können und Iterationen, d.h. nachträgliche Korrekturen einer Spezifikation oder eines Entwurfs, wirksam unterstützen.

Exception-Behandlung (kurzfristig)

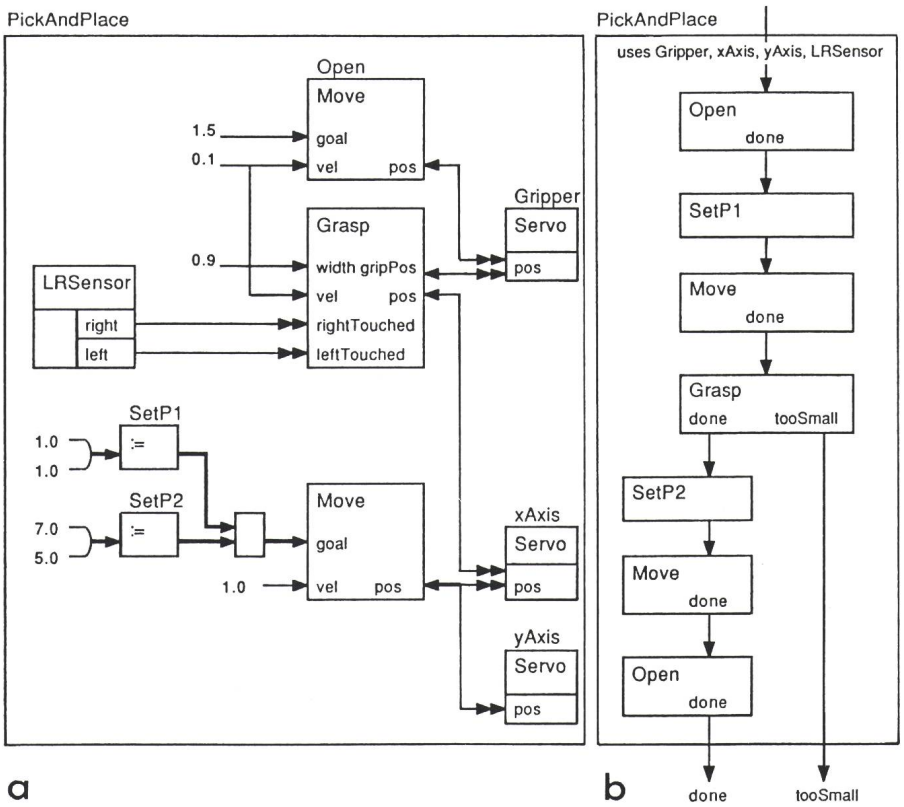
Eine *Exception* ist das Auftreten einer Bedingung, welche die normale Beendigung einer zugewiesenen Aufgabe verunmöglicht. Beispiele sind die Detektion von Fehlern durch die Hardware (illegale Adresse, Division durch Null) oder von Daten, die ihre Konsistenzbedingung nicht mehr erfüllen. In Systemen ohne Unterstützung zur Behandlung von Exceptions ist eine Exception gleichbedeutend mit einem Programmabsturz, den man mit *IF*- und *GOTO*-Anweisungen zu vermeiden sucht. Modernere Sprachen

(z.B. Ada) oder Betriebssysteme erlauben, Exceptions abzufangen und in speziellen Programmteilen, den *Exception-Handlern,* zu bearbeiten. Dort sucht man, den Fehler zu beheben oder zu vertuschen und in den normalen Programmfluss zurückzukehren (Recovery). Gelingt dies nicht, wird die Exception an die nächsthöhere Stufe weitergereicht. Normale Betriebszustände lassen sich so von der Fehlerbehandlung besser trennen. Es ergibt sich eine einfachere Programmstruktur - z.B. mit weniger *IF*-Anweisungen.

In Echtzeitsystemen muss die Exception-Behandlung mit der Synchronisation gekoppelt werden, um Deadlocks zu verhindern. Muss z.B. ein Prozess, der irgendein Betriebsmittel (z.B. einen Drucker) reserviert hat, infolge einer Exception abgebrochen werden, könnte sich nach und nach das ganze System blockieren, falls nicht garantiert wird, dass das Betriebsmittel wieder freigegeben wird. In [8] wird ein Exception-Behandlungsmechanismus für Echtzeitsysteme mit integrierter Objektverwaltung vorgestellt, der die Realisierung von fehlertoleranten Synchronisationskonzepten unterstützt. Ihre Semantik ist auch für den Fall von Exceptions klar definiert, und Zufälle können ausgeschlossen werden.

