Xpert.press

Peter Scholz

Softwareentwicklung eingebetteter Systeme

Springer

Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Peter Scholz

Softwareentwicklung eingebetteter Systeme

Grundlagen, Modellierung, Qualitätssicherung

Mit 30 Abbildungen

Springer

Peter Scholz
Fachhochschule Landshut
Fachbereich Informatik
Am Lurzenhof 1
84036 Landshut
peter.scholz@fh-landshut.de

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

ISSN 1439-5428
ISBN 3-540-23405-5 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142/YL - 5 4 3 2 1 0

Vorwort

„Man darf das Schiff nicht an einen einzigen Anker
und das Leben nicht an eine einzige Hoffnung binden!“

(Epiket, griechischer Philosoph, ca. 50 bis 138)

Eingebettete Systeme (engl. embedded systems) sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind. Solche Umgebungen können maschinelle Systeme wie etwa Kraftfahrzeuge, Flugzeuge, Fernsehgeräte, Waschmaschinen, Küchengeräte, Mobilfunktelefone u. a. sein, die der Interaktion eines menschlichen Benutzers bedürfen (z. B. Steer-by-Wire im Kraftfahrzeug) oder vollautomatisch agieren (z. B. Antiblockiersystem). Teilweise ist dieser Übergang auch fließend, so beispielsweise bei einem elektronischen Bremsassistenten.

Schon heute hat z. B. ein Mittelklassewagen Elektrik und Elektronik im Wert von rund 2.200 Euro an Bord. In zehn Jahren wird sich dieser Wert voraussichtlich auf ca. 4.200 Euro fast verdoppelt haben. Eingesetzt wird Informationstechnologie vor allem in den Bereichen Fahrwerksantrieb, Motormanagement, Sicherheit, Komfort, Emissionsreduzierung und Kommunikation/Entertainment bzw. Infotainment.

Eingebettete Systeme übernehmen komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben für (bzw. in) diese(n) technischen Systeme(n). Ihre Funktionalität wird durch das Zusammenspiel von Spezialhardware, Standardprozessoren, Peripherie und Software realisiert.

Sie unterscheiden sich daher von anderen Systemen, wie beispielsweise Textverarbeitungs-, Buchhaltungs-, Internet- oder Warenwirtschafssystemen grundlegend und in mannigfaltiger Weise: Diese Systeme sind ausschließlich in Software realisiert, arbeiten

ggf. durch Interaktion mit einem menschlichen Benutzer vor einer Tastatur bestimmte Aufgaben ab, d. h. sie transformieren in einer ersten Näherung Eingaben in Ausgaben: Benutzereingaben werden in einem iterativen Prozess gelesen, bearbeitet und schließlich in Ausgaben transformiert. Bei ihrer Beschreibung liegt der Schwerpunkt auf den Transformationsprozessen bzw. -algorithmen selbst. Man spricht hier auch von „(rein) transformationellen" oder „interaktiven" Systemen.

Dem gegenüber stehen die reaktiven Systeme, die als Verallgemeinerung von eingebetteten Systemen fortwährend auf Eingaben ihrer technischen Umgebung bzw. eines technischen Prozesses reagieren und entsprechende, Kontext-abhängige Ausgaben erzeugen. Eingebettete, reaktive Systeme sind in eine – möglicherweise recht feindliche (Schmutz, Hitze, Kälte, schnelle Bewegung etc.) – Umgebung eingebettet. Bei ihnen liegt der Schwerpunkt der Beschreibung daher auf der Beschreibung der Interaktion zwischen System und Umgebung und damit auf dem Ein-/Ausgabeverhalten des Systems. Reaktive Systeme müssen zu jedem Zeitpunkt in einer nicht vom Computersystem selbst getriebenen Weise auf Sensordaten der Umgebung reagieren können und unterliegen hierbei oft sogenannten Echtzeitanforderungen.

Bei Echtzeitanforderungen unterscheidet man zwischen harten und weichen. „Harte" Echtzeitanforderungen müssen zur Erfüllung der Funktion eingehalten werden (Beispiel: Ein Airbag muss innerhalb von wenigen hundertstel Sekunden voll aufgeblasen werden, sonst verfehlt er seine Wirkung). „Weiche" Echtzeitanforderungen dagegen erhöhen den Komfort, man spricht hier gerade im Automobilbau auch von „Komfortelektronik" (Beispiele sind elektrische Fensterheber, Zentralverriegelung usw.). Wird das Thema „Embedded Systems" aus dem Blickwinkel der Ingenieursdisziplinen betrachtet, so wird hier naturgemäß gerne der Schwerpunkt auf Hardware-Gesichtspunkte (Mikrocontroller, Signalprozessoren, Sensoren, Aktoren, Analog-Digital-Wandler usw.) gelegt. Daraus könnte leicht der falsche Eindruck erwachsen, bei der Entwicklung eingebetteter Systeme handele es sich um eine reine Hardwareentwicklungsaufgabe. Dies ist aber mitnichten so. Vielmehr handelt es sich beim Entwurf eingebetteter Systeme um eine mindestens genauso wichtige Softwareentwurfsaufgabe. Gerade letztere ist Kern regen wissenschaftlichen Interesses (Rosenstiel, 2003), was es in überschaubarer Zukunft wohl auch noch bleiben wird. Ein aktueller Wegweiser für die Forschung und Lehre für das Software Engineering im Bereich eingebetteter Systeme findet sich in (Broy und Pree, 2003).

In diesem Buch wollen wir einen ersten Überblick über das Thema geben. Wir starten dabei nach einer ausführlichen Einleitung und Hinführung zum Thema mit der Diskussion von nebenläufigen Systemen. Danach widmen wir uns den Gebieten Echtzeit, Echtzeitsysteme und Echtzeitbetriebssysteme. Im Anschluss werden wir dann einen Überblick zur Entwicklung eingebetteter Systeme geben. An eingebettete Systeme werden oft Echtzeitanforderungen gestellt. Ein Echtzeitsystem ist ein System, bei dem der Zeitpunkt, zu dem Ausgaben erzeugt werden, bedeutend ist. Programme, die auf einer (fast) beliebigen Hardware ablaufen, die Grundfunktionen von Betriebssystemen erfüllen und Echtzeitverhalten aufweisen, nennt man Echtzeitbetriebssysteme. Wir beginnen daher mit einer Beschreibung von Echtzeitbetriebssystemen und widmen uns dann in den folgenden Kapiteln der Programmierung von eingebetteten Systemen, bevor wir auf ausgewählte Techniken zum Softwareentwurf für diese Systeme eingehen. Da es sich bei eingebetteten Systemen oft um sicherheitskritische Systeme handelt, deren Fehlfunktion ihre Umgebung massiv beeinträchtigen kann und letztendlich sogar zur Gefährdung von Menschenleben führen kann, ist die Qualität solcher Systeme von zentraler Bedeutung. Dieses Buch enthält daher ebenfalls eine überblicksartige Betrachtung des Themas Softwarequalität und schließt mit einer Zusammenfassung verschiedener für Embedded Systeme geeigneter Vorgehensmodelle.

Ein kompaktes Buch wie das vorliegende kann naturgemäß ein derart komplexes und umfangreiches Thema wie die Softwareentwicklung eingebetteter Systeme nicht auch nur annähernd erschöpfend in allen Details behandeln. Dieser Anspruch wird demnach selbstverständlich gar nicht erst erhoben. Vielmehr soll es einen Zugang zu diesem – gerade für die deutsche Softwareindustrie in den kommenden Jahren wohl sehr zentralen – Thema schaffen und „Lust auf mehr“ generieren. Ein besonderes aber nicht ausschließliches Augenmerk gilt dabei der automobilen Softwareentwicklung.

Bei der Auswahl der Inhalte habe ich mich dabei in erster Linie davon leiten lassen, wo ich vor allem aufgrund meines persönlichen Hintergrunds aus Lehre, anwendungsnaher Forschung und Praxiserfahrung aus Industrietätigkeiten Handlungsbedarf in den softwareerstellenden Unternehmen sehe. Viele dieser Inhalte konnte ich mit zahlreichen Teilnehmern aus meiner Weiterbildungsreihe „IT Update“ für Fach- und Führungskräfte persönlich und ausführlich besprechen. Insbesondere fanden dabei von mir angebotene Tagesseminare zu Themenkomplexen wie „Software Engineering“,

„Softwarequalität“ und „Software Engineering eingebetteter Systeme“ großen Zuspruch. Inhaltlich geht das Buch an jenen Stellen in die Tiefe, wo ich vor allem in den industriellen Forschungs- und Entwicklungsbereichen Entwicklungspotential sehe. Mit meinem Buch möchte ich daher den Praktiker genauso ansprechen wie Studenten der Informatik, Elektrotechnik oder Mechatronik im Hauptstudium, die erstmals Zugang zu diesem Thema suchen.

Eingebettete Systeme sind eine der schnellstwachsenden Branchen unter den Informatikanwendungen. In Zukunft darf sogar eher noch mit einer Zunahme dieses Ungleichgewichts gerechnet werden. Das Buch zeigt auch deutlich auf, wo in Zukunft interessante Aufgabengebiete und berufliche Chancen für Informatiker und Informationstechniker liegen werden. Die Darstellungsweise der Inhalte orientiert sich dabei gezielt an der Sprachwelt der Informatik.

Dieses Buch wurde ganz bewusst in Deutsch verfasst, ist aber stets bemüht, englische Fachbegriffe weitestgehend ebenfalls einzuführen. Da viele der tangierten Themenbereiche überwiegend auf einer englischsprachigen Terminologie basieren, wird gar nicht erst der Versuch unternommen, Anglizismen zu vermeiden.

Lesehinweis: Nach der Lektüre der Kapitel 1 bis 3 können die nachfolgenden Kapitel in beliebiger Reihenfolge gelesen werden.

Die Erstellung dieses Werkes wäre niemals ohne die tatkräftige Unterstützung Anderer möglich gewesen. Zunächst möchte ich meinen Studenten des Studienprojekts „Embedded Systems“ aus dem Jahre 2004 (den Herren Philipp Konradi, Matthias Ecker, Christian Könik, André Hofmann und Florian Kandlinger) am Fachbereich für Informatik der Fachhochschule Landshut danken, die mit ihrer Literaturrecherche wichtige flankierende Arbeiten geleistet haben. Mit den Ergebnissen aus ihrem, von mir betreuten Studienprojekt konnten sie den ersten Preis bei den „Audi IT Tagen“ im Herbst 2004 gewinnen, was mich letztendlich zum Verfassen dieses Buches beflügelt hat. Besonders bedanken möchte ich mich auch bei Frau Stephanie Hahn für die Übernahme des Erstlektorates. Weiterhin gilt mein spezieller Dank Frau Jutta Maria Fleschutz, Frau Gabi Fischer und Frau Dorothea Glaunsinger vom Springer-Verlag, die bei redaktionellen Fragen stets mit Rat und Tat zur Seite standen.

Ich widme dieses Buch in Dankbarkeit meinen Eltern.

Peter Scholz Februar 2005

Inhaltsverzeichnis

1 Einleitung

Kapitelübersicht

Das erste Kapitel enthält eine allgemeine Hinführung zum Thema. Diese beginnt mit einer kurzen Motivation und geht dann nach einer Klassifikation eingebetteter Systeme auf relevante Beispiele und betroffene Branchen sowie Anwendungen ein. Es folgt ein Überblick über die Struktur eingebetteter Systeme. Nach einer Reihe begrifflicher Definitionen schließt das Kapitel mit einer Diskussion zentraler Herausforderungen auf diesem Gebiet. Nach der Lektüre dieses Kapitels sollte der Leser die zentrale Rolle eingebetteter Systeme für die Informatik verstanden haben, sie von anderen Systemen unterscheiden können und die wichtigsten Anwendungen und Begriffe kennen, sowie die noch zu lösenden Herausforderungen auf diesem Gebiet einschätzen können.

1.1 Motivation

Eingebettete Systeme

Unter einem *eingebetteten System* verstehen wir ein in ein umgebendes technisches System eingebettetes und mit diesem in Wechselwirkung stehendes Computersystem. Es übernimmt dort komplexe Steuerungs-, Regelungs-, Überwachungs- und Datenverarbeitungsaufgaben und verleiht damit dem umgebenden System oft einen entscheidenden Wettbewerbsvorsprung. Solche Systeme sind heute auf breiter Front in alle Bereiche der Technik eingedrungen. Mehr als *neun von zehn* aller elektronischen Bauelemente sind in eingebetteten Systeme implementiert. In der Tat sind in modernen Personenkraftwagen der Oberklasse zwischen 70 und 80 integrierte und miteinander vernetzte Steuergeräte enthalten (Broy et al. 1998).

Reaktive Systeme

Reaktive Systeme nehmen heute in modernen Computersystemen eine bedeutende Rolle ein. Im Gegensatz zu rein transformationellen Systemen, also solchen, die lediglich Eingaben, die beim Systemstart vollständig zur Verfügung stehen, in Ausgaben, die erst bei der

Systemterminierung vollständig berechnet sind, verarbeiten, interagieren reaktive Systeme beständig mit ihrer Umgebung (Halbwachs, 1993), (Harel, Pnueli, 1985). Dabei wird die Interaktion des Systems mit seiner Umgebung weniger durch das System selbst getrieben, sondern von Ereignissen aus der Umgebung.

Reaktive Systeme finden ihre Anwendung in der Flugzeug-, Automobil- oder Telekommunikationselektronik. Bemerkenswerterweise sind bereits mehr Mikroprozessoren in reaktiven Systemen als in Personalcomputern eingebaut. So hat eine statistische Erhebung im Jahre 2002 ergeben, dass von 8,3 Milliarden weltweit produzierten Prozessoren 8,15 Milliarden in eingebetteten Systemen verbaut wurden, aber lediglich 150 Millionen in transformationellen Computersystemen (wie Personal Computer, Server, Mainframes usw.). Dies entspricht einem Verhältnis von 98:2 zugunsten der eingebetteten Systeme. In (Fränzle, 2002) ist dieser Zusammenhang aus Sicht eines eingebetteten Systems humoristisch und doch sehr treffend formuliert: „Ach wie gut, dass niemand daran denkt, dass mich ein Computer lenkt".

Eingebettete Systeme sind eine der schnellstwachsenden Branchen unter den Informatikanwendungen. In Zukunft darf sogar eher noch mit einer Zunahme dieses Ungleichgewichts gerechnet werden. Es zeigt auch deutlich auf, wo in Zukunft interessante Aufgabengebiete und berufliche Chancen für Informatiker und Informationstechniker liegen werden.

Die Hauptgründe für das große Interesse an eingebetteten Systemen sind Fortschritte in Schlüsseltechnologien wie Mikroelektronik und formalen Methoden zu ihrer exakten Beschreibung als Grundlage für Sicherheit und Zuverlässigkeit sowie die sich daraus ergebende Vielfalt von Anwendungen.

Aktuelle Trends

Aktuelle Trends beim Entwurf eingebetteter Systeme sind:

- Steigender Anteil des elektronischen Teilsystems, dabei steigender Anteil des digitalen Teilsystems sowie *steigender SW-Anteil,*
- Trend zu immer mehr Intelligenz und fortschreitender *Vernetzung.*
- *Entwurfskompromiss:* kostengünstige Standardkomponenten vs. schnelle Spezialhardware

1.2 Klassifikation, Charakteristika

Die heute verfügbaren Computersysteme können in drei unterschiedliche Klassen eingeteilt werden (Halbwachs, 1993), (Harel, Pnueli, 1985): (rein) transformationelle, interaktive und reaktive Systeme. Sie werden in erster Linie durch die Art und Weise unterschieden, wie sie Eingaben in Ausgaben transformieren (Scholz, 1998).

Transformationelle Systeme

Transformationelle Systeme transformieren nur solche Eingaben in Ausgaben, die zum Beginn der Systemverarbeitung vollständig vorliegen. Die Ausgaben sind dann nicht verfügbar, bevor die Verarbeitung terminiert. In solchen Systemen ist der Benutzer, oder allgemeiner, die Systemumgebung, nicht in der Lage, während der Verarbeitung mit dem System selbst zu interagieren und so Einfluss auf ihr Ergebnis zu nehmen.

Interaktive Systeme

Interaktive Systeme dagegen, also beispielsweise Betriebssysteme, erzeugen Ausgaben nicht nur erst dann, wenn sie terminieren, sondern sie interagieren und synchronisieren stetig mit ihrer Umgebung. Diese Interaktion wird durch das System selbst und nicht etwa durch seine Umgebung bestimmt: Wann immer das System neue Eingaben zur Fortführung weiterer Verarbeitungsschritte benötigt, wird die Umgebung bzw. der Benutzer hierzu aufgefordert (engl. to prompt) – das System synchronisiert sich auf diese Weise *proaktiv* mit seiner Umgebung. Wird diese Synchronisierung dagegen durch die Systemumgebung anstelle des Systems bestimmt, so nimmt das System selbst eine reaktive Rolle ein und wir sprechen von einem reaktiven System.

Ein bedeutender Unterschied zwischen den beiden Arten von Systemen ist der „Herr“ der Interaktion. Bei interaktiven Systemen ist es der Computer. Die „Anfragenden“ warten bis sie bedient werden, der Computer entscheidet wer, wann und wie behandelt wird. Nennenswerte Herausforderungen bei interaktiven Systemen sind die Vermeidung von Verklemmungen (engl. deadlocks), „Fairness“ und die Konsistenz verteilter Informationen. Bei reaktiven Systemen ist es dagegen die Umgebung, die vorschreibt was zu tun ist. Der Computer hat in der vorgegebenen Zeit auf Stimuli der Umgebung zu reagieren. Größte Belange sind die Sicherheit und die Rechtzeitigkeit (Berry, 1998).

Der Unterschied zwischen reaktiven und interaktiven Systemen hat großen Einfluss auf den Verhaltensdeterminismus in diesen Systemen. Interaktive Systeme werden größtenteils als nichtdeterministisch angesehen, denn der Computer trifft intern die Entscheidung ob und wann eine Anfrage beantwortet werden soll. Auch die Reak-

tion auf eine Sequenz der Anfragen muss nicht immer gleich sein. Im Gegensatz dazu ist das deterministische Verhalten ein fester Bestandteil eines reaktiven Systems. Die Reaktion muss eindeutig durch die Eingabesignale und evtl. deren zeitliche Reihenfolge definiert sein. Beispiel dafür ist die Steuerung eines Flugzeugs oder Autos.

Das Verhalten eines nichtdeterministischen Systems ist weitaus komplexer zu modellieren als das eines deterministischen. Die Charakteristik der beiden Systeme muss unbedingt bei der Entwicklung von geeigneten Techniken, Methoden und Werkzeugen berücksichtigt werden.

Reaktive Systeme

Die wichtigsten Eigenschaften *reaktiver Systeme* kann man wie folgt charakterisieren (Halbwachs, 1993): Sie arbeiten oftmals nebenläufig, müssen äußerst zuverlässig sein und dabei Zeitschranken einhalten. Sie können sowohl in Hardware wie auch in Software realisiert und auf einer komplexen, verteilten Systemplattform implementiert werden.

Die funktionale Korrektheit reaktiver Systeme spielt eine entscheidende Rolle bei der Entwicklung solcher Systeme. Nicht nur, dass sich die Markteinführung eines Produktes verzögern kann, wenn Systemfehler erst in einer späten Entwicklungsphase entdeckt werden, schlimmer noch sind kostspielige Rückrufaktionen, wenn Fehler erst dann bekannt werden, wenn das Produkt schon beim Endabnehmer angelangt ist. Schließlich können unentdeckte Fehler bei reaktiven Systemen in kritischen Anwendungen auch noch zu Konsequenzen von größerer Tragweite führen.

In aller Regel sind reaktive Systeme in eine komplexe, beispielsweise mechanische, chemische oder biologische Systemumgebung eingebettet. Es handelt sich dann um *eingebettete Systeme*. Mit Umgebung bezeichnen wir nicht nur die natürliche Umgebung in der das kontrollierende, reaktive System agiert, sondern auch den kontrollierten Teil des komplexen Gesamtsystems. Für solche reaktive, eingebettete Systeme ist in aller Regel nicht nur deren funktionale Korrektheit wichtig, sondern auch ihre Reaktionszeiten. Man spricht in diesem Zusammenhang von *Echtzeitsystemen*, vgl. Abbildung 1.1 (siehe auch Abschnitt 4).

Eingebettete Systeme lassen sich nach einer Reihe weiterer, unterschiedlicher Kriterien klassifizieren. Zum einen bietet sich eine Einordnung in die im folgenden Abschnitt 1.3 genannten Produktkategorien an. Zum anderen treten diese Systeme in unterschiedlichen technischen Ausprägungen auf. So lassen sich *kontinuierliche* von *diskreten* und *verteilte* von *monolithischen* Systemen unterscheiden. Enthält ein System sowohl kontinuierliches als auch

diskretes Verhalten, so spricht man von einem *hybriden* System (siehe Abschnitte 1.4 und 5.8).

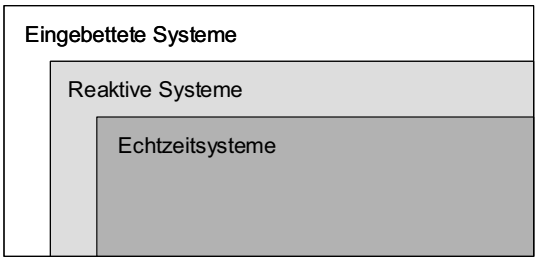


Abb. 1.1 Klassifikation eingebetteter Systeme

Sicherheitskritische Systeme

Eine weitere Zuordnungsmöglichkeit ergibt sich aufgrund der Sicherheitsrelevanz der Aufgaben, die ein eingebettetes System im Gesamtsystem übernimmt. Dies führt zur Einteilung in sicherheitskritische und nicht-sicherheitskritische eingebettete Systeme. Ein *sicherheitskritisches eingebettetes System* liegt vor, wenn von seiner korrekten Funktionsweise Menschenleben oder die Unversehrtheit von Einrichtungen abhängen. Während in der Konsumelektronik hauptsächlich nicht sicherheitskritische eingebettete Systeme zum Einsatz kommen, treten in der Avionik, Medizintechnik, insbesondere aber auch im Kraftfahrzeugbereich zunehmend sicherheitskritische eingebettete Systeme auf. Darüber hinaus lassen sich zeitkritische Systeme von nicht-zeitkritischen Systemen differenzieren. Bei letzteren spielt die Zeitspanne, innerhalb derer das System auf Eingabesignale reagiert, keine Rolle.

Damit zeichnet sich gerade die Automobiltechnik gegenüber den anderen später (siehe Abschnitt 1.3) genannten Produktkategorien durch zwei wesentliche Aspekte aus:

- Kraftfahrzeuge und damit die in ihnen (in Form von Steuergeräten) auftretenden eingebetteten Systeme bilden einen Massenmarkt.
- Eingebettete Systeme in Kraftfahrzeugen übernehmen zeit- und zunehmend auch sicherheitskritische Aufgaben.

Wir wollen uns daher bei unseren Betrachtungen im Folgenden vor allem auf eingebettete Systeme in der Automobiltechnik konzentrieren.

(Nicht-) Determinismus

Eingebettete Systeme müssen zu jedem Zeitpunkt ein *deterministisches*, also vorhersagbares Verhalten besitzen. Diese Eigenschaft gilt für ihre Realisierung, jedoch nicht notwendigerweise für ihre Anforderungs- oder Entwurfsspezifikation. In den Entwicklungsphasen der Anforderungsanalyse (engl. requirements analysis) bzw.

des Enwurfs (engl. design) kann die Spezifikation eines eingebetteten Systems durchaus Nichtdeterminismen aufweisen. Hier handelt es sich im Falle der Verhaltensspezifikation um das semantische Ergebnis gewünschter Unter- bzw. Überspezifikationen. Mit beiden Spezifikationsmitteln wird der Entwickler in die Lage versetzt, Teile des Systemverhaltens noch offen zu lassen, d. h. davon zu abstrahieren. Eine Konkretisierung der Spezifikation und damit Auflösung der Nichtdeterminismen muss zu einem späteren Zeitpunkt, spätestens jedoch zur Implementierung des Systems geschehen. Eine an das geschilderte Vorgehen angepasste Entwurfssmethode ist die sogenannte schrittweise Verfeinerung bzw. der inkrementelle Entwurf.

1.3 Anwendungen, Beispiele und Branchen

Anwendungsgebiete eingebetteter Systeme

Eingebettete Systeme sind heute wie bereits eingangs erwähnt und mit Zahlenmaterial unterlegt schon weit verbreitet. Sie sind insbesondere in folgenden Anwendungsgebieten zu finden:

- Telekommunikation (Vermittlungsanlagen, Mobiltelefone, Telefone, Faxgeräte etc.)
- Haushalt (Waschmaschine, Mikrowelle, Fernseher etc.)
- Geräte für Freizeit und Hobby
- Automobiltechnik (ABS, Wegfahrsperre, Navigationssysteme, Verkehrsleitsysteme etc.)
- Öffentlicher Verkehr (Fahrkartenautomat, etc.)
- Schienenverkehr (ICE, TGV)
- Luft- und Raumfahrttechnik, Avionik (Airbus-Reihe, Boeing 777, Militärsektor)
- Fertigungstechnik, Anlagenbau
- Steuerungs- und Regelungstechnik
- Medizintechnik
- Umwelttechnik
- Militärtechnik
- u. v. m.

Avionik und Automobilbau

Besonders interessante Anwendungsgebiete eingebetteter Systeme sind die *Avionik* (Flugzeugbau) und der *Automobilbau*. In einem modernen Fahrzeug sind heute oftmals mehr als 70 Mikroprozessoren oder elektronische Kontrolleinheiten integriert, die wir unter dem Begriff Steuergeräte zusammenfassen. Die Zahl der Prozessoren ist in den letzten Jahren sprunghaft gestiegen und wird auch in Zukunft noch weiterhin beständig anwachsen. Auf diesen Mikroprozessoren ist eine immer komplexer werdende Software implementiert, die im Auto Steuerungsfunktionen angefangen von der Zentralverriegelung über die Klimaanlage bis hin zur Motorsteuerung übernimmt. Die so entstehenden Kontrolleinheiten unterstützen den Fahrer in Standard- oder auch sicherheitskritischen Situationen.

Beispiel: Airbag

Ein konkretes Beispiel aus dem Automobilbau ist dabei folgendes: Nahezu alle heute gebauten Kraftfahrzeuge sind mit einem *Airbag-System* ausgestattet. Der Airbag bläst sich im Falle eines Unfalls mit hoher Geschwindigkeit auf und schützt so den Autoinsassen vor dem Aufprall auf das Lenkrad bzw. den Armaturenträger.

Das korrespondierende eingebettete System arbeitet in etwa wie folgt: Im Fahrzeugs sind mehrere Sensoren verbaut, die dessen Beschleunigung messen. Prallt ein Wagen auf ein Hindernis, entsteht eine negative Beschleunigung. Diese wird vom Sensor registriert, der einen Gasgenerator zündet und damit das Aufblasen des Airbags in Gang setzt. Der geschilderte Vorgang passiert in der sehr kurzen Zeit von nur wenigen Hundertstel Sekunden. Die hohe Aufblasgeschwindigkeit des Airbags stellt sicher, dass das Luftkissen bereits voll aufgeblasen ist, wenn der Oberkörper des Autoinsassen nach vorne geschleudert wird. Nach dem selben Prinzip funktionieren auch zusätzliche Airbags für den Seiten-, Brust-, oder Kopfbereich. Hier sind die Sensoren an den jeweiligen Stellen an den Seiten des Wagens angebracht.

Weitere Beispiele

Eingebettete Steuer- und Regelsysteme sind mittlerweile aber nicht nur in Kraftfahrzeugen und Flugzeugen verbaut, sondern in beinahe allen technischen Geräten des täglichen Lebens enthalten (Broy et al., 1998). Im Bereich der Konsumgüterelektronik (Unterhaltungs- und Haushaltsgerätebereich) sind Fotoapparate, Videokameras, HiFi- und TV-Geräte, Set-Top-Boxen (Decoder für digitale TV-Programme), Waschmaschinen, Wäschetrockner, Mikrowellengeräte, Staubsauger sowie Heizungssteuerungen prominente Beispiele für Produkte, die im Allgemeinen mehrere eingebettete Systeme enthalten.

Aus der großen Anzahl der hier aufgelisteten Produktkategorien und deren breiter Streuung über das gesamte Produktspektrum lässt sich bereits die Bedeutung heute im Einsatz befindlicher eingebetteter Systeme erahnen. Hinzu kommt, dass letztere insbesondere in Massenmärkten wie Konsumelektronik, Telekommunikation und Automobiltechnik weite Verbreitung gefunden haben und in immer größerem Umfang eingesetzt werden. Beispielsweise hat sich im Zeitraum zwischen 1985 und 1994 der Programmumfang, also der Software-Anteil, in Automobilsystemen der Firma Siemens Automotive S.A. (Toulouse) alle zwei bis drei Jahre verdoppelt, siehe (Siemens, 1994).

1.4 Begriffsdefinitionen

Zum besseren Verständnis der in diesem Buch verwendeten Begriffe geben wir im Folgenden eine Reihe entsprechender Definitionen an. Dieser Abschnitt ist insbesondere deshalb von Bedeutung, da sowohl im akademischen, als auch im industriellen Umfeld einige der Begriffe mit unterschiedlicher Bedeutung belegt sind. Die hier formulierten Definitionen beziehen sich überwiegend auf die informationsverarbeitenden Systeme (Hardware beziehungsweise Software) und weniger auf die technisch-physikalische Umgebung (Broy et al., 1998):

Definition (System)

Definition (System):
Unter einem System versteht man ein mathematisches Modell S, das einem Eingangssignal der Größe x ein Ausgangssignal y der Größe y=S(x) zuordnet.

Wenn das Ausgangssignal nur vom aktuellen Wert des Eingangssignals abhängt, so spricht man von einem *gedächtnislosen* System. Wenn aber dieser von vorhergehenden Eingangssignalen abhängt, so spricht man von einem *dynamischen* System.

Definition (reaktives System)

Definition (reaktives System):
Ein reaktives System kann aus Software und/oder Hardware bestehen und setzt Eingabeereignisse (deren zeitliches Auftreten meist nicht vorhergesagt werden kann) – oftmals aber nicht notwendigerweise unter Einhaltung von Zeitvorgaben – in Ausgabeereignisse um.

Definition (hybrides System)

Definition (hybrides System):
Systeme, die sowohl kontinuierliche (analoge), als auch diskrete Datenanteile (wertkontinuierlich) verarbeiten und/oder sowohl über kontinuierliche Zeiträume (zeitkontinuierlich), als auch zu diskreten Zeitpunkten mit ihrer Umgebung interagieren, heißen hybride Systeme.

Zur expliziten Unterscheidung zwischen beiden Formen hybrider Systemanteile werden die Begriffe “datenkontinuierlich/-diskret” oder „wertkontinuierlich/-diskret“ und “zeitkontinuierlich/-diskret” verwendet.

Definition (verteiltes System)

Definition (verteiltes System):
Ein verteiltes System besteht aus Komponenten, die räumlich oder logisch verteilt sind und mittels einer Koppelung bzw. Vernetzung zum Erreichen der Funktionalität des Gesamtsystems beitragen.

Die Entwicklung von Steuergeräten hat in der zweiten Hälfte des letzten Jahrhunderts einen bedeutenden Wechsel vollzogen. Angefangen von elektromechanischen Steuergeräten über elektrische Steuergeräte ist man heutzutage bei elektronischen (vollprogrammierbaren) Steuergeräten angelangt.

Definition (Steuergerät)

Definition (Steuergerät):
Ein Steuergerät (engl. Electronic Control Unit, kurz ECU) ist die physikalische Umsetzung eines eingebetteten Systems. Es stellt damit die Kontrolleinheit eines mechatronischen Systems dar. In mechatronischen Systemen bilden Steuergerät und Sensorik/Aktorik oft eine Einheit.

Steuergeräte sind im Prinzip wie folgt aufgebaut: Die Kernkomponente des Steuergeräts stellt ein Mikrocontroller oder Mikroprozessor (Beispiele: Power PC, Alpha PC) dar. Zusätzlich kann es optional ein externes RAM und/oder ROM besitzen sowie sonstige Peripherie und Bauelemente.

Definition (mechatronisches System, Mechatronik)

Definition (mechatronisches System, Mechatronik):
Wird Elektronik zur Steuerung und Regelung mechanischer Vorgänge räumlich eng mit den mechanischen Systembestandteilen verbunden, so sprechen wir von einem mechatronischen System. Der Forschungszweig, der sich mit den Grundlagen und der Entwicklung mechatronischer Systeme befasst, heißt Mechatronik.

Der Begriff *Mechatronik* (engl. mechatronics) ist ein Kunstwort bestehend aus *Mecha*nik und Elek*tronik*. Bei der Mechatronik handelt sich um ein interdisziplinäres Gebiet der Ingenieurwissenschaften, das auf Maschinenbau, Elektrotechnik und der Informatik aufbaut. Mechatronische Systeme enthalten in zunehmendem Maße hierarchisch angeordnete, untereinander gekoppelte eingebettete Systeme. Ein typisches mechatronisches System nimmt Signale auf, verarbeitet sie und gibt wiederum Signale aus, die dann ggf. in Kraft oder Bewegung umgesetzt werden.

1.5 Logischer Aufbau eingebetteter Systeme

In diesem Abschnitt charakterisieren wir die grundlegenden Bestandteile eingebetteter Systeme und erklären die daraus resultierende logische Strukturierung der Anforderungen. Nach der Lektüre dieses Abschnittes sollte der Leser den prinzipiellen logischen Aufbau eines eingebetteten Systems und dessen wesentliche Bestandteile kennen.

Beim Einsatz von Hardware in eingebetteten Systemen ist auf die *Umgebungsbedingungen am Einsatzort* besonders zu achten. Sie muss unter anderem robust gegen folgende Störfaktoren sein: Wärme bzw. Kälte, Staub, Feuchtigkeit, Spritzwasser, mechanische Schwingungen bzw. Stöße, Fremdkörper und elektromagnetische Störungen (EMS). Insbesondere in der Automobilindustrie gibt es hier genaue, oftmals Hersteller-spezifische Vorschriften, die diesbezüglich genaue Vorgaben enthalten. Sie können durch physikalische (z. B. durch Schirmung bzw. Gehäuse) und geometrische (beispielsweise Vermeidung von EMS durch Verdrillen der Kabel) Maßnahmen eingehalten werden.

Die 5 Strukturbestandteile eingebetteter Systeme

Ein eingebettetes System kann in der Regel in die folgenden fünf strukturellen Bestandteile zerlegt werden (Broy und Scholz, 1998):

- Die Kontrolleinheit bzw. das Steuergerät, d. h. das eingebettete Hardware/Software System,
- die Regelstrecke mit Aktoren (auch: Aktuatoren) und Sensoren, d. h. das gesteuerte physikalische System,
- die Benutzerschnittstelle,
- die Umgebung sowie
- den Benutzer.

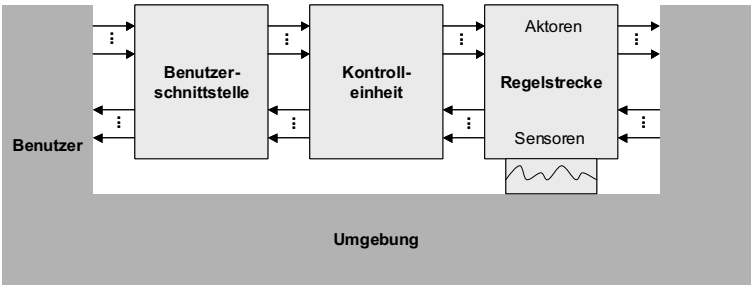


Abb. 1.2: Logische Referenz-architektur eines eingebetteten Systems

Grobarchitektur

Die *Grobarchitektur* eines eingebetteten Systems unterteilt das System in diese Bestandteile und beschreibt, wie sie miteinander verbunden sind. Abbildung 1.2 zeigt diese Architektur als Datenflussdiagramm. Die gerichteten Pfeile stellen gerichtete Kommunikationskanäle dar. Auf diesen Kanälen werden Ströme diskreter Nachrichten oder kontinuierlicher Signale („diskret" bedeutet hier immer ereignisdiskret, „kontinuierlich" meint zeit- und wertkontinuierlich) übermittelt. Die gezackte Linie, welche die Kommunikationskanäle zwischen Regelstrecke und Umgebung markiert, deutet an, dass diese Bestandteile auf sehr komplexe, schwer formalisierbare Weise in Wechselwirkung stehen können. Die Kontrolleinheit ist nicht direkt mit der Umgebung verbunden, sondern nur mit der Benutzerschnittstelle und der Regelstrecke. Die Grauschattierung in der Zeichnung weist darauf hin, dass wir zwischen Benutzer und Systemumgebung nicht klar trennen.