Für das Problem des gegenseitigen Ausschlusses wird ein abstrakter Datentyp *Region* mit den Operationen *Enter(region)* und *Exit(region)* vorgeschlagen. Verlässt ein Prozess einen mit *Enter/Exit* geschützten, kritischen Programmteil, so wird nicht nur garantiert, dass der Bereich wieder freigegeben wird, sondern auch, dass die Konsistenz der geschützten Daten überprüft werden kann, bevor der nächste Prozess zugreifen darf. Das System erlaubt, bestimmte Programmteile (z. B. Gruppen von Prozessen) vor Auswirkungen von Exceptions in anderen Programmteilen zu schützen.

Die Erfahrung hat gezeigt, dass ein wichtiger Vorteil darin liegt, dass nach einer Exception automatisch gewisse Objekte gelöscht werden. Damit ist beispielsweise jederzeit bekannt, welche der dynamisch gestarteten Prozesse noch laufen, und ein Programmteil, der infolge einer Exception abgebrochen wurde, kann ohne weiteres neu gestartet werden.

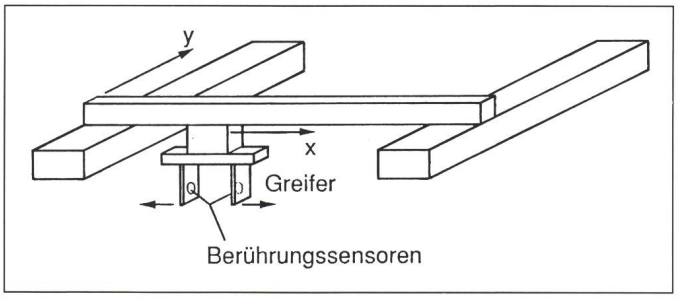


Figur 2 Programm PickAndPlace
a Datenfluss b Kontrollfluss

Vollgraphische Bedienungsschnittstellen (mittelfristig)

In grösseren Systemen wird man auch in der Mechatronik mehr und mehr vollgraphische Bedienungsschnittstellen, welche auf Fenster- und Menütechnik basieren, anbieten. Diese stellen dem Benützer das Modell der sogenannten *direkten Manipulation* zur Verfügung: Objekte werden graphisch dargestellt und können mit einer Maus direkt selektiert, verschoben, kopiert, gelöscht und eingefügt werden. In *Pull-Down-* oder *Pop-Up-Menüs* stehen jederzeit alle gerade erlaubten Befehle zur Verfügung, und in *Dialogfenstern* kann der Benützer des Systems zusätzlich benötigte Daten eingeben.

So einfach eine solche Bedienungsschnittstelle erlernt und eingesetzt wird, so kompliziert ist ihre Realisierung. Der Inhalt eines Fensters muss beim Verschieben, Verkleinern und Vergrössern überlappender Fenster sowie beim Scrollen[2] aufdatiert werden, und die Ereignisse, welche Maus und Tastatur erzeugen, müssen verarbeitet werden. Die Programmstruktur ist heute meist auf die Ereignisverarbeitung ausgerichtet. Die eigentliche Aufgabe eines Programms ist dann nicht mehr so leicht ersichtlich. Weiter



Figur 1 X-Y-Tisch mit Greifer

[2] Scrollen ist der Fachausdruck für das Verschieben des sichtbaren Ausschnitts eines Dokuments mit dem Ziel, einen anderen Ausschnitt darzustellen. Scrollen wird für grosse Dokumente gebraucht, die nicht mehr in ihre Fenster passen.

sind die verschiedenen Programmierschnittstellen, z.B. auf dem Macintosh und dem IBM PC (Graphikpaket GEM), so unterschiedlich, dass eine Übertragung eines Programms sehr aufwendig ist.