Wirkungskette

Wie wir der Abbildung 1.2 entnehmen können, handelt es sich beim Ablauf eines eingebetteten Systems um eine geschlossene Wirkungskette, vgl. Abbildung 1.3. Die Aufgaben dieser Wirkungskette sind:

- Die Erfassung von Eingabeereignissen durch Sensoren,
- die steuernde Einwirkung auf den zu kontrollierenden Prozess durch Aktoren und
- die Umwandlung (kurz: Wandlung) analoger elektrischer Größen der Eingabeereignisse in digitale Signale zur Verarbeitung durch Rechner (die sogenannte Analog-Digital/Digital-Analog-Wandlung, kurz AD/DA-Wandlung).

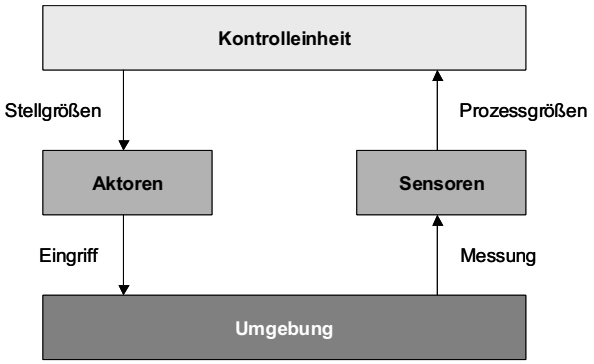


Abb. 1.3: Wirkungskette System/ Umgebung

Der zu steuernde technische Prozess ist also über Sensoren und Aktoren an das Steuergerät gekoppelt und kommuniziert über diese mit ihm. Sensoren und Aktoren fasst man unter dem Begriff *Peripherie* oder *I/O-System* zusammen. I/O (engl. Input/Output) steht hier für Eingabe/Ausgabe (kurz: E/A).

1.5.1 Kontrolleinheit

Die Kontrolleinheit empfängt Signale von der Benutzerschnittstelle und den Sensoren an der Regelstrecke. Diese Eingaben werden verarbeitet und Antwortsignale werden an die Benutzerschnittstelle und die Aktoren an der Regelstrecke geschickt. Die Kontrolleinheit bildet den Kern des eingebetteten Systems. Selbstverständlich muss die Kontrolleinheit keine monolithische Komponente sein. Sie kann im Entwurf durchaus in ein Netzwerk von parallelen, örtlich verteilten Subkomponenten zerlegt werden, die mit Benutzerschnittstelle und Regelstrecke interagieren. Fragen der Parallelität und der Verteilung sind beim Entwurf von eingebetteten Systemen allerdings Teil der sogenannten „Glass-Box-Sicht“ auf die Kontrolleinheit. In den frühen Phasen der Systementwicklung sind wir dagegen hauptsächlich an der „Black-Box-Sicht“ interessiert. Eine direkte Verbindung von Benutzerschnittstelle und Regelstrecke kann als Spezialfall der Kontrolleinheit modelliert werden. Wie wir bereits gesehen haben, stellen Steuergeräte die Kontrolleinheit eines mechatronischen Systems dar.

Moderne PKWs der Oberklasse enthalten inzwischen bis zu 100 Steuergeräten (ECU, Electronic Control Unit), Tendenz steigend. Damit einhergehend werden ca. 3 km Kabel und ca. 2.000 Steckverbindungen verbaut. In den Steuergeräten reguliert Software

mit mehr als 600.000 LOCs (Lines of Code, Quellcodezeilen) zahlreiche Funktionen sowie deren Zusammenspiel. Das umfasst u. a. die Regelung der Bremskraft, die Zentralverriegelung des Fahrzeugs und die Funktion des Scheibenwischers (Quelle: DaimlerChrysler).

Damit ändert sich auch signifikant die Wertschöpfung im Automobilbau. 90% der Innovationen im Fahrzeug werden durch Elektronik geprägt, davon 80% im Bereich der Software. Dabei wächst der Anteil der Software an der Produkt-Wertschöpfung stetig (Quelle: Audi).

Bis 2010 prognostizieren Analysten eine jährliche Wachstumsrate für elektronische Funktionen im Auto von mehr als 10% pro Jahr. Der Elektronikanteil der Automobilproduktionskosten wird sich von 22 bis 25% im Jahre 2000 auf 35 bis 40% im Jahre 2010 erhöhen. Der Gesamtmarkt für Elektronik und Software soll in diesem Bereich um 115% bis auf 270 Mio. Euro zunehmen (Quelle: TTTech).

Grundlagen und Aufbau von Mikrocomputern bzw. Mikroprozessoren und somit der Kontrolleinheit sind nicht Inhalt dieses Buches und können beispielsweise in (Flynn et al, 2001), (Kelch, 2003), (Kelch, 2003a), (Märtin, 2003), (Stallings, 2000), (Tanenbaum, 2001) oder (Zargham, 1996) studiert werden. Mikrocontroller kennzeichnen eine Klasse von Mikroprozessoren, die auf den speziellen Anwendungsbereich der Steuerung von Prozessen zugeschnitten sind. Wir betrachten im Folgenden einige Spezialfälle genauer.

Definition (ASIP)

Definition (Application Specific Instruction Set Processor, ASIP): Ein ASIP (applikationsspezifischer Prozessor) ist ein Prozessor, der von seiner Struktur als auch von seinem Befehlssatz her auf seinen Einsatz für bestimmte Anwendungen hin optimiert ist.

ASIPs besitzen spezielle Instruktionssätze, funktionale Einheiten, Register und spezielle Verbindungsstrukturen. Diese Spezialisierung führt zu Architekturen, die sich von denen eines Mikroprozessors stark unterscheiden. ASIPs stellen hinsichtlich Flexibilität und Performanz die Nahtstelle von der Software- zur Hardwareseite her. Aus Kostengründen ist ein ASIP oft nur ein um verschiedene Umfänge reduzierter Prozessor und damit günstiger als ein „Vielzweckprozessor“ (engl. multi purpose processor), aber aufgrund seiner Programmierbarkeit immer noch flexibler als dedizierte Hardware. Er leistet eine höhere Verarbeitungsgeschwindigkeit und benötigt eine geringere Leistungsaufnahme aufgrund

optimierter Strukturen. Sein Nachteil ist die komplizierte und aufwändige Entwicklung.

Zu der Klasse anwendungsspezifischer Prozessoren gehört auch die Klasse der digitalen Signalprozessoren (DSPs).

Definition (DSP)

Definition (Digitaler Signal Prozessor, DSP):
Ein DSP ist ein spezieller Mikroprozessor, der Befehle (und damit Verarbeitungseinheiten) zur Durchführung von Signalverarbeitungsaufgaben besitzt. Er ist dabei auf die häufig vorkommenden Operationen bei der digitalen Signalverarbeitung wie z. B. die schnelle Fourier Transformation (FFT) spezialisiert. Der Effizienz der Operationen kommt hierbei ein hoher Stellenwert zu.

Digitale Signal Prozessoren werden beispielsweise in MP3-Decodern, bei der Bildverarbeitung sowie in der Sprachsignalverarbeitung eingesetzt.

Definition (ASIC)

Definition (Application Specific Integrated Circuit, ASIC):
Ein ASIC ist eine anwendungsspezifische integrierte Schaltung (engl. Integrated Circuit, kurz: IC), die für spezielle Anwendungen entwickelt wurde und spezielle Schaltungen enthält, die hohe Leistungs- und Zeitanforderungen erfüllen können. Er kann wie ein ASIP programmierbar oder auch festverdrahtet sein.

Definition (FPGA)

Definition (Field Programmable Gate Array, FPGA):
Das Field Programmable Gate Array (FPGA) ist ein komplexer, programmierbarer Logikbaustein, der zum Aufbau digitaler, logischer Schaltungen dient. Er besteht im Wesentlichen aus einzelnen Funktionsblöcken, die in einer regelmäßigen Struktur (engl. Array) angeordnet sind und einem Netzwerk von Verbindungen zwischen diesen Blöcken. Die speziellen Funktionen der einzelnen Blöcke und die Auswahl der benötigten Verbindungen sind programmierbar.

Bei FPGAs wird die Implementierung von logischen Funktionen hauptsächlich durch die Programmierung der Verbindungsleitungen zwischen den Logikblöcken erreicht. Damit kann in das FPGA eine bestimmte vom Anwender entwickelte Schaltung implementiert werden. Man spricht hier auch von einer Personalisierung des Bausteins.

FPGA-Arten

Es gibt zwei verschiedene Arten von FPGAs, rekonfigurierbare sowie nicht-rekonfigurierbare. Für die erstgenannten verwendet man flüchtige Speichertechnologien wie SRAM. Ihr Nachteil ist, dass die Programmierung nur solange im Baustein anhält, wie auch eine

Stromversorgung besteht. Wird sie unterbrochen, so muss der Baustein neu konfiguriert (rekonfiguriert) werden bzw. seine Konfiguration neu eingelesen werden.

Die nicht-rekonfigurierbaren FPGAs können dagegen nur einmal programmiert werden, behalten ihre Programmierung dafür aber für immer und unabhängig von der Stromversorgung. Dies gelingt meist durch die physikalische Zerstörung nicht gebrauchter Leitungen; die Schaltungskonfiguration ergibt sich somit indirekt.

Im Gegensatz zu gewöhnlichen Gate Arrays (GA) sind FPGAs *programmierbare* Logikbausteine, deren Funktionalität durch das Zusammenschalten verschiedener Funktionsblöcke erreicht wird. Während PLAs (engl. Programmable Logic Arrays) nur Boolesche Funktionen in zweistufiger Form realisieren können, kann man mit FPGAs auch Speicherzellen realisieren und so eignen sich FPGAs auch zur Realisierung von Steuerwerken (in Form endlicher Automaten).

1.5.2 Regelstrecke

Die Regelstrecke zeichnet sich im Allgemeinen durch ihren heterogenen Charakter aus. Sie kann aus elektronischen, elektrischen und mechanischen Teilen bestehen. Anders als in der Regelungstechnik betrachten wir beim Softwareentwurf von eingebetteten Systemen die Sensoren und Aktoren als Teil der Regelstrecke. Zumindest aus der Sicht der Kontrolleinheit kann die Regelstrecke daher als Komponente gesehen werden, die diskrete und/oder kontinuierliche Kontrollsignale erhält und über ihre Sensoren diskrete und/oder kontinuierliche Signale ausgibt.

Zusätzlich steht die Regelstrecke auch mit ihrer Umgebung in Wechselwirkung. Es ist eine wesentliche und manchmal schwierige Entscheidung des Systemanalytikers, die Grenze zwischen Regelstrecke und Umgebung festzulegen. Wenn diese Grenze bestimmt ist, muss die Interaktion zwischen Umgebung und Regelstrecke modelliert werden.

Diese Wechselwirkung kann durch komplexe kausale Abhängigkeiten zwischen komplizierten physikalischen oder chemischen Prozessen charakterisiert sein, wie z. B. beim Motormanagement von Kraftfahrzeugen. Daher kann es in manchen Fällen sehr schwierig sein, ein Modell zu finden, in dem die Kommunikation zwischen Regelstrecke und Umgebung dargestellt werden kann.

Wir nehmen deshalb vereinfachend an, dass die Interaktion zwischen beiden Komponenten allein durch eine Zahl diskreter

und/oder kontinuierlicher Ströme modelliert werden kann. Diskrete Ströme von Nachrichten codieren dabei Ereignisse, kontinuierliche Ströme (Funktionen über der Zeit) modellieren sich stetig ändernde Parameter des Systemzustandes. Unterschiedliche mögliche Reaktionen (man spricht hier auch von „Unterspezifikation“) der Umgebung oder Unwissen über ihre genauen Reaktionen können durch Nichtdeterminismus oder durch Prädikate, die Mengen von möglichen Verhalten beschreiben, modelliert werden.

In Fällen, in denen ein vereinfachtes Modell nach diesem Muster nicht geeignet ist, kann versucht werden, die Regelstrecke sehr klein zu wählen, oder allein die Interaktion von Kontrolleinheit und Regelstrecke ohne die Umgebung zu untersuchen. Der Einfluss von externen Störungen muss dann gesondert betrachtet werden. Eine andere Möglichkeit ist, die Regelstrecke wesentlich größer zu wählen und einen großen Teil der Umgebung in sie zu integrieren. Dieser Ansatz kann dann zu einer leichter handhabbaren Schnittstellenbeschreibung der Regelstrecke führen. Schließlich kann es in manchen Fällen auch notwendig sein, mit Wahrscheinlichkeiten zu arbeiten, und die Systemanforderungen mit Wahrscheinlichkeiten auszudrücken.

1.5.2.1 Peripherie

Sensoren und Aktoren können sowohl jeweils *einzeln* als auch über *Bussysteme* verschiedenartiger Topologie (Bus, Ring usw.) mit Steuergeräten verbunden sein.

Die Kommunikationsschnittstelle zwischen Steuergerät und technischem Prozess ist in Abbildung 1.3 stark vereinfacht dargestellt. Sie zeigt den Spezialfall der *digitalen Schnittstelle*, ist aber Grundlage sowohl für den Aufbau als auch für das Verständnis für alle weiteren Ein-/Ausgabeschnittstellen. Bei ihr handelt es sich um die einfachste Ein-/Ausgabemöglichkeit, weil das Steuergerät selbst mithilfe eines speziellen I/O-Registers (Eingabe-/Ausgabe-Registers) über digitale elektrische Signale mit dem Prozess kommuniziert.

Gängige Spannungen für die *digitale Ein- bzw. Ausgabe* sind beispielsweise 0Volt/230Volt, 0V/5V bei TTL-Technologie oder 0V/12V im Falle des Bordspannungsnetzes eines Kraftfahrzeugs.

Da die im realen technischen Prozess vorkommenden Signale rein analoger Form sind, Steuergeräte genauso wie andere Mikrocomputer aber nur digitale Signale verarbeiten können, sind weitere technische Hilfsmittel notwendig, um eine *Konvertierung*

(Wandlung) der Signale (von analog nach digital und vice versa) vornehmen zu können.

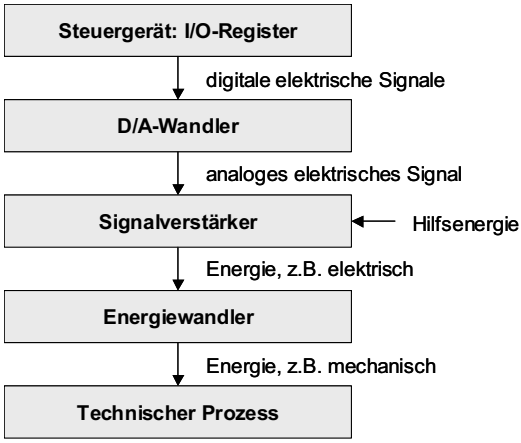


Abb. 1.4: Signal-verarbeitungs-kette mit D/A-Wandler

1.5.2.2 Digital/Analog-Wandler

Digital/Analog-Wandler (kurz: D/A-Wandler) erzeugen aus digitalen Signalen ein analoges Signal (Spannung). Mit ihrer Hilfe lässt sich eine komplexere Signalverarbeitungskette darstellen (vgl. Abbildung 1.4).

D/A-Wandler können auf verschiedene Weise in die Wirkungskette zwischen Steuergerät und Sensor bzw. Aktor integriert sein; Abbildung 1.5 beschreibt diese.

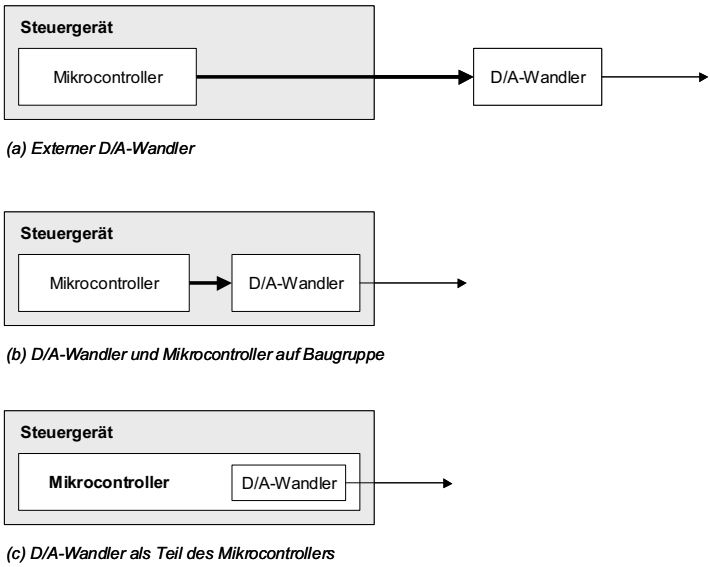


Abb. 1.5: Integration von D/A-Wandlern, nach (Bender, 2003)

Im ersten Fall (a) handelt es sich beim D/A-Wandler um eine eigene Komponente, die außerhalb des Steuergeräts angesiedelt ist. Im zweiten Fall (b) ist der D/A-Wandler zusammen mit dem Mikrokontroller auf derselben Baugruppe innerhalb des Steuergeräts aufgebracht. Die höchste Form der Integration wird erreicht, wenn (c) der D/A-Wandler Teil des Mikrocontrollers selbst ist.

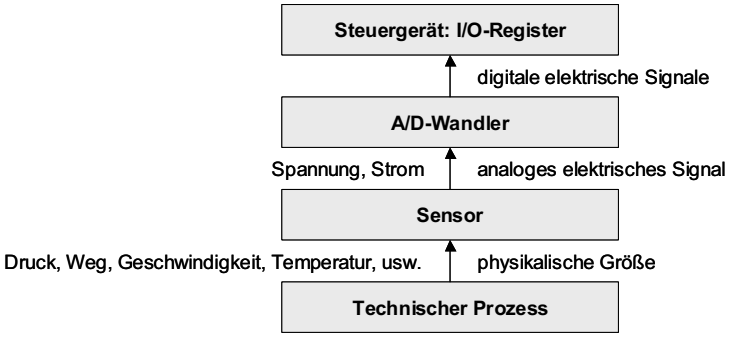
PCM-Wandler, DM-Wandler

Eine weitere Möglichkeit, Wandler zu unterscheiden, ist in PCM- und DM-Wandler. *PCM-Wandler* (PCM = Pulse Code Modulation) sind Wandler, bei denen am Ausgang eine duale Codierung des Analogwertes am Eingang, also der Eingangsspannung anliegt. Die Ausgabe eines *DM-Wandlers* (Delta Modulation) beschreibt die Differenz zum Wert des vorangegangenen Abtastzeitpunktes. Das Wandler-Ausgangssignal codiert also eine Differenz, die aus den Werten des Eingangssignals zwischen zwei Abtastzeitpunkten bestimmt wird. Der Vorteil der Delta Modulation liegt in der Datenreduktion, die besonders wichtig bei höheren Abtastraten wie z. B. von Audio-, oder Videosignalen wird. Die grundlegende Idee hierbei ist, dass die zu wandelnden Signale i.d.R. eine hohe Datenredundanz aufweisen, weil benachbarte Abtastwerte nur geringfügig voneinander abweichen.

1.5.2.3 Analog/-Digital-Wandler

Analog/Digital-Wandler (kurz: A/D-Wandler) erzeugen aus einem analogen Signal (in Form von Spannung oder Strom) digitale Signale. Ihre Funktionsweise ist analog zu D/A-Wandlern. Auch sie dienen zur Abbildung komplexerer Signalverarbeitungsketten (siehe Abbildung 1.6). Hinsichtlich ihrer Integration in diese Wirkungskette sind die gleichen drei Alternativen wie beim D/A-Wandler möglich (vgl. Abbildung 1.5).

Abb. 1.6: Signalverarbeitungskette mit A/D-Wandler



1.5.2.4 Sensoren

Grundsätzlich unterscheidet man zweierlei Betriebsarten, wie das Steuergerät Informationen in Form von Signalen von Sensoren erhält: Den Interrupt-Betrieb sowie den Polling-Betrieb.

Beim *Interrupt-Betrieb* führt ein vom Sensor aufgenommenes Ereignis dazu, dass das Steuergerät seine aktuelle Verarbeitung unterbricht (engl. to interrupt) und auf das Ereignis reagiert. Das Steuergerät nimmt hier eine passive Rolle ein.

Beim *Polling-Betrieb* fragt das Steuergerät in regelmäßigen, meist sehr kurz aufeinander folgenden zeitlichen Abständen selbst aktiv beim Sensor nach, ob ein neues Ereignis anliegt (engl. request) und erhält eine Antwort (engl. response) vom Sensor.

Definition (Sensor)

Definition (Sensor), nach (Schürmann, 2001):
Ein Sensor ist eine Einrichtung zum Feststellen von physikalischen oder chemischen Eingangsgrößen, die optional eine Messwertzuordnung (Skalierung) der Größen treffen kann, sowie ggf. ein digitales bzw. digitalisierbares Ausgangssignal liefern kann.

Der Begriff Sensor kommt dabei vom lateinischen „Sensus“, der Sinn. Die IEC (International Electronical Commission) definiert den Begriff Sensor wie folgt: Das primäre Element in einer Messkette, das eine variable, im Allgemeinen nichtelektrische Eingangsgröße in ein geeignetes Messsignal, insbesondere einer elektrischen, umsetzt.

Der Sensor ist damit der Spezialfall des „Transducers“, der eine Energieform in eine andere umwandelt. Ein Sensor ist im Allgemeinen ein preiswerter und zuverlässiger Messwertaufnehmer mit ausreichender Genauigkeit, der für die Massenherstellung geeignet ist.

Definition (Sensorik)

Definition (Sensorik)
Unter Sensorik versteht man die Signalaufnahme von der Umwelt mit optionaler Signalverarbeitung für den Einzelsensor oder Sensorsysteme.

Vorsicht: Unter Sensorik wird in anderen naturwissenschaftlichen Teilgebieten oft auch der Einsatz menschlicher Sinne (Riechen, Schmecken usw.) zu Prüf- und Messzwecken verstanden.

Klassifizierung von Sensoren nach Ausgabe, Messgröße, Wirkungsprinzip

Sensoren können nach unterschiedlichen Maßgaben klassifiziert werden. Eine erste Möglichkeit ist nach der *Ausgabe*, der *Messgröße* und dem *Wirkungsprinzip* (Schürmann, 2001).

Betrachten wir zunächst die *Art der Ausgabe* am Ausgang des Sensors. Ein Sensor kann mechanische, pneumatische, hydraulische oder elektrische Werte liefern, wobei wir im Folgenden ausschließlich elektrische Werte betrachten werden.

Bei der Art der *Messgröße* können wir nach Weg, Winkel, Kraft, Druck, Beschleunigung, Temperatur, Gas, usw. unterscheiden.

Mögliche *Wirkungsprinzipien* des Sensors sind ohmsch, kapazitiv, induktiv, optoelektronisch, thermoelektrisch, piezoelektrisch, etc.

Klassifikation von Sensoren nach Energienutzung

Das gemeinsame Merkmal aller Sensortypen ist die Verknüpfung der Messgröße mit einer elektrischen Ausgangsgröße. Ihr Unterscheidungsmerkmal ist die *Art der Nutzung der Energie* der Messgröße, die in elektrische Energie gewandelt wird:

- **Rezeptiver Sensor:** Nimmt ein vorhandenes Signal auf und wandelt ggf. nur um. Die Signalverarbeitung erfolgt anschließend. Rezeptive Sensoren werden auch passive Sensoren bzw. Wandler genannt. Beispiele für sie sind Kameras und Mikrophone.
- **Signalbearbeitender Sensor:** Stimuliert die Umwelt und nimmt die „Antwort" auf. Er benötigt hierfür eine zusätzliche Energie- oder Signalquelle. Beispiele sind Ultraschall-Sensoren oder Laser-Scanner. Signalbearbeitende Sensoren werden auch als aktive Sensoren bzw. Wandler bezeichnet.

Klassifikation von Sensoren nach Anwendung

Eine weitere Klassifikationsmöglichkeit für Sensoren ergibt sich nach der *Art ihrer Anwendung* (Schürmann, 2001):

- **Interne Sensoren** sind Sensoren, welche die inneren Zustände (Position, Geschwindigkeit, Beschleunigung, Orientierung) einer Maschine erfassen. Beispiele hierfür sind das Erfassen der Gelenkstellung, der Gelenkgeschwindigkeit und der Gelenkbeschleunigung eines Roboterarms.
- **Externe Sensoren** dagegen erfassen (aus der Sicht des Gesamtsystems) äußere Zustände, also Zustände der Umwelt. Beispiele hierfür sind das Messen des Abstands zwischen Fahrzeug und umgebenden Hindernissen (beispielsweise andere Fahrzeuge) z. B. mit Radar, die (optische) Werkstückidentifikation an einem Fertigungsabschnitt oder das Feststellen des Tankfüllstand eines Kraftfahrzeugs.

Die letzte Klassifikationsmöglichkeit ergibt sich durch die *Analyse des Wertebereichs* der Sensoren (Schürmann, 2001): Hier kann zwischen binären (Beispiel: Lichtschranke), skalaren (Beispiele: Winkel-, Abstandssensor), vektoriellen (Beispiel: Gyrosensor) und Muster-erkennenden Sensoren (Beispiele: Laserscanner, Videocamera) unterschieden werden.

Klassifikation von Sensoren nach Wertebereich

1.5.2.5 Aktuatoren

Aktuatoren bzw. Aktoren sind die Verbindungsglieder zwischen dem informationsverarbeitenden Teil eines eingebetteten Systems und dem Prozess. Sie wandeln die Energie, die an ihrem Eingang anliegt, in mechanische Arbeit um.

Beispiele für Aktuatoren sind Elektromotoren, Elektromagneten, hydraulische Aktuatoren, piezoelektrische Aktuatoren u.v.m.

Üblicherweise sind Aktuatoren mit *Standardschnittstellen* (wie etwa RS232 oder USB) ausgestattet, über die, mehr oder weniger aufwändig, eine Systemsteuerung möglich ist. Der Umfang der möglichen Instruktionen ist meist Sensor-abhängig, zur Erstellung eines Steuerungsablaufs ist ein genaues Studium der Sensorbeschreibung zwingend. In den ersten Generationen intelligenter Aktoren wurden aus Performanzgründen oft für den menschlichen Benutzer kryptische Kommandos aus Binärsequenzen verwendet. Heutzutage sind dies textuelle Kommandos, die weitgehend selbsterklärend sind.

1.5.3 Benutzerschnittstelle

Die meisten eingebetteten Systeme interagieren nicht nur mit der Regelstrecke, sondern reagieren auch auf Benutzereingaben und liefern Ausgaben, um den Benutzer über den gegenwärtigen Systemzustand zu informieren. Diese Kommunikationsaufgaben übernimmt die Benutzerschnittstelle.

Ebenso wie die Regelstrecke kann auch die Benutzerschnittstelle nicht als ein reines Hardware oder Software System gesehen werden. Im Allgemeinen besteht es aus einer Menge von Eingabegeräten (Schaltern, Tasten, Joysticks, etc.) und Ausgabegeräten (Signalleuchten, Bildschirmen, etc.). Oft gibt es Teile des Gesamtsystems, die sowohl als Teil der Benutzerschnittstelle, als auch als Teil der Regelstrecke gesehen werden können. Auch hier muss der Systemanalytiker eine geeignete Aufteilung festlegen. Natürlich kann auch

die Benutzerschnittstelle in eine Menge von dezentralen Bedienstationen zerlegt werden.

1.6 Softwareentwicklung eingebetteter Systeme

Die Entwicklung der Software eingebetteter Systeme ist eine komplexe Aufgabe, die derzeit noch nicht in allen Teilen zufriedenstellend beherrscht wird. Um die korrekte Funktionsweise solcher Systeme und kurze Entwicklungszeiten zu gewährleisten, ist es erforderlich, den Systementwickler in allen Phasen der Entwicklung möglichst gut methodisch und technisch durch ein passendes Werkzeug (CASE-Tool, CASE = Computer Aided Software Engineering) zu unterstützen.

1.6.1 Motivation

Die Software in eingebetteten Systemen muss sorgfältig auf Verbraucherwünsche und -anforderungen abgestimmt werden. Beta-Tests, wie sie etwa in der PC-Standardsoftwareindustrie weit verbreitet sind, können zum Test von Software in eingebetteten Systemen nicht verwendet werden. Kunden können im täglichen Verkehr schließlich kaum als Testpersonen fungieren. Aus diesen Gründen erfordert die Entwicklung von Software für eingebettete Systeme große Umsicht und Sorgfalt. Mittel- und langfristig wird es nur dann gelingen, zuverlässige reaktive, eingebettete Systeme herzustellen, wenn dem Entwickler ein speziell darauf angepasster Entwicklungsprozess zur Verfügung steht. Aufgrund der hohen Anforderungen an Nutzernähe und Zuverlässigkeit ist es unerlässlich, dass alle Konzepte und Anforderungen eines reaktiven Systems so früh wie möglich und beständig im Entwicklungsprozess validiert und verifiziert werden.

1.6.2 Begriffsklärung

Spezifikation

Unter einer *Spezifikation* versteht man die Festlegung (formal mit eindeutiger Semantik) was das System können muss. Dies umfasst das gewünschte Verhalten ebenso wie die Schnittstellen (Zahl und Art der Ein-/Ausgänge, evtl. sogar Semantik) sowie evtl. auch Vorgaben bzgl. Geschwindigkeit, Kosten, Fläche, Leistungsverbrauch usw.

Korrektheit

Die *Korrektheit* (oder der Nutzen) eines eingebetteten Systems muss durch die Beobachtung des Verhaltens des Gesamtsystems, also der Reaktionen auf Eingaben von Benutzer und Umgebung, bestimmt werden. Wir sind normalerweise nicht am Verhalten der Kontrolleinheit, sondern vielmehr am Verhalten der Regelstrecke unter dem Einfluss der Kontrolleinheit interessiert. Wenn die Regelstrecke unerwünschtes Verhalten zeigt, so ist der Entwurf ungeeignet. Beobachtungen müssen dabei auch die Eingaben von Benutzer und Umgebung umfassen. Nur relativ zu diesen Eingaben ist es möglich, festzulegen welches Verhalten die Regelstrecke jeweils zeigen soll. Mögliche Gründe für falsches oder unerwartetes Verhalten sind: Die Annahmen über das Verhalten der Regelstrecke sind falsch; die Kontrolleinheit sendet nicht die richtigen Signale oder sendet sie nicht rechtzeitig; die Benutzerschnittstelle übermittelt nicht die richtige Information zwischen Benutzer und Kontrolleinheit.

Implementierung

Mit dem Begriff der *Implementierung* meinen wir die reine Umsetzung der Spezifikation in eine konkrete Realisierung des zu entwerfenden Systems.

1.6.3 Entwurf

Der Entwurf eingebetteter Systeme ist durch seine hohe Komplexität gekennzeichnet. Diese Komplexität kann durch einen hierarchischen Entwurfsstil mit Simulation und Optimierung auf mehreren Ebenen bewältigt werden. Den einzelnen Ebenen werden spezifische Modellvorstellungen (Sichten, Eigenschaften, Domänen) zugeordnet.

Software wird nicht in einem einzigen Schritt entwickelt, sondern man unterteilt den Entwicklungsprozess für Software in mehrere Phasen. Hierbei haben sich einige Standardvorgehensweisen wie beispielsweise das Wasserfall-, Spiral-, oder V-Modell etabliert. Man unterteilt solche Prozesse beispielsweise in die Phasen

Anforderungserfassung (Requirements Capture), Systemanalyse, Entwurf (Design), Validierung, Verifikation, Prototypengenerierung, Simulation und Implementierung.

Bei der Entwicklung eingebetteter Systeme ist die Spezifikation der geforderten Funktionalität eine besonders kritische Aufgabe. Statistiken zeigen, dass in typischen Anwendungsgebieten mehr als 50% der Fehler, die in den ausgelieferten Systemen auftauchen und von den Kunden gemeldet werden, keine Fehler bei der Implementierung, sondern konzeptionelle Fehler in den erarbeiteten Anforderungen sind. Eingebettete Systeme, insbesondere solche in sicherheitskritischen Bereichen, erfordern ein hohes Maß an Zuverlässigkeit. Basis dieser Zuverlässigkeit ist die sorgfältige Erfassung der Anforderungen der Benutzer. In diesem Beitrag konzentrieren wir uns daher auf die beiden Phasen Anforderungsanalyse und Entwurf. In diesen Phasen spielen Modellierungstechniken eine wichtige Rolle. Folgende Aspekte stellen bei solchen Techniken eine besondere Herausforderung dar:

- Modellierung von Verhalten (digital, analog) und Zeit (diskret, kontinuierlich);
- Modularität; Integration verschiedener Modelle;
- Abstraktion und Verfeinerung; Eignung für (werkzeugunterstützte) Validierung und Verifikation;
- Anschluss an eine automatische Codegenerierung.

1.7 Besondere Herausforderungen

Beobachtungen

Bei der näheren Betrachtung der gegenwärtigen Situation der Entwicklung eingebetteter Systeme ergeben sich folgende Beobachtungen (Broy und Scholz, 1998):

- Der Kostenanteil der Software im Vergleich zur Hardware steigt.
- Aufgabenkomplexität und Vernetzungsgrad nehmen zu.
- Die Komplexität der Steuergeräte wächst exponentiell und gleichzeitig sollen Entwurfszeiten signifikant verkürzt werden.
- Eingebettete Systeme interagieren und verschmelzen mit anderen Softwaresystemen.
- Anwendungsgebiete und Funktionsumfang erweitern sich.

- Neuartige Modellierungsparadigmen sind zur Komplexitätsbewältigung erforderlich.
- Verschiedene Disziplinen wie etwa Kontroll- und Regelungstechnik, Schaltungstechnik, Software Engineering, Betriebssysteme, Compilerbau und Maschinenwesen wachsen zusammen.

Herausforderungen

Auf Basis dieser Beobachtungen ergeben sich für die Zukunft im Bereich des Software Engineering für eingebettete Systeme folgende Herausforderungen (Broy und Scholz, 1998):

- Die „Time to Market" muss verkürzt und die Nutzungsadäquanz (Nutzerakzeptanz) erhöht werden.
- Die Beherrschbarkeit des Entwicklungsprozesses (Kosten, Termine, Qualität) ist zu verbessern.
- Die Durchgängigkeit des Entwicklungsprozesses muss hergestellt werden.
- Die Modellierung, insbesondere von Zeit-, Reaktivitäts- und Verteilungsaspekten muss verbessert werden.
- Die Entscheidung, welche Teile des Systementwurfs in Hardware und welche in Software realisiert werden sollen, muss so spät wie möglich im Entwicklungsprozess fallen (Stichwort „Hardware/Software-Codesign).
- Sicherheit, Zuverlässigkeit, Vorhersagbarkeit, Adaptierbarkeit, Modifizierbarkeit (Design to Change) der Entwicklungen sind zu erhöhen und eine größere Arbeitsteilung ist zu gewährleisten.
- Modellierungskonzepte für Verteiltheit, Mobilität und Dynamik sind nötig.

1.8 Zusammenfassung

Eingebettete Systeme sind in zahlreichen Gebieten des täglichen Lebens zu finden. Sie fallen in die Klasse der reaktiven Systeme und grenzen sich daher von transformationellen und interaktiven Systemen deutlich ab. Eingebettete Systeme müssen teilweise in Echtzeit bestimmte Anforderungen lösen. In diesem Fall spricht man von Echtzeitsystemen. Aufgrund dieser Unterschiede besitzen eingebettete Systeme andere Eigenschaften als beispielsweise Büroanwendungen; dies muss bei ihrer Entwicklung unbedingt beachtet werden. Die aktuellen Herausforderungen bei der Entwicklung von

Embedded Systems liegen vor allem im Bereich der Software und sind weniger auf dem Gebiet der Hardware zu suchen.