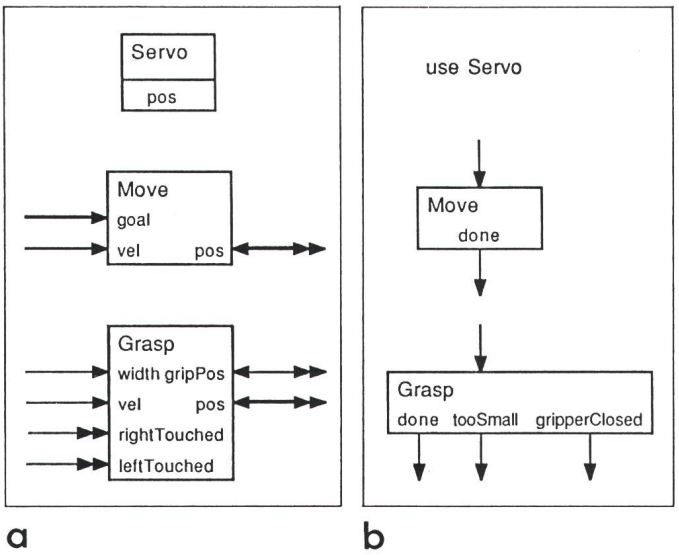
Die weitere Entwicklung läuft in Richtung sogenannter *User-Interface-Management-Systeme,* die eine Entkopplung der Anwendung und der graphischen Bedienungsschnittstelle zum Ziel haben. Längerfristig kann mit einer wesentlichen Vereinfachung des Entwicklungsaufwands für vollgraphische Bedienungsschnittstellen gerechnet werden. Ein erster Schritt in Richtung einer vereinfachten Programmierschnittstelle und der Portabilität zwischen verschiedenen Rechnern ist der optionale Teil des Modula-2 Operating System Standard Interface OSSI [9], der Graphik, Fenster- und Menütechnik unterstützt.

Anwendernahe graphische Programmierwerkzeuge (mittelfristig)

Beim Programmentwurf werden oft einzelne Aspekte in Diagrammen veranschaulicht. Darstellungen von Datenflüssen, Abläufen oder der statischen Programmstruktur entsprechen der Gewohnheit der Ingenieure, Strukturen und Zusammenhänge sichtbar zu machen. Es ist deshalb naheliegend, die Programmiertechnik in diese Richtung zu erweitern. Im Gebiet der speicherprogrammierbaren Steuerungen sind graphische Programmierwerkzeuge bereits üblich [10]. Ein Programm kann dort als Funktionsplan (Datenflussgraph) dargestellt werden, weil die Spezifikation im wesentlichen durch eine Abbildung der Eingänge auf die Ausgänge formuliert werden kann. Für grosse Systeme können diese Werkzeuge nicht eingesetzt werden, da die Darstellung des Datenflusses für die Ereignisverarbeitung und für die Bearbeitung komplexer Algorithmen nicht ausreicht.

Die Anwendung graphischer Programmiersprachen für allgemeine Echtzeitanwendungen mit automatischer Codegenerierung wird die Anforderungen an den Anwendungsprogrammierer reduzieren, die Dokumentation und Qualität verbessern sowie Änderungen und Erweiterungen auf tiefster Stufe durch den Anwendungsingenieur ermöglichen (z.B. Programmierung und Integration eines neuen Sensors in einen Roboter, inkl. Anpassung der Regelalgorithmen).

Figur 3
Schnittstellen der Gerätetypen *Servo* und der Funktionen *Move* und *Grasp*
a Datenfluss
b Kontrollfluss



Beispiel eines anwendernahen Programmierwerkzeuges

Am Forschungszentrum der Asea Brown Boveri AG läuft gegenwärtig ein Projekt, das die Anwendbarkeit graphischer Programmiertechniken wesentlich erweitern soll. Die zentrale Idee ist, sowohl den Datenfluss als auch den Kontrollfluss eines Programmteils in zwei unabhängigen, sich ergänzenden Sichten (Daten- und Kontrollfluss) darzustellen. Die Ziele bei der Entwicklung dieser Sprache waren des weitern klar definierte Schnittstellen und einfache Wiederverwendung von Programmteilen, Unabhängigkeit vom Zielsystem[3], Eignung für Debugging und interpretierte Ausführung.

Diese graphische Programmiersprache und ihr Einsatz werden an einem einfachen Beispiel illustriert: Es steht ein X-Y-Tisch mit einem Greifer zur Verfügung (Fig. 1). Seine Finger werden in X-Richtung geschlossen und sind mit Berührungssensoren bestückt. Das Programm *PickAndPlace* ergreift ein Objekt im Punkt P1 und legt es im Punkt P2 wieder ab. Jede der drei Achsen wird über den Sollwert ihres Lageregelkreises *(Servo)* gesteuert. Die Funktion *Move* erlaubt eine koordinierte Bewegung mehrerer Achsen zu einem beliebigen Ziel, die Funktion *Grasp* ein Greifen von Objekten.