Ein eingebettetes System kann in der Regel in folgende fünf logische Bestandteile zerlegt werden: Die Kontrolleinheit, die Regelstrecke, die Benutzerschnittstelle, die technische Systemumgebung sowie der menschliche Systembenutzer. Sie ergeben zusammen mit der Beschreibung, wie diese Bestandteile miteinander verbunden sind, die Grobarchitektur jedes eingebetteten Systems.

Zum Bau eingebetteter Systeme sind hardwareseitig Steuergeräte sowie eine auf die spezifische Steuerungsaufgabe angepasste Sensorik bzw. Aktuatorik erforderlich, zu deren Realisierung neben unterschiedlichen Sensoren bzw. Aktoren (auch: Aktuatoren) auch sogenannte Wandler nötig sind. Die Verbindung von Sensor und Aktor geschieht durch ein „intelligentes“ System zur Regelung eines Prozesses, das Steuergerät. Steuergeräte können in unterschiedlicher Ausprägung vorkommen: Durch komplexe Mikroelektronik können beispielsweise Sensor und Steuergerät als intelligentes System auf einem einzigen Chip integriert werden.

2 Nebenläufige Systeme

Motivation

Für den Einsatz von Nebenläufigkeit in Systemen gibt es verschiedene Argumente – auch Echtzeitanforderungen spielen dabei eine Rolle. Nebenläufige Programmverarbeitung ist in vielen unterschiedlichen Inkarnationen in Programmiersprachen vorhanden und hat vor allem für neuere Sprachen wie etwa Java eine große Bedeutung. Sie ist das Grundmodell für Multitasking und Multithreading.

In einem nebenläufigen Programm werden mehrere Kontrollflüsse erzeugt, die potentiell gleichzeitig ausgeführt werden können. Findet die Ausführung tatsächlich gleichzeitig statt, so spricht man von Parallelität (Butenhof, 1997). In deklarativen Programmiersprachen kann die Nebenläufigkeit einzelner Programmteile oftmals abgeleitet werden, z. B. durch die Konfluenzeigenschaft referentiell transparenter funktionaler Sprachen (Peyton Jones, 1989). Bei imperativen Programmiersprachen hingegen fehlt die Möglichkeit einer Nebenläufigkeit, da imperative Programme ein starres Ablaufschema aufweisen.

Zwei Prozesse sind dann nebenläufig, wenn sie unabhängig voneinander arbeiten können und es keine Rolle spielt, welcher der beiden Prozesse zuerst ausgeführt oder beendet wird. Derartige nebenläufige Prozesse können jedoch indirekt voneinander abhängig sein, da sie möglicherweise gemeinsame Ressourcen beanspruchen und untereinander Nachrichten austauschen. Dies macht eine Synchronisation an bestimmten Knotenpunkten in den Prozessen notwendig. Dieser Sachverhalt kann zu Problemen führen, die von schwerwiegenden Fehlern bis hin zu Programmstillstand und Absturz führen können. Eines der wesentlichen Ziele der Nebenläufigkeit ist eine gleichmäßige Ressourcen-Auslastung, wobei diese besonders in objektorientierten Programmiersprachen wünschenswert ist.

Kapitelübersicht

In diesem Kapitel werden wir nach einer Einführung in die Themen Multitasking, Multithreading und Prozesssynchronisiation

bzw. -kommunikation Modelle für Nebenläufigkeit kennenlernen und uns schließlich verteilten eingebetteten Systemen zuwenden.

2.1 Einführung

Einsatz von Nebenläufigkeit

Grundsätzlich werden folgende zwei Argumente für den Einsatz von Nebenläufigkeit angeführt (Butenhof, 1997):

- Viele Probleme lassen sich einfacher modellieren, wenn sie als mehr oder weniger unabhängige Aktivitäten verstanden und durch entsprechende Sprachkonstrukte umgesetzt werden können. Jede Aktivität kann dann isoliert entworfen und implementiert werden. Nebenläufigkeit wird in diesem Kontext also als Abstraktionskonstrukt verstanden, das den Softwareentwicklungsprozess vereinfacht. Ob tatsächlich eine parallele, also gleichzeitige Verarbeitung stattfindet, ist hierbei von untergeordneter Bedeutung.
- Rechner enthalten mehrere ausführende Entitäten, in der Regel einen oder mehrere Prozessoren sowie Ein-Ausgabe-Geräte. Will man diese Ressourcen gleichzeitig nutzen, so muss man ihnen entsprechend mehrere Anweisungsströme zuführen. Diese können dann parallel ausgeführt werden und damit zu einer Reduktion der Ausführungszeit führen. Das Ziel der nebenläufigen Programmdefinitionen ist hier also Parallelität zur schnelleren Programmausführung, indem man das Programm der vorhandenen Hardwarekonfiguration anpasst.

Für beide Modelle existiert eine mehr oder weniger kanonische Implementierung. Sie wirken in unterschiedlichen Systemebenen und weisen daher unterschiedliche Charakteristika bezüglich ihrer Performanz und Skalierbarkeit auf. Eine der entscheidenden Kennzahlen ist hierbei die Zahl der unterstützten nebenläufigen Aktivitäten. Diese ist im ersten Fall durch die Problemgröße definiert, während im zweiten Fall die Hardwarekonfiguration die Grenze zieht.

2.1.1 Multitasking

Multitasking ist die Fähigkeit von Software, beispielsweise Betriebssystemen, mehrere Aufgaben scheinbar gleichzeitig auszuführen. Dabei werden die verschiedenen Prozesse in so kurzen Abständen immer abwechselnd aktiviert, dass für den Beobachter der Eindruck der Gleichzeitigkeit entsteht. Man spricht daher auch oft von „Quasi-Parallelität".

Konzepte

Hierbei gibt es verschiedene Konzepte zur Handhabung des Multitasking (vgl. Kapitel 3). Das am häufigsten angewandte Konzept ist das präemptive Multitasking, bei dem der Betriebssystemkern die Abarbeitung der einzelnen Prozesse steuert und jeden Prozess nach einer bestimmten Abarbeitungszeit zu Gunsten anderer Prozesse anhält. Eine alternative Form des Multitasking ist das u. a. von älteren Windows-Versionen bekannte kooperative Multitasking. Bei dieser Form des Multitasking ist es jedem Prozess selbst überlassen, wann er die Kontrolle an den Kern zurückgibt. Dies birgt den signifikanten Nachteil, dass Programme die nicht kooperieren bzw. Fehler enthalten, das gesamte System zum Absturz bringen können. Bei Echtzeitsystemen ist das Multitasking besonders auf die geforderten Reaktionszeiten hin optimiert (siehe ebenfalls Kapitel 3).

Scheduler

Der Teil des Betriebssystems, der die Prozessumschaltung übernimmt, heißt Scheduler (siehe Abschnitt 3.3.6). Die dem Betriebssystem bekannten, aktiven Programme bestehen aus Prozessen. Ein Prozess setzt sich aus einem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Die Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Daher ist es unmöglich, dass ein Prozess den Speicherraum eines anderen Prozesses sieht. Schließlich gehören zu jedem Prozess noch Ressourcen wie z. B. geöffnete Dateien oder belegte Schnittstellen.

2.1.2 Multithreading

Die Fähigkeit mehrere Bearbeitungsstränge in einem Prozess gleichzeitig abarbeiten zu können, wird Multithreading genannt. Es ist von dem jeweilig verwendeten Betriebssystem bzw. von der Hardware abhängig, ob es sich dabei um eine reale oder nur um eine scheinbare Gleichzeitigkeit handelt. Durch Verwendung eines

Mehrprozessorsystems und eines Betriebssystems, welches den Prozessen erlaubt mehrere CPUs für verschiedene Threads gleichzeitig zu verwenden, kann eine reale Gleichzeitigkeit erreichen werden. Ein Thread (deutsch „Faden“ bzw. „Ausführungsstrang“) ist

- eine selbstständige,
- ein sequentielles Programm ausführende,
- zu anderen Threads nebenläufig arbeitende,
- von einem Betriebs- oder Laufzeitsystem zur Verfügung gestellte

Aktivität. Werden Threads vom Betriebssystem selbst unterstützt, so spricht man von „nativen Threads“. Oft möchte man nicht nur auf Betriebssystemebene nebenläufig arbeiten können, sondern auch innerhalb eines einzigen Programms. Die Programmiersprache Java unterstützt dies beispielsweise in Form von Threads, deren nebenläufige Ausführung durch die virtuelle Maschine geschieht. Innerhalb eines Prozesses kann es mehrere Threads geben, die stets alle zusammen in dem selben Adressraum ablaufen. Mehrere Threads können wie Prozesse zeitlich verzahnt ausgeführt werden, so dass sich der Eindruck von Gleichzeitigkeit ergibt. Threads besitzen im Vergleich zu Prozessen den Vorteil, vom Scheduler sehr viel schneller umgeschaltet werden zu können, d. h. der Kontextwechsel vollzieht sich rascher. Der Ablauf wird dann vom Java-Interpreter geregelt, ebenso die Synchronisation und die verzahnte Ausführung. Die Sprachdefinition von Java lässt die Art der Implementierung – also nativ oder nicht – von Threads bewusst frei.

Die Verwendung von Threads führt im technischen Sinne nicht zu einer beschleunigten Ausführung. Im Gegenteil, durch Verwaltungssaufwände bei der Erzeugung und Koordination ergibt sich im Allgemeinen sogar eine insgesamt vergrößerte Ausführungszeit, jedoch bedingt die effizientere Ressourcennutzung einerseits die bessere Auslastung der vorhandenen Hardware und andererseits entsteht für den Anwender der Eindruck einer flüssigen Verarbeitung. Erst beim Einsatz von mehreren Prozessoren in einer Maschine ergibt sich ein echter positiver Laufzeiteffekt durch die Möglichkeit, Threads auf verschiedenen CPUs zur Ausführung zu bringen.

2.1.3 Prozesssynchronisation und -kommunikation

Die nebenläufige Programmierung von Systemen muss eine Synchronisation und Kommunikation ermöglichen. Nebenläufige Prozesse stehen in der Regel in Wechselwirkung, weshalb man sie nicht an beliebige Stellen in unabhängige Ausführungspfade aufteilen kann. Die korrekte Formulierung der Prozesswechselwirkungen nennt man Prozesssynchronisation. Unter Kommunikation versteht man den Nachrichtenaustausch zwischen nebenläufigen Prozessen.

Synchronisation und Kommunikation

Das Warten eines Prozesses auf ein Ereignis, das ein anderer Prozess auslöst, ist die einfachste Form der Prozesssynchronisation oder Prozesswechselwirkung. Eine Verallgemeinerung der Prozesssynchronisation stellt die Prozesskommunikation dar, d. h. die Zusendung von Daten von einem Prozess zu einem anderen. Sie erfordert das Warten auf das Ereignis „Eintreffen der Daten“ und erfordert außerdem die Bereitstellung eines logischen Datenübertragungsweges zwischen den Prozessumgebungen. Im Zusammenhang mit Synchronisation stößt man immer wieder auf das vielzitierte Problem der „dinierenden Philosophen“ (engl. dining philosophers). Es macht deutlich, dass Prozesse geschickt synchronisiert werden müssen. Tut man dies nicht, können Verklemmungen (engl. deadlocks) die Folge sein.

Verklemmungen (Deadlocks)

Eine Menge von Threads (Prozessen) heißt verklemmt, wenn jeder Thread (Prozess) dieser Menge auf ein Ereignis im Zustand „blockiert“ wartet, das nur durch einen anderen Thread (Prozess) dieser Menge ausgelöst werden kann. Im Prinzip warten dann diese Threads (Prozesse) ewig, da ja keiner dieser Threads jemals wieder auf den Prozessor zugeordnet wird, denn jeder dieser Threads ist ja blockiert. Eine Verklemmung liegt also nur dann vor, wenn eine Menge von Prozessen bzw. Threads, die entweder um eine Ressource konkurrieren oder miteinander kommunizieren, *dauerhaft* sind.

Verklemmungsursachen

Es gibt prinzipiell zwei verschiedene Ursachen von Verklemmung. Die erste Ursache von Verklemmung liegt innerhalb des nebenläufigen Programms selbst d. h. es liegt eine fehlerhafte (irrtümliche) Programmierung vor, die sich in der Regel nur bedingt auf das Restsystem auswirkt. Die zweite Ursache einer Verklemmung kann zwischen zwei oder mehreren unabhängigen Prozessen auf eine wenig durchdachte Ressourcenverwaltung im System zurückzuführen sein, die vergleichsweise schwerwiegende Auswirkungen auf das Gesamtsystem haben kann.

Im ersten Fall kann man diese Verklemmungssituation aus Anwendersicht relativ einfach mit Hilfe eines Timeouts erkennen. Diese Verklemmungssituation ist unter Umständen sogar reproduzierbar, während im zweiten Fall eine Verklemmung sporadisch auftritt, je nachdem welche sonstigen Anwendungen, die jede für sich Ressourcen benötigt, gleichzeitig im System abgearbeitet werden.

Verklemmungsbedingungen

Folgende Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommen kann:

- Exklusivität der Ressourcennutzung
- Halten von Ressourcen beim Warten auf weitere Ressourcen
- Keine Verdrängungsmöglichkeit von Ressourcen
- Zirkuläre Wartebedingung

Maßnahmen

Jede dieser Bedingungen ist notwendig, um eine Verklemmung auszulösen. Es existieren verschiedene Strategien, um Verklemmungen entgegen zu wirken: Die Prävention, die Vermeidung von Deadlocks, das Entdecken und Beseitigen sowie – von eher theoretischer Bedeutung – das Ignorieren von Deadlocks.

2.2 Grundlegende Modelle für die Nebenläufigkeit

Anforderungen

An ein Modell für die Nebenläufigkeit werden nachstehende Anforderungen gestellt (Berry, 1998). Es muss

- einfach und intuitiv,
- kompositional,
- mathematisch fundiert, um Basis für formale Semantik und Verifikation zu bieten, und
- physikalisch sinnvoll

Modelle

sein. Drei grundlegende und vollkommen unterschiedliche Modelle bieten sich zur Realisierung dieser Anforderungen an. Sie werden durch Analogien zur Physik beschrieben (Berry, 1998):

- **Das Chemische Modell:** Die Recheneinheiten werden als eine Art Moleküle betrachtet. Die Kommunikation unter ihnen findet durch Kontakt und anschließende Reaktion statt.
- **Das Newtonsche Modell:** Die Recheneinheiten werden als Planeten angesehen. Die Planeten tauschen untereinander die Information bezüglich ihres Gewichts und Position aus und bewegen sich gemäß dieser Informationen und ihrer momentanen Geschwindigkeit und Beschleunigung. Dieser Informationsaustausch findet zu jeder Zeiteinheit statt und verbraucht keine Zeit.
- **Das Vibrationsmodell:** Die Recheneinheiten werden als Moleküle eines Kristallgitters angesehen. Wenn ein Molekül angestoßen wird, dann stößt es seine Nachbarn an, welche wiederum ihre Nachbarn anstoßen und so weiter und so fort. Dies erzeugt eine Welle, die sich mit definierter Geschwindigkeit im Kristallgitter ausbreitet.

Unterschiede

Alle drei Modelle erfüllen die oben genannten Anforderungen auf unterschiedliche Weise. Der Hauptunterschied ist dabei die erforderliche Zeit für das Aufbauen einer Kommunikation:

- *Chemisches Modell:* Immer unterschiedliche Zeit
- *Newtonsches Modell:* Keine Zeit, d. h. Null-Zeit
- *Vibrationsmodell:* Konstante Zeit

Das Chemische Modell ist nichtdeterministisch und asynchron. Seine mathematische Version bietet die Basis für die Semantik der Sprachen und des Kalküls der interaktiven Prozesse. Für reaktive Prozesse eignet es sich nicht, da die Rechtzeitigkeit nicht dargestellt werden kann.

Das Newtonsche Modell ist deterministisch und, da der Austausch der Information und ihre Verarbeitung in null Zeit stattfindet, „perfekt synchron“. Dieses Modell dient als Basis für die Definition der synchronen Sprachen und deren Semantik. Für deren Implementierung wird aber das komplexere Vibrationsmodell, wo die Information sich verzögert ausbreitet, verwendet, weil es besser der Arbeitsweise der Rechner entspricht.

Ein Beispiel, wo das Newtonsche und das Vibrationsmodell auch gemeinsam verwendet werden, ist der synchrone Schaltungsentwurf, vgl. Abbildung 2.1:

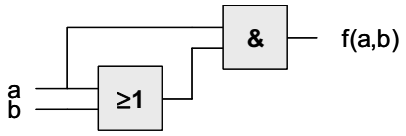


Abb. 2.1: Logische Schaltung

Hier wird angenommen, dass die Eingänge alle synchron in „0 Zeit“ ausgewertet werden (Newtonsche Modell). Aber in Wirklichkeit besteht diese Schaltung aus Gattern (logisches Und „&“ bzw. Oder „≥1“ im Beispiel in Abbildung 2.1), die eine Laufzeit besitzen (Vibrationsmodell). So liegt das Ergebnis von $(a \vee b)$ im Vergleich zu a mit der Verspätung Δ (= Gatterschaltzeit von „≥1“) an. Bei Einsatz von Schaltwerken in Schaltungen wird durch eine gemeinsame Taktung sichergestellt, dass die richtigen Signale ausgewertet werden. Dabei wird vorausgesetzt, dass der Abstand zweier konsekutiver Takt-Signale nicht kürzer ist als die Summe aller Gatterschaltzeiten Δ des längsten Pfades. Eine analoge Bedingung existiert für die Anwendung des Modells der perfekten Synchronität (siehe weiter unten).

2.3 Verteilte Systeme

Insbesondere Automobilhersteller und deren Zulieferer stehen im Bereich der Softwareentwicklung der Herausforderung einer stark steigenden Anzahl verteilter eingebetteter Spezialsysteme im Fahrzeug gegenüber. Gesucht wird daher eine Architektur-unabhängige Vorgehensweise zum Entwurf solcher Systeme, die noch keine Annahmen bezüglich der verteilten Implementierung trifft und das Ziel verfolgt, eine höhere Systemauslastung sowie eine bessere Wiederverwendung für weitere Baureihen zu gewährleisten. Ein Lösungsansatz hierfür wird in diesem Abschnitt skizziert.