Das Hauptprogramm in Figur 2 zeigt im Kontrollfluss die Sequenz der einzelnen Operationen und im Datenfluss die Abhängigkeiten der beteiligten Geräte und Funktionen. Die drei Achsen *xAxis, yAxis* und *Gripper* werden durch Instanzen (Tab. I) des Gerä-

[3] So soll z.B. die Aufteilung auf parallele Prozesse und ihre Verteilung auf die Knoten eines Rechnernetzwerkes gegen aussen versteckt werden.

Begriffsdefinitionen

Eine *Funktion* ist ein seiteneffektfreier Algorithmus, der aus den Werten der Funktionseingänge Werte für die Ausgänge der Funktion berechnet. Eine Funktion kann mehrfach (auch gleichzeitig) mit verschiedenen Eingangswerten aufgerufen werden. Ein Aufruf wird als Funktionsinstanz bezeichnet.

Ein *Gerätetyp* (Device Type) ist ein aktiver Datentyp. Seine Schnittstelle besteht aus Datenfeldern. Wie Funktionen, sind Gerätetypen mehrfach verwendbar. Eine Instanz eines Gerätetyps wird *Gerät* (Device) genannt. Im Gegensatz zu den normalen (passiven) Datentypen werden die Werte der Datenfelder eines Geräts intern verarbeitet oder modifiziert. Die beiden Begriffe entsprechen somit der Vorstellung mittels Software realisierter Peripheriegeräte.

Das Verhalten einer Funktion (oder eines Gerätetyps) wird durch ihr *Interface* (Schnittstelle) definiert. Die *Implementation* (Realisierung) ist von der Schnittstelle getrennt und ist für den Anwender der Funktion (oder des Gerätetyps) nicht sichtbar.

Tabelle I

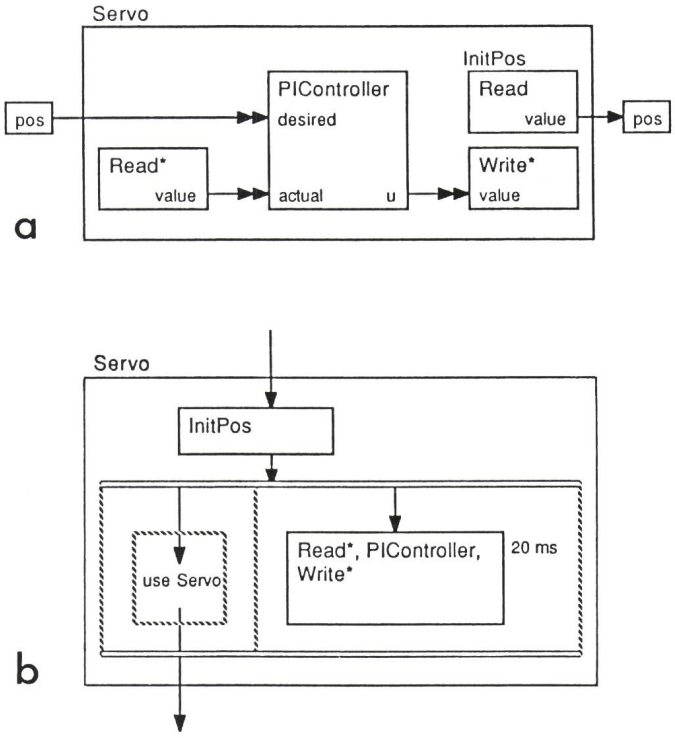
tetyps *Servo*, welcher das Datenfeld *pos* (Positionssollwert) enthält, modelliert (Fig. 3). Die Funktion *Move* wird sowohl zum Öffnen des Greifers als auch zur Positionierung des X-Y-Tisches eingesetzt. Die Instanz des Gerätetyps *LRSensor* modelliert die zwei Berührungssensoren im Greifer. Der Gerätetyp *Servo* wird als abgetasteter PI-Regelkreis realisiert. Er setzt sich aus dem Lesen der Soll- und Istposition (Positionssensor), dem Regelalgorithmus und der Ausgabe des Steuerwerts (Fig. 4) zusammen. Die Sollposition *pos* darf nur in kleinen Schritten verändert werden, damit die effektive Position nur wenig davon abweicht. Beim Starten des Servos wird *pos* mit der aktuellen Position initialisiert. Der Regelkreis arbeitet parallel zum restlichen Programm *(use Servo)*.

Die Figur 5 zeigt, wie in der Funktion *Move* aus dem aktuellen und dem gewünschten Positionsvektor die Distanz und Bewegungsrichtung bestimmt *(Plan)* und an die Funktion *GenSetPoint* übergeben werden, die dann den Ausgangsvektor *pos* schrittweise verändert, bis die Zielposition *goal* erreicht ist. Die Funktion *Grasp* (Fig. 6) hat die Aufgabe, einen Greifer zu schliessen, bis das Objekt mit beiden Fingern berührt wird. Falls einer der Sensoren zuerst anspricht, muss mit einer Kompensationsbewegung der Greiferaufhängung verhindert werden, dass das Objekt umgestossen werden könnte. Für den allgemeineren Fall eines drehbaren Greifers muss als Vorbereitung dessen Orientierung bestimmt werden.

Die graphische Programmiersprache basiert auf seiteneffektfreien[4] Funktionen und Gerätetypen. Die Schnittstelle einer Funktion umfasst im Datenfluss die Ein- und Ausgänge, im Kontrollfluss die verschiedenen Ereignisse, die eine Ausführung beenden können. *Grasp* z.B. kann erfolgreich mit *done* oder mit den Fehlerbedingungen *tooSmall* oder *gripperClosed* enden.

Bei Ein- und Ausgängen einer Funktion wird unterschieden, ob pro Ausführung ein einzelnes Datenobjekt (einfacher Pfeil) oder ein kontinuierlicher Fluss von Datenobjekten (Doppelpfeil) konsumiert oder produziert wird. Eine einzelne Ausführung von

Figur 4
Implementation des Gerätetyps *Servo*
a Datenfluss
b Kontrollfluss

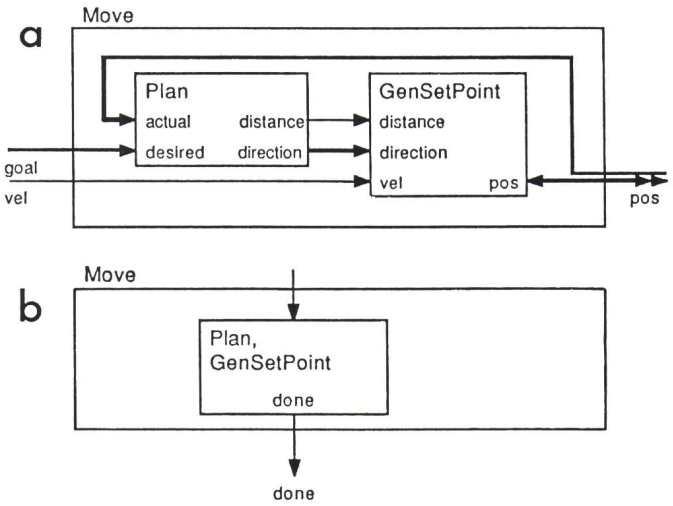


Move erzeugt aus einer Zielposition eine Sequenz von neuen Positionssollwerten. Eine Funktion, die nur einzelne Datenobjekte bearbeitet, kann mit dem Operator * repetitiv aufgerufen werden, um Datenflüsse zu verarbeiten (z.B. *Read** in Fig. 2).

Skalare Daten werden in der Datenflusssicht mit dünnen, strukturierte Daten (z.B. Vektoren) mit dicken Linien dargestellt. Direkte Signale zwischen Funktionen im Datenfluss bestimmen auch die Ausführungsreihenfolge. Im Kontrollfluss erscheint eine Menge so gekoppelter Funktionen zusammengefasst als einzelne Operation (z.B. *Read*, PIController, Write** in Fig. 4). Falls die Ausführungsreihenfolge nicht aus dem Datenfluss abgeleitet werden kann, müssen Variablen eingefügt werden (z.B. *rightDelta* in Fig. 6). Eine detaillierte Beschreibung des Beispiels und der graphischen Programmiersprache ist in [11] zu finden.