Der aktuelle Trend der Elektronik-Entwicklung im Fahrzeug geht zu immer mehr Funktionalität, die auf einer ebenfalls stark wachsenden Anzahl von Steuergeräten bzw. Prozessoren implementiert wird. Um die 70 davon werden in Abhängigkeit der vom Kunden gewählten Ausstattungsvariante in aktuellen Oberklassemodellen eingesetzt. In den allermeisten Fällen führt jeder von ihnen eine ganz dedizierte Aufgabe durch. Ist die Arbeitslast des Prozessors hoch oder die Funktion stark sicherheitskritisch, macht diese Strategie durchaus Sinn.

Andererseits kommt die ausschließliche Verwendung eines Steuergeräts für eine einzige Aufgabe gerade im Bereich der

Komfortelektronik einer Verschwendung von Ressourcen gleich. Darüber hinaus wünschen sich viele Entwicklungsingenieure, dass eine einmal von ihnen erfolgreich entworfene Funktion für ein Fahrzeug auch in weiteren Baureihen mit einer ggf. ganz anderen Elektrik-/Elektronik-Architektur ebenfalls wiederverwendet werden kann – dies beschleunigt die Time-to-Market und spart Entwicklungskosten.

Partitionierung, Komponierbarkeit

Gesucht wird daher eine Architektur-unabhängige Vorgehensweise zum Entwurf solcher Systeme, die noch keine Annahmen bezüglich der verteilten Implementierung trifft und das Ziel verfolgt, eine höhere Systemauslastung sowie eine bessere Wiederverwendung für weitere Baureihen zu gewährleisten. Die hier verwendeten Schlüsselbegriffe sind die *„Partitionierung“* (Verteilung von Funktionen nach ihrer Entwicklung) und die *„Komponierbarkeit“*.

TTA, TTP

Komponierbarkeit bedeutet hier, dass Eigenschaften, die auf der Ebene der Komponenten gegeben sind, beispielsweise ein garantiertes zeitliches Verhalten, ebenfalls auf der Systemebene Gültigkeit besitzen. Eines von mehreren Beispielen hierfür ist die *Time Triggered Architecture (TTA),* eine *zeitgesteuerte* Architektur, die von der Technischen Universität Wien entwickelt wurde (Kopetz et al., 2002). Sie erlaubt mithilfe einer exakten Vorausplanung *aller* zeitlichen Aspekte des Systems dessen zeitliche Reaktionsdauer zu minimieren und Systemkomponenten so zusammenzusetzen, dass die Interaktionsmuster optimal zueinander passen. Alle Schnittstellen sind bezüglich des Zeitbereichs exakt definiert und verändern sich damit auch bei der Systemintegration nicht: Das Gesamtsystem ist komponierbar in Bezug auf das zeitliche Verhalten. Die exakte Kenntnis des zeitlichen Ablaufs im Vorhinein ist allerdings auch ein Nachteil der TTA, der zu einer geringen Flexibilität führt. Wann immer aber das Echtzeitverhalten verteilter eingebetteter Systeme optimiert werden muss, z. B. beim Drive-by-Wire, ist die TTA eine gute Wahl.

Einige Automobilhersteller haben diese Herausforderung bereits erkannt und mit entsprechenden Gegenmaßnahmen begonnen, konnten sie allerdings bisher mangels industriell verfügbarer, technologischer und methodischer Unterstützung noch nicht in die Serie umsetzen.

Auch bei den Softwarefirmen gibt es bereits vereinzelt erste Ansätze die Unterstützung bieten. So hat beispielsweise erst kürzlich ein Softwarehersteller ein CASE-Tool entwickelt, das es dem Entwickler erlaubt, Funktionen, die bereits mit Hilfe anderer Spezifikationswerkzeuge fertig entwickelt wurden, auf ein Prozessor-Netzwerk per Drag-and-Drop zu verteilen. Als Ergebnis wird ein Protokoll für die nach der Verteilung erforderliche

asynchrone Kommunikation zwischen den Prozessoren automatisch erstellt.

Dieser Ansatz ist sehr zu begrüßen, weil er den Entwickler bei der Umsetzung von Protokollinformation zur Kommunikation zwischen einem nun verteilten System entlastet. Genau dieser teilweise sehr komplexe Kommunikationsfluss sollte aber genau analysiert werden: Deadlocks müssen genauso vermieden werden wie endlose Signal-Pingpongs. Insgesamt muss Sorge getragen werden, dass das implementierte Verhalten des verteilten eingebetteten Systems hinsichtlich Funktionalität und zeitlichem Verhalten identisch mit dem ursprünglich geplanten System ist.

Grundvoraussetzung für die spätere Verteilung einer Funktion auf ein Prozessor-Netzwerk ist ein modularer Ansatz mit einer exakten Beschreibung aller Schnittstellen. Nur so können überhaupt Module, also Teilfunktionalitäten als potentielle Kandidaten zur Verteilung identifiziert werden.

Vor allem aber muss das richtige Kommunikationsprinzip für den zunächst noch rein virtuellen Austausch von Nachrichten zwischen diesen Modulen gewählt werden. Dabei sollte bedacht werden, dass die virtuelle Kommunikation ggf. zu keiner physikalischen führen muss. Dies ist immer dann der Fall, wenn beide Module auf ein und demselben Steuergerät implementiert werden.

Die asynchrone Kommunikation hat hier gleich mehrere Nachteile: Entsprechende Pufferbausteine zum Sammeln von Nachrichten an den Modulschnittstellen müssen vorgehalten werden und auch die Kommunikationsdauer bleibt gänzlich unspezifiziert, was für das Echtzeitverhalten des Systems negative Effekte haben könnte.

Perfekte Synchronie

Abhilfe schafft das sogenannte „perfekt synchrone“ Kommunikationsprinzip (siehe Kapitel 4). Es unterstellt auf der Ebene der Spezifikation zunächst, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen, ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Zeit vergeht lediglich in stabilen Systemzuständen. „Keine Zeit zu verbrauchen“ darf hier keinesfalls missinterpretiert werden; es handelt sich dabei nur um eine Abstraktion der Realität, die annimmt, dass die tatsächliche Zeitspanne, die das später realisierte System verbrauchen wird, um seinen Zustand zu verändern kürzer sein wird als die Zeit, die zwischen dem Eintreffen aufeinanderfolgender Signale der Sensorik vergeht (vgl. logischer Schaltungsentwurf, Abbildung 2.1). Diese Annahme hat mehrere Vorteile: Die Reaktionszeiten sind in der Phase des Entwurfs noch unabhängig von der konkreten, ggf. verteilten Implementierung. Künstliche, vielleicht sogar falsche, zusätzliche Verzögerungszeiten werden nicht eingeplant; und schließlich kann auf diese Weise jede Reaktion in beliebig viele

Sub-Reaktionen zerlegt werden, ohne das zeitliche Verhalten der Spezifikation zu beeinflussen.

Bei den sogenannten „synchronen Sprachen“ wie etwa Esterel, Lustre und Signal wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt. Allerdings wurden die Vorteile dieser Sprachen außerhalb der Hochschulen oder der französischen Avionik-Industrie bis dato noch nicht in ausreichendem Maße erkannt.

Darüber hinaus müssen diese Konzepte von der Ebene der Programmiersprachen auf die Ebene der Modellierungstechniken erweitert werden. Hierzu gibt es ebenfalls schon Lösungen für die Modellierung zustandsbasierter Systeme, z. B. für die an Statecharts angelehnte Techniken Argos oder μ-Charts (Scholz, 1998). Gerade letztere gibt dem Entwickler eine methodische Hilfestellung bei der Verwendung von Statecharts, um ein eingebettetes System von der ersten, gänzlich architekturabhängigen Idee bis hin zur ggf. verteilten Implementierung zu konstruieren und zu partitionieren. Es handelt sich dabei um ein rein konstruktives Verfahren: Hält der Benutzer einige vorgegebene Entwicklungsregeln streng ein, so verfeinert er das Systemverhalten nicht nur schrittweise, sondern es kann die Partitionierung der Spezifikation auf ein Prozessornetzwerk automatisch garantiert werden. Andernfalls ist eine nachgeschaltete Analyse erforderlich, die mögliche Kommunikationsprobleme ausschließt.

3 Echtzeit, Echtzeitsysteme, Echtzeitbetriebssysteme

Kapitelübersicht

Bisher haben wir für die Reaktionszeit eines eingebetteten Systems keine zeitlichen Vorgaben (engl. deadlines) gemacht. Dies gilt nicht für alle eingebetteten Systeme: Einige davon fallen in die Kategorie der Echtzeitsysteme. In diesem Kapitel werden wir Charakteristika und Herausforderungen von Echtzeit- sowie Echtzeitbetriebssystemen kennenlernen.

DIN 44300

Die *DIN 44300* (DIN = Deutsche Industrienorm) beschreibt den Begriff Echtzeit wie folgt: Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.

3.1 Echtzeitsysteme

Definition (Echtzeit)

Das Oxford Dictionary of Computing definiert Echtzeitsysteme wie folgt (freie Übersetzung):

Ein Echtzeitsystem ist ein System, bei dem der Zeitpunkt, zu dem Ausgaben erzeugt werden, bedeutend ist. Das liegt für gewöhnlich daran, dass die Eingabe mit einigen Änderungen der physikalischen Welt korrespondiert und die Ausgabe sich auf diese Änderungen beziehen muss. Die Verzögerung zwischen der Zeit der Eingabe und der Zeit der Ausgabe muss ausreichend klein für eine akzeptable „Rechtzeitigkeit“ (engl. timeliness) sein.

Echtzeitsysteme sind also Systeme, die korrekte Reaktionen innerhalb einer definierten Zeitspanne produzieren müssen. Falls die

Reaktionen diese Zeitlimits überschreiten, führt dies zu Leistungseinbußen und/oder Fehlfunktionen.

Die korrekte Funktionsweise eines Echtzeitsystems ist demnach nicht nur von der logischen Korrektheit der Ergebnisse seiner Verarbeitungsschritte abhängig, sondern auch von dem *Zeitpunkt* an dem die Ergebnisse produziert wurden. Es handelt sich selbst dann um ein inkorrektes Verhalten, wenn das Ergebnis nicht zum richtigen Zeitpunkt vorliegt und nicht nur dann, wenn das Ergebnis selbst falsch ist. Ein Fehler in der Reaktion, oder mit anderen Worten ein falscher Reaktionszeitpunkt ist daher genauso falsch wie eine falsche Reaktion: Ein richtiges Ergebnis zur falschen Zeit ist ein Fehler.

Echtzeitsysteme können in harte und weiche Echtzeitsysteme unterteilt werden:

Harte Echtzeitsysteme

Bei einem *harten Echtzeitsystem* hat das Verpassen einer Deadline katastrophale Folgen für das System. Für solche Systeme ist es zwingend erforderlich, dass Reaktionen innerhalb einer vorgegebenen Deadline erfolgen. Mit anderen Worten könnte man also sagen, die Wahrscheinlichkeit, dass dies der Fall ist, muss 100% sein. Ein Beispiel hierfür ist das Flugsteuersystem (engl. flight control system) eines Flugzeugs. Unter einer *harten Echtzeitbedingung* versteht man eine zeitliche Bedingung, die vom System stets erfüllt werden muss, da auch nur gelegentliche Verletzungen erhebliche Folgen (z. B. die Schädigung der Umgebung) nach sich ziehen würde.

Weiche Echtzeitsysteme

In *weichen Echtzeitsystemen* ist die Einhaltung von Deadlines zwar wichtig, sie funktionieren jedoch weiterhin korrekt, wenn diese Deadlines verpasst werden. Unter einer *weichen Echtzeitbedingung* versteht man eine zeitliche Bedingung, bei der gelegentliche Verletzungen tolerierbar sind. Das typische Beispiel für weiche Echtzeitsysteme sind Multimediasysteme, die für diverse Systemleistungen eine gewisse Taktrate *nach Möglichkeit* erbringen sollten, z. B. die Darstellung von 25 Bildern pro Sekunde, wobei eine gelegentliche Verletzung dieser weichen Echtzeitbedingung eine meist tolerierbare Störung darstellt, nämlich beispielsweise ein leichtes Bildflackern.

Zeitschranken

Eine weitere Unterscheidung bei den Echtzeitbedingungen ergibt sich daraus, dass Zeitschranken die Dauer einer Zeitspanne theoretisch von oben, von unten oder auch von beiden Seiten beschränken können. In der Literatur wird oft nur den *oberen Zeitschranken*, welche die maximal erlaubte Zeitspanne festlegen, Beachtung geschenkt (Fränzle, 2002). *Untere Zeitschranken*, welche Anforderungen an die minimale Dauer einer Phase zum Ausdruck bringen oder auch *beidseitige Zeitschranken* werden hingegen selten

diskutiert. Dies liegt daran, dass obere Zeitschranken eine Anforderung an die Mindestgeschwindigkeit des eingebetteten Systems oder Teilen hiervon stellen, welche im Allgemeinen wesentlich weitergehende konstruktive Maßnahmen erfordert als eine zeitliche Beschränkung von unten, die im Prinzip durch Warten erfüllt werden kann. Wenn aber untere Zeitschranken in Kombination mit oberen Zeitschranken auftreten (beidseitige Zeitschranken) muss ihnen wieder besondere Beachtung geschenkt werden.

Beispiel: Zeitschranken

Ein Beispiel für obere sowie beidseitige Zeitschranken kann in der digitalen Zündelektronik einer Motorsteuerung gefunden werden (Fränzle, 2002). Obere Zeitschranken ergeben sich hier bei der Berechnung des Drehwinkels sowie der Impulszählung der Drehzahlberechnung. Beide Berechnungen müssen in der Lage sein, alle 143 Mikrosekunden einen Impuls entgegen zu nehmen. Auch die Klopferkennung und die Zündzeitpunktberechnung müssen jeweils ihre Berechnungen innerhalb dieser Zeitspanne erledigt haben. Eine weitere, enge beidseitige Zeitschranke von nur 12 Mikrosekunden Breite ergibt sich für die Zündimpulserzeugung aus der Notwendigkeit, den Zündwinkel auf ein halbes Grad genau einzustellen.

3.2 Ereignissteuerung versus Zeitsteuerung

Eingebettete Systeme können in ereignisgesteuerte (engl. event triggered) und zeitgesteuerte (engl. time triggered) Systeme unterschieden werden, wobei der größere Teil der ersten Kategorie zuzuordnen ist.

Ereignisgesteuerte Systeme

Ereignisgesteuerte Systeme werden durch Unterbrechungen gesteuert. Liegt an einem Sensor ein Ereignis (der Umgebung) vor, so wird eine Unterbrechung (engl. interrupt) ausgelöst. Dieses Vorgehen führt zu kurzen Reaktionszeiten, ist aber bei der Bearbeitung vieler gleichzeitiger Ereignisse (sogenannte event showers) anfällig. Darüber hinaus sind die zeitlichen Abläufe im Systeme schwer planbar und manchmal auch schwer nachvollziehbar.

Zeitgesteuerte Systeme

Bei *zeitgesteuerten Systemen* erfolgt keine Reaktion auf Eingabeereignisse, sondern Unterbrechungen werden lediglich durch periodische Zeitgeber ausgelöst. Sensoren werden vom Steuergerät aktiv selbst abgefragt. Dieses Verfahren wird als Polling bezeichnet. Hierbei ist es wichtig, passende Pollingintervalle zu wählen: Kurze Signale müssen ggf. gepuffert werden, um sie nicht zu verlieren. Der Vorteil zeitgesteuerter Systeme liegt darin, dass das zeitliches Verhalten sämtlicher Systemaktivitäten bereits vor der Laufzeit vollständig planbar ist. Dies ist gerade für den Einsatz in Echtzeit-

systemen ein erheblicher Vorteil, da a priori überprüft werden kann, ob Echtzeitanforderungen eingehalten werden können.

In eingebetteten Systemen ist die Zeitsteuerung schon seit mehreren Jahren ein etabliertes Konzept. Sie sorgt durch ihren zweistufigen Entwicklungsansatz dafür, dass die Integration von Komponenten verschiedener Zulieferer auf Anhieb klappt. Das Time-Triggered Protocol (TTP) (Kopetz et al., 2002) beispielsweise kommt in Fly-by-Wire Cockpits von Honeywell zum Einsatz und wird auch von Alcatel als Feldbusprotokoll in der Bahnhofssignalsteueranlage Elektra 2 eingesetzt. TTP, FlexRay Protocol und TTCAN gehören zurzeit zu den aktuellen zeitgesteuerten Technologien die auch ein eingebettetes Betriebssystem beherrschen sollte.

3.3 Echtzeitbetriebssysteme

Wie wir bereits diskutiert haben, werden an eingebettete Systeme oft Echtzeitanforderungen gestellt. Ein Echtzeitsystem ist ein System, bei dem der Zeitpunkt, zu dem Ausgaben erzeugt werden, bedeutend ist. Programme, die auf einer (fast) beliebigen Hardware ablaufen, die Grundfunktionen von Betriebssystemen erfüllen und Echtzeitverhalten aufweisen nennt man Echtzeitbetriebssysteme.

Echtzeitbetriebssysteme erfüllen dieselben Aufgaben wie ein normales Betriebssystem wie etwa die Verwaltung von Betriebsmitteln, Prozessen, Prozessor, Speicher und Peripherie. Sie werden überwiegend in komplexen eingebetteten Systemen benötigt, da verschiedenste Aufgaben gleichzeitig abgearbeitet werden müssen. In weniger komplexen eingebetteten Systemen werden sie selten integriert, da sie dort nur unnötig Ressourcen belegen würden und zu keiner Leistungssteigerung beitragen könnten. Man findet sie in zunehmenden Maße schon in fast allen Alltagsgegenständen wie beispielsweise Drucker, Automobil (Navigationssystem, Bordcomputer, ABS-Steuerung usw.), Handy, PDA, Waschmaschine, Kühlschrank, DVD-Spieler u.v.m. Dort werden sie manchmal auch als „Firmware“ bezeichnet.

Abschnittsübersicht

In diesem Abschnitt geben wir eine Einführung in das Gebiet der Echtzeitbetriebssysteme. Zu diesem Zweck wiederholen wir zunächst den allgemeinen Aufbau sowie Aufgaben von Betriebssystemen. Bei Echtzeitbetriebssystemen nimmt die Festlegung der zeitlichen Reihenfolge (das Scheduling) von Prozessen einen besonderen Stellenwert ein. Verschiedene Schedulingalgorithmen wollen wir deshalb ebenfalls in einem Abschnitt gesondert

betrachten. Schließlich werden einige in der Praxis weit verbreitete Echtzeitbetriebssysteme kurz vorgestellt.

Wie wir gesehen haben, ist der Begriff „Echtzeit“ nicht mit einer hohen Rechenleistung der unterliegenden Hardware gleichzusetzen. Unter Echtzeit ist vielmehr die *garantierte Bearbeitung* einer Aufgabe innerhalb einer *definierten Zeitspanne* zu verstehen.

3.3.1 Aufbau und Aufgaben von Betriebssystemen

Betriebssysteme bilden die Schnittstelle zwischen Hard- und Software. Unter einem Betriebssystem (engl. Operating System kurz: OS) versteht man Software, die zusammen mit den Hardwareeigenschaften des Computers die Grundlage zum Betrieb bildet und insbesondere die Abarbeitung von Programmen steuert und überwacht. In der DIN 44300 bzw. im Informatikduden wird ein Betriebssystem wie folgt definiert:

Definitionen (Betriebssystem)

Definition (Betriebssystem) nach DIN 44300:
Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Definition (Betriebssystem) nach dem Duden Informatik:
Zusammenfassende Bezeichnung für alle Programme, die die Ausführung der Benutzerprogramme, die Verteilung der Betriebsmittel auf die einzelnen Benutzerprogramme und die Aufrechterhaltung der Betriebsart steuern und überwachen.

Ebenfalls nach der DIN 44300 umfasst ein Betriebssystem die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Aufgaben eines Betriebssystems

Ein Betriebssystem hat folgende *Aufgaben*. Es verwaltet:

- *Hardware-Betriebsmittel* (Prozessor, Speicher, Ein-Ausgabe-Geräte usw.) und Software-Betriebsmittel (Dateien, spezielle Programme)
- *Prozesse* (in Ausführung befindliche Programme)

Komponenten eines Betriebssystems

Das Betriebssystem besteht aus folgenden *Komponenten:*

- **Prozess-Verwaltung** (Kreieren und Terminieren von Applikations- und System-Prozessen, Suspendieren und Reaktivieren von Prozessen, Prozess-Synchronisation und Kommunikation, Deadlock-Behandlung)
- **Speicher-Verwaltung** ("Buchführung" über freie und belegte Hauptspeicherbereiche, Kommunikation mit den in der Hierarchie angrenzenden Speichermedien, Dynamische Allokation und Deallokation von Hauptspeicher, Speicherschutz und Zugriffskontrolle)
- **Prozessor-Verwaltung** (Dispatching, Scheduling, Unterbrechungsbehandlung)
- **Geräte-Verwaltung**
- **Datei-Verwaltung** (Datei-Konzepte, Datei-Attribute und Datei-Operationen, Zugriffs-Methoden, Verzeichnis-Strukturen und -Implementierungen, Allokations-Methoden)

Definition (Prozess)

Definition (Prozess):
Ein Prozess ist ein ablauffähiges Programm mit seinen dafür notwendigen betriebssystemseitigen Datenstrukturen wie zugeordneten Eigenschaften (z. B. Stack- und Programmzähler, Registerinhalte, Prozesszustand, sowie Eigenschaften der Speicher- und Dateiverwaltung).

Ein Prozess wird durch das Betriebssystem als eigenständige Instanz – in der Regel unabhängig und geschützt voneinander – ausgeführt. Multitaskingsysteme gestatten die nebenläufige Prozessausführung.

3.3.2 Betriebssystemarchitekturen

Universalbetriebssysteme wie etwa UNIX oder VMS besitzen meist einen relativ großen monolithischen Kern (engl. kernel), der einen umfangreichen Hauptspeicherbedarf besitzt. Neuere Betriebssysteme (z. B. Windows XP, Solaris oder Linux) hingegen benutzen entweder einen dynamischen Betriebssystemkern oder einen sehr kleinen Mikrokern (engl. microkernel). Dies birgt den Vorteil, die Ressource Hauptspeicher effizienter ausnutzen zu können. Folgende alternative Architekturen für Betriebssysteme sind möglich:

- **Monolithischer Betriebssystemkern:** Dieser Kern enthält die Treiber für alle Geräte, die evtl. an dem Rechner betrieben werden sollen und alle traditionellen Funktionen wie z. B. Prozessverwaltung, Prozessorverwaltung, Hauptspeicherverwaltung, Dateiverwaltung und Netzwerkdienste.
- **Dynamischer Betriebssystemkern:** Dieser Kern enthält nur die Funktionen, die häufig benutzt werden. Gerätetreiber oder zusätzliche Funktionen sind in separaten Modulen gekapselt, die je nach Bedarf in den Hauptspeicher geladen oder daraus entfernt werden.
- **Mikro-Betriebssystemkern:** Dieser Kern bietet nur fünf minimale Basisdienste an: Den Prozesskommunikationsmechanismus, eine einfache Speicherverwaltung, eine minimale Prozessverwaltung, ein einfaches Scheduling und eine einfache I/O-Funktionalität. Eine Aufgabe wird nur dann vom Kern erledigt, wenn die Funktionalität des Systems außerhalb des Kerns nicht gewährleistet ist. Mikro-Betriebssystemkerne sind unter gewissen Umständen weniger effizient, da häufig zwischen den Modi User und Supervisor umgeschaltet werden muss.

Anders als monolithische Betriebssystemkerne lagert ein Microkernel alle nicht unmittelbar zur Steuerung der Programmabläufe benötigten Funktionen in externe Subsysteme aus. Diese werden anschließend je nach Bedarf in den Kernel eingefügt. Auf diese Weise lässt sich das Betriebssystem für jeden Einsatzzweck anpassen und auch unter sehr restriktiven Rahmenbedingungen einsetzen.

3.3.3 Echtzeitfähige Betriebssysteme

Wie wir gesehen haben, ist der Begriff „Echtzeit“ nicht mit einer hohen Rechenleistung der unterliegenden Hardware gleichzusetzen. Unter Echtzeit ist vielmehr die *garantierte Bearbeitung* einer Aufgabe innerhalb einer *definierten Zeitspanne* zu verstehen.

Um die Echtzeitanforderungen zu erreichen, wurden früher für Echtzeitsysteme überwiegend genau auf die zu erfüllenden Aufgaben zugeschnittene und in Assembler geschriebene Programme verwendet. Diese Vorgehensweise bedingte allerdings hohe Entwicklungskosten, geringe Flexibilität und erforderte viel Spezialwissen seitens der Entwickler. Das Programm selbst musste viele typische Aufgaben, die heute ein Betriebssystem erledigt,

realisieren. Die Steigerung der Leistungsfähigkeit der Hardware bzw. der Anforderungen an die Systeme ermöglichte und erzwang immer mehr den Einsatz höherer Programmiersprachen bei der Softwareentwicklung auch für Echtzeitsysteme. Die Nutzung allgemein verfügbarer und damit kostengünstiger Komponenten für diese Systeme war eine weitere Motivation. Bei der Programmentwicklung greift man möglichst auf „Grundprogramme“ zurück, die sich durch Änderungen und Ergänzungen auf eine Anwendung zuschneiden lassen. Im Folgenden wollen wir Struktur und Eigenschaften von Programmen behandeln, die auf einer (fast) beliebigen Hardware ablaufen, die Grundfunktionen von Betriebssystemen erfüllen und Echtzeitverhalten aufweisen. Diese Programme nennt man Echtzeitbetriebssysteme.

Definition (Echtzeitbetrieb)

Definition (Echtzeitbetrieb) nach DIN 44300:
Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind und zwar derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen.

Echtzeitanforderungen

An eingebettete Systeme werden also oft Echtzeitanforderungen gestellt. Ein Echtzeitbetriebssystem führt zwischen der Hardware und den Applikationen eine Abstraktionsebene ein und ermöglicht es Softwareentwicklern auf diese Weise, von der Plattform unabhängige echtzeitfähige Applikationen zu programmieren.

Anforderungen an Echtzeitbetriebssysteme

Ein eingebettetes Betriebssystem sollte wenig Ressourcen verbrauchen und dabei stets zuverlässig und stabil laufen. Dabei dürfen Programmfehler weder das Betriebssystem noch andere Programme beeinflussen. Kurze Antwortzeiten sind erforderlich. Echtzeitsysteme und ihre Zeitanforderungen spiegeln die Größenordnung wider, unter denen Realzeitbetriebssysteme Verwendung finden. Hat man Zeitanforderungen im Minutenbereich, lässt sich ein System durchaus noch von Hand steuern, unterhalb dieser Marke (etwa eine Minute) reicht eine Automatisierung auf Basis von Mechanik aus. Zeitanforderungen im Sekunden-Bereich lassen sich mit einem Standardbetriebssystem erfüllen, im Millisekunden-Bereich benötigt man dagegen ein Realzeitbetriebssystem. Gilt es Anforderungen im Mikrosekunden-Bereich zu erfüllen, müssen die notwendigen Aktionen innerhalb einer Unterbrechungsroutine (engl. Interrupt Service Routing, kurz: ISR) durchgeführt werden. Unterhalb dieser Grenze hilft nur noch eine Realisierung in Hardware (Timmermann, 1997).

Zusammenfassend werden nach (Ber, 1996) an ein Echtzeitbetriebssystem folgende Anforderungen gestellt:

- Sie besitzen ein *deterministisches zeitliches Verhalten*, das besonders bei der zyklischen Einplanung von Prozessen notwendig ist,
- sie erlauben die Bearbeitung von vielen externen Ereignissen (mithilfe eines *Unterbrechungskonzepts*),
- die *Antwortzeiten* auf eine Unterbrechung sind definiert,
- sie gewährleisten eine *schnelle Reaktion,* d. h. einen geringen Overhead beim direkten Zugriff auf physikalische Hardware-Adressen bzw. geringer Overhead beim Kontextwechsel,
- eine *Multitaskingfähigke*it muss gegeben sein, die den Prozessor an andere laufwillige Prozesse vergibt, wenn die Prozessortätigkeit durch Interrupt, Timerablauf etc. unterbrochen wurde,
- lauffähige Programme (Tasks, Prozesse) müssen *prioritätsgesteuert* anlaufen,
- eine effiziente und *schnelle Interprozesskommunikation* muss möglich sein,
- durch ihre *Skalierbarkeit* erlauben sie ihren Einsatz in Systemen mit begrenzten Ressourcen,
- ihre *Last-Unabhängigkeit* muss garantiert werden,
- sie verwenden *Standardschnittstellen,*
- sie können auch mit *nicht echtzeitfähigen Systemen* kommunizieren,
- sie sind für unterschiedliche Prozessorarchitekturen verfügbar,
- es gibt *spezielle Entwicklungswerkzeuge,* die für Echtzeitanwendungen optimiert sind und
- ein guter und schneller Durchgriff auf den angeschlossenen *technischen Prozess* ist erforderlich.

Weist ein Betriebssystem diese Eigenschaften auf, so ist es prinzipiell für den Echtzeitbetrieb geeignet. Allerdings enthalten die oben genannten Eigenschaften noch keine tatsächlichen Zeitangaben. Antwortzeiten auf Unterbrechungen, Prozesswechselzeiten, Interprozess-Kommunikationszeiten oder gar die Angabe der Zeitäquidistanz bei zyklischer Einplanung müssen im Einzelfall genau überprüft werden.

Unterschiede zwischen Betriebssystemen und Echtzeitbetriebssystemen

Betriebssysteme in eingebetteten Systemen unterscheiden sich von Standardbetriebssystemen wie etwa Microsoft Windows, Unix, Linux usw. meist vor allem darin,

- dass sie deutlich kleiner (sowohl hinsichtlich des Codevolumens wie auch bezüglich des Funktionsumfangs) sind, um eine kosteneffiziente Integration in eingebettete Systeme zu erlauben, und
- dass sie spezielle Mechanismen zur Herbeiführung eines verlässlichen Zeitverhaltens der einzelnen Tasks enthalten (z. B. angepasste Prozessor- und Speicherverwaltung).

Aufgaben von Echtzeitbetriebssystemen

Echtzeitbetriebssysteme erfüllen grundsätzlich die gleichen *Aufgaben* wie „normale“ Betriebssysteme. Sie verwalten:

- Hardware-Betriebsmittel (Prozessor, Speicher, Schnittstellen, Ein-Ausgabe-Geräte usw.) und Software-Betriebsmittel (Dateien, spezielle Programme) und
- Prozesse (in Ausführung befindliche Programme)

und gestatten ferner den Aufruf von Systemprogrammen, die zu diesem Betriebssystem gehören oder zusätzlich bereit gestellt werden und stellen schließlich Programmierschnittstellen zur Verfügung, mit deren Hilfe eigene Programme die Funktionen des Betriebssystems nutzen können.

Zusätzliche Eigenschaften

Zusätzlich zu den allgemeinen Betriebssystemaufgaben müssen Echtzeitbetriebssysteme die Reaktionszeiten des Gesamtsystems, auf dem sie eingesetzt sind, gewährleisten. Diese werden von der verwendeten Hardware erheblich mitbestimmt.

Einschränkende Eigenschaften

Einschränkend verfügen Echtzeitbetriebssysteme nicht automatisch über eine Dateiverwaltung, da in vielen Anwendungen von Echtzeitbetriebssystemen keine mechanischen Laufwerke (Disketten, Festplatten) benötigt werden. Außerdem ist der Einsatz der Dateiverwaltung in Echtzeitbetriebssystemen problematisch, weil der Zugriff auf Datenträger zunächst selbst nichtdeterministisch ist, zum anderen aber auch durch die Interrupt Service Routine (ISR) zu Nichtdeterminismen im sonstigen System führen kann. Aus diesem Grunde ist bei Echtzeitbetriebssystemen die Speicherverwaltung ebenfalls oft einfacher aufgebaut als bei modernen Betriebssystemen. Hinzu kommt, dass viele Realzeitsysteme in Umgebungen eingesetzt werden, in denen aus Sicherheitsgründen bewegte Komponenten nicht zulässig sind. Damit scheiden ohnehin klassische Hintergrundspeichermedien (Festplatten, Diskettenlauf-

werke) aus; allerdings besteht hier technisch die Möglichkeit, sogenannte Flash-Speicher einzusetzen.

Nachstehende Tabelle gibt einen Überblick über die Unterschiede der beiden Betriebssystemarten:

Eingebettetes Betriebssystem	**Universalbetriebssystem**
Betriebssystem ist meistens in einem ROM abgelegt	Betriebssystem ist auf einer Festplatte abgelegt
Muss speichereffizient sein	Kann speichereffizient sein
Hat normalerweise keine GUI, da es meistens mit dem Embedded System interagiert	Hat überwiegend eine GUI, da es meistens mit dem Benutzer interagiert
Einarbeitungsaufwand meistens sehr hoch	Einarbeitungsaufwand eher gering
Wenige Applikationen bzw. Entwicklungswerkzeuge vorhanden	Reichliche Auswahl an Applikationen und Entwicklungswerkzeugen
Industrietauglich	Kaum industrietauglich
Geringe Lizenzkosten für den Einsatz in eingebetteten Systemen	Hohe Lizenzkosten für den Einsatz in eingebetteten Systemen
Betriebssystem ist deterministisch	Betriebssystem ist nichtdeterministisch

Tab. 3.1: Unterschiede zwischen eingebetteten und Standard-Betriebssystemen

Manche Universalbetriebssysteme bieten mittlerweile Systemerweiterungen an, die das System in die Lage versetzen, weiche Echtzeitanforderungen zu erfüllen. Bei diesen Erweiterungen handelt es sich um:

- Memory Locking
- Deaktivierung von Caches (führt allerdings gleichzeitig zu einem erheblichen Performanzverlust)
- Echtzeit Scheduling Strategien (POSIX Scheduling)

Beispiele: RTOS

Derzeit verfügbare, bekannte Echtzeitbetriebssysteme sind beispielsweise VxWorks (von Wind River), die Linux-nahen Echtzeitbetriebssysteme LynxOS (von Lynx-Works) und Linux-RT,

QNX (vom kanadischen Hersteller QNX Software Systems) und OSE (OSE Systems).

Die existierenden Echtzeitbetriebssysteme lassen sich im Wesentlichen in die zwei folgenden Kategorien unterteilen:

- Systeme, die auf optimierten Versionen von *konventionellen Betriebssystemen* aufsetzen wie z. B. Lynx und Linux-RT sowie
- Systeme, die von Grund auf *neu entwickelt* wurden, wie beispielsweise QNX, OSE und VxWorks.

3.3.4 Zeitgeber und Zugriffsebenen auf Zeit

Zur Bestimmung der Zeit gibt es in einem Echtzeitsystem sogenannte Realzeituhren bzw. Timer. Sie lassen sich prinzipiell auf zwei Arten realisieren: Sowohl in Hardware als auch in Software in Form von Vorwärts- und Rückwärtszählern. Bei Rückwärtszählern unterscheidet man repetitive Zähler von den Single Shot Zählern. Repetitive Zähler laden sich mit einem Zählwert selbständig nach Ablauf (wenn der Zählerstand 0 erreicht wird), während der Single Shot Zähler – wie bereits sein Name andeutet – explizit neu gestartet werden muss. Diese Zeitgeber erfüllen folgende Aufgaben:

Aufgaben von Zeitgebern

- **Zyklische Generierung von Unterbrechungen (Interrupts):** Ein Timer ist im System dafür verantwortlich, zyklisch (z. B. alle 10 ms) einen Interrupt zu generieren. In der zugehörigen Interrupt Service Routine (ISR) wird der Scheduler aufgerufen, es werden zeitabhängige Systemdienste (Weckaufrufe) bearbeitet und Softwaretimer realisiert. Damit hängt die Genauigkeit der zeitabhängigen Systemdienste von dieser Zeitbasis ab.
- **Zeitmessung:** Das Messen von Zeiten ist eine oft vorkommende Aufgabe in der Automatisierungstechnik. Geschwindigkeiten lassen sich beispielsweise über eine Differenzzeitmessung berechnen.
- **Watchdog (Zeitüberwachung):** Neben der Zeitmessung spielt die Zeitüberwachung eine sicherheitsrelevante Rolle in Echtzeitsystemen. Zum einen werden einfache Dienste (z. B. die Ausgabe von Daten an eine Prozessperipherie) zeitüberwacht, zum anderen aber auch das gesamte System (Watchdog). Dazu muss das System in regelmäßigen Abständen einen Rückwärtszähler zurücksetzen. Ist das System in einem undefinierten Zu-

stand und kommt nicht mehr dazu, den Zähler zurückzusetzen, zählt dieser bis 0 und bringt das System in den sicheren Zustand (löst beispielsweise an der CPU ein Reset aus).