Praktisch kann eine derartige Programmiersprache erst eingesetzt werden, wenn ein syntaxgesteuertes, graphisches Werkzeug mit automatischer Codegenerierung verfügbar ist. Die Konsistenz zwischen Datenfluss und Kontrollfluss muss vom Werkzeug dauernd garantiert werden, damit der Programmierer die Sprachregeln nicht verletzen kann und sie deshalb auch nicht in allen Details kennen muss. Ein solches Werkzeug befindet sich

Figur 5
Implementation der Funktion *Move*
a Datenfluss
b Kontrollfluss



[4] d.h. Funktionen, die nur die eigenen Ausgänge beeinflussen.

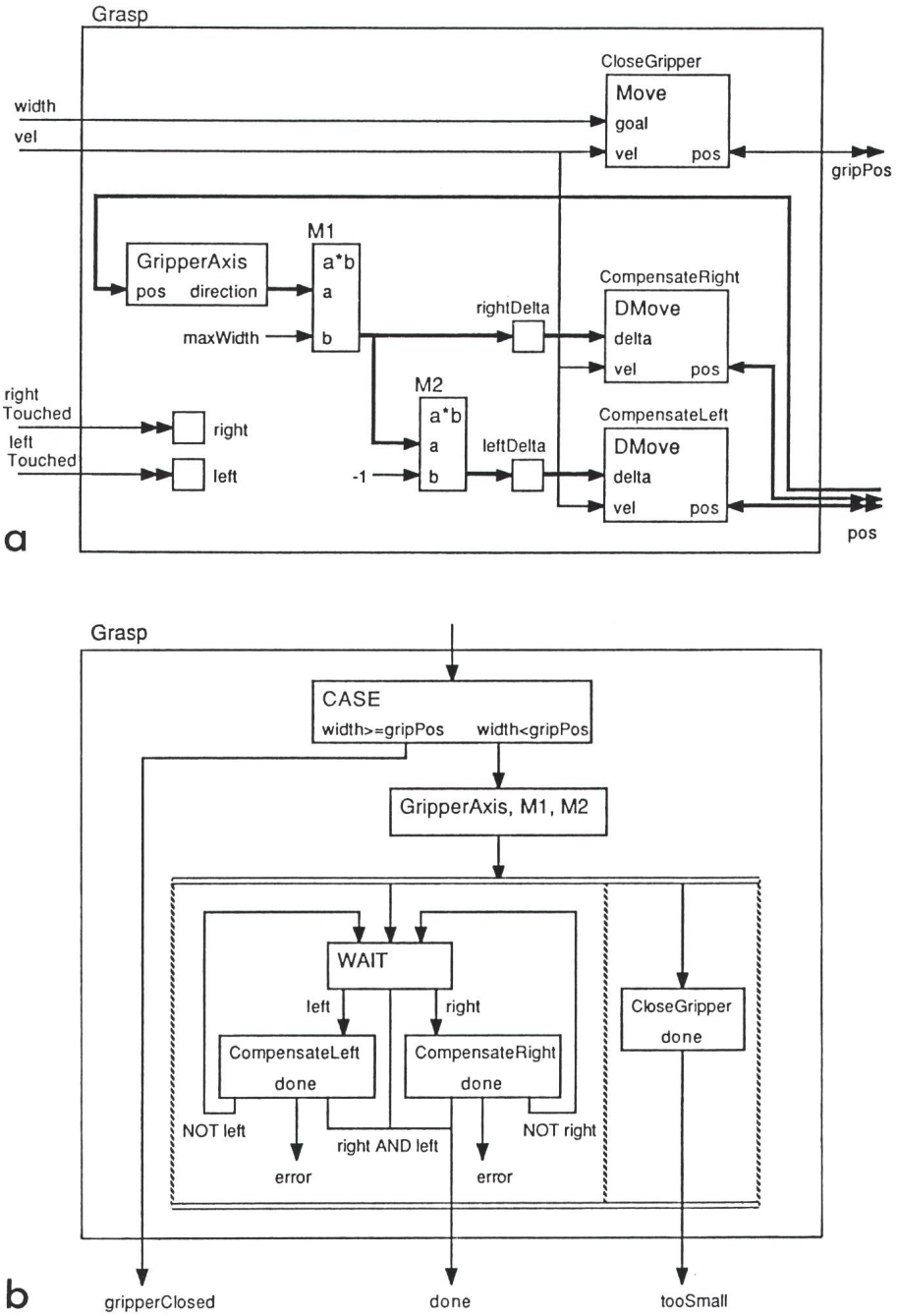
gegenwärtig im ABB-Forschungszentrum in Entwicklung. Der Prototyp wird automatisch Modula-2-Quellencode generieren, um den Aufwand zur Einbettung in eine konkrete Umgebung minimal zu halten.

Längerfristig muss das Programmierwerkzeug in eine Umgebung eingebettet werden, die das Testen von Programmen erlaubt und Zugriff auf eine Engineeringdatenbank mit der Anlagekonfiguration besitzt, um das Anbinden eines Programms an I/O-Geräte (Sensoren und Stellglieder) und die Verteilung des Programms auf mehrere Rechner zu unterstützen.

Im Vergleich zu allgemeinen CASE-Werkzeugen stellt sich die Frage, ob bei der Automatisierung technischer Systeme weiterhin das Modell der phasenweisen Software-Erstellung im Vordergrund stehen wird oder eher ein Modell, das mehrere orthogonale (d.h. einander nicht beeinflussende) Tätigkeiten vorsieht, wie das Erfassen der Anlagekonfiguration, das Programmieren der Funktion und das Abbilden der Funktion auf die Konfiguration.

Schlussbetrachtungen

Die heutigen Probleme bei der Erstellung von qualitativ hochstehender Echtzeitsoftware haben ihre Ursache primär in den vielfältigen Anforderungen, die an den Entwickler gestellt werden und damit in dessen begrenzter Ausbildung und Erfahrung. Wesentliche Verbesserungen sind nur zu erwarten, wenn es gelingt, die Summe aller Anforderungen zu senken. In der Vergangenheit wie in der Gegenwart wurde und wird versucht, wiederkehrende Aufgaben nicht immer neu zu lösen, sondern Standardsoftware zu entwickeln und einzusetzen. So wurden projektspezifische Lösungen zur Befriedigung von Parallelitätsanforderungen durch Standard-Echtzeitbetriebssysteme abgelöst, und anstelle von individuellen Mechanismen zur Fehlerbehandlung werden mehr und mehr allgemein verwendbare Exception-Behandlungsmechanismen eingesetzt. In Zukunft werden - das lässt sich bereits heute mit Sicherheit sagen - Programmiersprachen und Werkzeuge verfügbar sein, welche die heute übliche manuelle Aufteilung eines Programms in parallele Prozesse und deren Verteilung auf mehrere Rechner zumindest teilweise automatisieren werden.



Figur 6 Implementation der Funktion *Grasp*
a Datenfluss b Kontrollfluss

Literatur

[1] *K. Jensen* and *N. Wirth:* Pascal - User manual and report. Lecture notes in computer science, vol. 18. Berlin/Heidelberg, Springer-Verlag, 1974.

[2] *N. Wirth:* Programmierung in Modula-2. Texts and monographs in computer science. Berlin/Heidelberg, Springer-Verlag, 1982.

[3] Reference manual for the Ada programming language. ANSI/MIL-Standard 1815A-1983.

[4] *B.W. Kernighan* and *D.M. Ritchie:* The C programming language. Englewood-Cliffs, N.J., Prentice-Hall,1978.

[5] *H.H. Nägeli:* Programmieren mit Portal. Eine Einführung. Zug, Landis & Gyr, 1981.

[6] *W. Sammer* und *H. Schwärtzel:* Chill. Eine moderne Programmiersprache für die Systemtechnik. Berlin/Heidelberg, Springer-Verlag, 1982.

[7] *D.J. Hatley* and *I.A. Pirbhai:* Strategies for real-time system specification. New York, Dorset House Publishing Corporation, 1987.

[8] *G.E. Maier:* Exception-Behandlung und Synchronisation. Entwurf und Methode. Informatik-Fachberichte 105. Band. Berlin/Heidelberg, Springer-Verlag, 1985.

[9] *E. Biagioni* a.o.: OSSI - A portable operating system interface and utility library for Modula-2. ETH-Institut für Informatik, Report 79. Zürich, ETH, 1987.

[10] *R. Güth* und *J. Kriz:* Informatikforschung in einem Industrieunternehmen. Technische Rundschau (1986) 27, S. 58-63.

[11] *G.E. Maier:* Towards graphical programming in control of mechanical systems. Proceedings of the IUTAM/IFAC Symposium on Dynamics of Controlled Mechanical Systems. Berlin/Heidelberg, Springer-Verlag.