- **Zeitsteuerung für Dienste:** Spezifische Aufgaben in einem System müssen in regelmäßigen Abständen durchgeführt werden. Zu diesen Aufgaben gehören Backups ebenso wie Aufgaben, die der Benutzer dem System überträgt (z. B. zu bestimmten Zeitpunkten Messwerte erfassen).

Dabei können nachstehende drei Zugriffsarten bzw. -ebenen auf die Zeit unterschieden werden:

Zugriffsebenen auf Zeit

(1) Hardware Level:
Auf Hardwareebene werden Timer in Form von Echtzeituhren (Absolutzeitgeber), Frequenzteilern, Rückwärtszählern und Watchdog Timern zur Verfügung gestellt.

Absolutzeitgeber halten die Absolutzeit (Tag, Monat, Jahr, Stunde, Minute, Sekunde, Millisekunde) vor. Diese Uhren sind batteriegepuffert, so dass auch nach Abschalten des Stroms die Uhrzeit weitergezählt wird. Absolutzeiten sind vor allem bei eingebetteten Systemen oftmals nicht notwendig. Insbesondere für verteilte Echtzeitsysteme jedoch ist eine genaue Absolutzeit erforderlich. Dabei ist es wichtig darauf zu achten, dass alle Teilsysteme die gleiche Zeitzone verwenden (zu einem Zeitpunkt ist die Uhrzeit unterschiedlich an den verschiedenen Orten der Erde). Um hier Probleme von vorneherein zu vermeiden, wird im Regelfall die Hardwareuhr auf die Zeitzone UTC (Universal Time) eingestellt. Die Betriebssysteme rechnen dann auf die lokale Zeit (mit Sommer-/Winterzeit usw.) um. Die eigentlichen Applikationen selbst jedoch holen sich die Zeit über Systemdienste (siehe unten) und sollten Zeiten nicht umrechnen müssen.

Über *Frequenzteiler* werden für die unterschiedlichen Hardwarekomponenten des Systems korrekte Frequenzen von einer Standardfrequenz abgeleitet (z. B. für die Baudrate einer seriellen Schnittstelle). Bei den Frequenzteilern handelt es sich um Dualzähler, an deren Zähleingang die Standardfrequenz gelegt ist. Über einen Multiplexer kann man einen der Ausgänge des Zählers auswählen. Dieser Ausgang liefert jetzt die entsprechend dem Ausgang zugeteilte Standardfrequenz.

Ein *Rückwärtszähler* dient zur Messung von Relativ-Zeiten (Zeitdifferenzen). Die Zähler werden mit einem Wert belegt und zählen dann mit einer Eingangsfrequenz (die über einen Frequenz-

teiler mit Multiplexer eingestellt werden kann) auf 0. Ist der Zähler abgelaufen, wird ein Interrupt ausgelöst.

(2) Kernel Level:
Im Betriebssystemkern werden Zeiten insbesondere auf Basis eines zyklischen Timer Interrupts verarbeitet. Dabei zählt das Betriebssystem diese Interrupts (in Linux sind dies die sogenannten Jiffies) und bietet eine Reihe auf diesen Timerticks basierenden zeitgesteuerten Diensten an:

- Gerätetreiber können Tasks für eine wählbare Zeitdauer in den Zustand "wartend" versetzen.
- Module können zyklisch Funktionen (Threads) aufrufen lassen.
- Module können einmalig (zu einem Relativzeitpunkt) Funktionen aufrufen lassen.

Vorsicht ist geboten, wenn innerhalb eines (Betriebssystem-) Moduls Absolutzeiten benötigt werden. Der Abstand zwischen zwei Timerticks muss nicht zwangsläufig in jedem System identisch sein, sondern kann variieren.

Weiterhin kritisch ist der Umstand, dass Zähler eine endliche Breite haben und damit nach endlicher Zeit ein Zählerüberlauf stattfindet (in Linux beispielsweise nach 417 Tagen). Dieser Zeitpunkt kann für ein System kritisch sein, wenn nicht an allen Stellen (sowohl im eigentlichen Kernel als auch in den Treibern und Modulen) auf Zählerüberlauf geprüft wird.

(3) User Level:
An der Programmierschnittstelle lässt sich in UNIX-Betriebssystemen und deren Derivaten über die Funktion „gettimeofday" die Absolutzeit bestimmen. Um einen Prozess oder einen Thread für eine definierte Zeit in den Zustand „wartend" zu versetzen, existieren die Funktionen „sleep" und „select".

An der Dienstschnittstelle stellt das Betriebssystem für periodische Aufgaben „cron" zur Verfügung. Um eine Task zu einem bestimmten (Absolut-) Zeitpunkt zu starten, gibt es das Kommando „at". Die Programme „cron" und „at" dienen zur zeitverzögerten oder regelmäßig wiederkehrenden Ausführung von Befehlen. In Echtzeitsystemen stehen oftmals noch zusätzliche Hardware Timer zur Verfügung. Diese ermöglichen eine zeitgenauere Steuerung von Prozessen, als dies über das System bzw. über Software Timer möglich ist.

Während auf der Kernelebene Relativzeiten wegen der möglichen Zählerüberläufe kritisch sind, stellen auf der Applikations- bzw. Userebene Absolutzeiten eine Herausforderung dar.

3.3.5 Prozesse

Logische Struktur von Betriebssystemen

Wie wir bereits erfahren haben, wird das Betriebssystem zur logischen Strukturierung meist in mehrere Schichten eingeteilt. Die unterste Schicht, der sogenannte Betriebssystemkern (Kernel) beinhaltet alle hardwareabhängigen Teile des Betriebssystems, insbesondere auch die Verarbeitung von Unterbrechungen (engl. interrupts). Auf diese Weise ist es möglich, das Betriebssystem leicht an unterschiedliche Rechner mit unterschiedlichen Ressourcenausstattungen anzupassen. Die nächste Schicht enthält die grundlegenden *I/O-Dienste* für Plattenspeicher und Peripheriegeräte. Die darauffolgende Schicht behandelt *Kommunikations- und Netzwerkdienste, Dateien und Dateisysteme.* Weitere Schichten können je nach Anforderung folgen.

Nebenläufigkeit

Die meisten Applikationen im Bereich eingebetteter Systeme erfordern ein hohes Maß an nebenläufiger Verarbeitung sowie kurze Reaktionszeiten auf externe Ereignisse aufgrund harter Echtzeitanforderungen. Um die Entwicklungszeiten solcher Echtzeitsysteme so kurz wie möglich zu halten, werden hier Multitasking Betriebssysteme bzw. sogenannte Real Time Kernels eingesetzt. Ein Echtzeitbetriebssystem muss vom Entwickler in der Regel lediglich auf die entsprechende Zielhardware angepasst werden. Es ermöglicht ein flexibles Aufteilen von Systemressourcen (CPU, Speicher, etc.) auf einzelne Tasks bzw. Prozesse.

Prozesse, Tasks

Wie bereits festgestellt, verwalten Betriebssysteme Hardware-Ressourcen sowie Prozesse. Insbesondere das zeitgenaue Management von Prozessen erfordert bei Echtzeitbetriebssystemen eine besonders genaue Beachtung. Ein *Prozess* (auch *Task,* engl. für Aufgabe) ist ein ausführbares Programm plus die dazu gehörenden aktuellen Werte des Programm Counters (PC), der Register, Variablen und sonstigen Betriebsmittel (Speicher, I/O, Signale, usw.). Für die Implementierung eines Tasks hält das Betriebssystem eine eigene Tabelle bereit, den sogenannten *Task Control Block (TCB)* oder *Process Control Block (PCB).* Der PCB enthält die Prozessinformation sowie den CPU- und Systemkontext für den jeweiligen Task. Die Prozessinformation wiederum definiert den Prozesszustand (Beispiele: erzeugt, lauffähig, laufend, suspendiert,

terminiert) und die *Prozesspriorität* (Beispiele: hoch, mittel, niedrig).

Prozesszustände

Mögliche *Prozesszustände* sind im Wesentlichen (hier muss ergänzend hinzugefügt werden, dass Anzahl und Bennennung der Prozesszustände im Allgemeinen abhängig vom konkreten Betriebssystem sind):

- **Generiert:** Die Task wurde erzeugt.
- **Bereit** bzw. **lauffähig:** Eine Task ist bereit, wenn sie alle außer der CPU angeforderten Betriebsmittel zugeteilt bekommen hat.
- **Aktiv:** Eine Task ist aktiv, wenn sie die CPU gerade benutzt.
- **Blockiert** bzw. **suspendiert:** Eine Task ist blockiert bzw. suspendiert, wenn sie ihre Operation nicht ausführen kann, weil sie auf die Zuteilung mindestens eines noch angeforderten Betriebsmittels wartet.
- **Terminiert:** Eine Task ist beendet, wenn sie alle Anweisungen abgearbeitet und die ihr zugeteilten Betriebsmittel wieder freigegeben hat.

3.3.6 Multitasking und Scheduling

In Multitasking Betriebssystemen ist ein spezieller Systemprozess erforderlich, der aus den bereiten Tasks die nächste „aktive" Task auswählt. Diesen Prozess bezeichnet man als Scheduler. Sobald mehr als eine Task den Zustand „bereit" besitzt, muss der Scheduler durch Anwendung eines Schedulingalgorithmus festlegen, welche Task die CPU erhält.

Eigenschaften von Schedulern

Ein Scheduler sollte folgende *Eigenschaften* erfüllen:

- **Fairness:** Jeder Prozess erhält einen „fairen" CPU-Anteil.
- **Effizienz:** Die CPU sollte stets maximal ausgelastet sein.
- **Antwortzeit:** Interaktive Benutzer sollten so wenig lange wie möglich warten müssen.
- **Verweilzeit:** Stapelaufträge (Batchbetrieb) sollten eine so geringe Verweilzeit im Stapel haben wie möglich.
- **Durchsatz***:* Möglichst viele Aufträge sollten in einer bestimmten Zeitspanne bearbeitet werden.

- **Terminerfüllung:** Manche Ergebnisse müssen rechtzeitig zu festgelegten Zeitpunkten zur Verfügung gestellt werden.

Bei Multitasking Betriebssystemen werden zwei grundsätzlich unterschiedliche *Arten des Schedulings* unterschieden:

Kooperatives Multitasking

(1) Kooperatives Multitasking (engl. non preemptive multitasking): Die aktuell aktive Task gibt von sich aus freiwillig die CPU zu einem für sie geeigneten Zeitpunkt frei und eine andere Task kann diese bei Bedarf nun nutzen. In diesem Fall ist nur ein geringer Verwaltungsaufwand seitens des Schedulers nötig. Es besteht jedoch die Gefahr, dass ein „unkooperativer" oder fehlerhafter Prozess alle anderen Prozesse auf unbestimmte Zeit blockieren könnte.

Verdrängendes Multitasking

(2) Verdrängendes Multitasking (engl. preemptive multitasking): Das Gegenteil des kooperativen Multitasking ist das verdrängende Multitasking. Gibt es einen lauffähigen Task mit einer höheren Priorität als die des eben laufenden Tasks, so wird diesem die CPU entzogen obwohl dieser eventuell noch nicht vollständig abgearbeitet wurde. Der Scheduler kann einem solchen Prozess die CPU entziehen (z. B. ausgelöst durch den Ablauf einer Uhr, einem sogenannten Timer Interrupt). Dadurch kann die Bearbeitung dringlicherer Tasks mit höherer Priorität jederzeit begonnen werden. Ein fehlerhafter Prozess kann das System nicht blockieren. Hier handelt es sich um die in Echtzeitbetriebssystemen bevorzugte Variante.

Der Anstoß für den Prozesswechsel durch eine Verdrängung erfolgt in Abhängigkeit von der Scheduling Strategie. Man kann im Wesentlichen die *folgenden grundlegenden Strategien* unterscheiden:

Grundlegende Scheduling Strategien

- **Zeitgesteuerte Strategien:** Jeder Prozess erhält die CPU für eine bestimmte Zeitspanne, die sogenannte Zeitscheibe (engl. time slice). Danach wird die CPU dem nächsten Prozess zugeteilt. Man spricht hier von der sogenannten Zeitscheibensteuerung bzw. Round Robin Verfahren.
- **Ereignisgesteuerte Strategien:** Ein Prozesswechsel findet dann statt, wenn ein Ereignis (z. B. ein Hardware Interrupt) einen anderen Prozess benötigt. Hier werden allgemein den einzelnen Prozessen Prioritäten zugeordnet, die sich dynamisch ändern können. Ein bestimmtes Ereignis verleiht dem gewünschten Prozess eine höhere Priorität.

Sowohl beim kooperativen Multitasking als auch bei den ereignisgesteuerten Scheduling Strategien wird die Zuteilung mit Hilfe von Taskprioritäten gesteuert. Hier spricht man beim kooperativen System von *relativem Vorrang* (der erst zum nächstmöglichen Zeitpunkt nach Freigabe der CPU durch den aktiven Prozess wirksam wird) und beim ereignisgesteuerten System *vom absoluten Vorrang* (der sofort zu einem Prozesswechsel führt). Die Zeitscheibensteuerung kann als Sonderfall der Ereignissteuerung betrachtet werden, das entsprechende Ereignis wird in diesem Fall durch den Ablauf der zugeteilten Zeitscheibe ausgelöst.

Einige bewährte *Scheduling Strategien,* die in der Praxis verwendet werden, sind:

Bewährte Scheduling Strategien aus der Praxis

- **First Come, First Served (FCFS):** Die Verteilung der Prioritäten erfolgt strikt nach Ankunftszeit, d. h. dem Zeitpunkt, wenn die Task erzeugt wird und so erstmals den Status „generiert" erhält. In dem Fall, dass zwei oder mehr Prozesse genau gleichzeitig diesen Status erhalten, wird eine zufällige Auswahl getroffen. Diese Strategie zeichnet sich durch eine gute Systemauslastung aus, hat aber ein schlechtes Antwortzeitverhalten, so dass langlaufende Prozesse (also solche, die über eine längere Zeit hinweg den Status „aktiv" einnehmen) Kurzläufer behindern. FCFS ist ein vergleichsweise einfach zu implementierendes Verfahren.
- **Zeitscheiben Verfahren (Round Robin):** Jedem Prozess wird eine feste Zeitspanne (auch: Zeitscheibe) zugeordnet. Die Zeitscheibe (engl. time slice) wird dabei so kurz gewählt, dass keine bemerkbaren Verzögerungen auftreten und es für den Benutzer den Anschein hat, ihm bzw. seinem Task „gehöre" die CPU alleine. Nach Ablauf dieser Zeitspanne wird er verdrängt und der nächste Prozess erhält die CPU. Insgesamt ergibt sich auf lange Sicht eine zyklische Zuteilung (daher auch der Namensteil „round"). Alle Prozesse haben über ihre komplette Lebenszeit hinweg eine gleichbleibende Priorität. Die Zeitscheibe kann konstant sein oder abhängig von der Prozessorbelastung variieren. Bei kleinen Zeitscheiben lassen sich vergleichsweise kurze Antwortzeiten in Verbindung mit höheren Verlusten durch die häufigen Prozesswechsel erreichen. Beispiel: Windows bis Version 3.0.
- **Prioritätssteuerung:** Jedem bereiten Prozess wird eine Priorität zugeordnet. Die Vergabe der CPU erfolgt in absteigender Priorität. Ein Prozess niedrigerer Priorität kann die CPU erst dann er-

halten, wenn alle Prozesse mit einer höheren Priorität abgearbeitet wurden. Ein Prozess mit höherer Priorität, der gerade den Zustand „bereit“ erhält, verdrängt daher stets einen aktiven Prozess niedrigerer Priorität. Alle Prozesse gleicher Priorität werden im Allgemeinen in jeweils einer eigenen Warteschlange zwischengespeichert. Zur Prioritätssteuerung gibt es verschiedene, teilweise gemischte Detailverfahren, so z. B.:

Spezielle Prioritätssteuerungen

- *Reine Prioritätssteuerung:* Alle Prozesse gleicher Priorität werden nach deren Eingangsreihenfolge abgearbeitet. Hier handelt es sich um einen oft in Echtzeitbetriebssystemen verwendeten Ansatz.
- *Prioritätssteuerung mit unterlegtem Zeitscheiben Verfahren:* Sämtliche Prozesse gleicher Priorität werden nach dem Zeitscheiben Verfahren abgearbeitet. Beispiel: UNIX.
- *Dynamische Prioritätsvergabe:* Die Priorität der auf die Zuteilung der CPU wartenden Prozesse wird mit zunehmender Wartezeit allmählich erhöht.
- *Mehrstufiges Herabsetzen (Multilevel Feedback):* Eine maximale Zuteilungszeit der CPU wird für alle Tasks einer Prioritätsstufe festgelegt. Dies geschieht für alle Prioritätsstufen getrennt. Hat ein Prozess die maximale Rechenzeit seiner Priorität verbraucht, wird ihm sodann die nächstniedrigere Priorität zugewiesen, bis er ggf. die niedrigste Prioritätsstufe erreicht.

Darüber hinaus gibt es noch zahlreiche weitere Scheduling Strategien, die in diesem Rahmen nicht aufgeführt werden können (Tanenbaum, 2001). In Dialogsystemen (also beispielsweise in Betriebssystemen in PCs) wird normalerweise das Round Robin Verfahren verwendet, um allen Benutzern akzeptable Antwortzeiten zu bieten. Bei Stapelbetrieb und gemischten Systemen kommen oft Kombinationen der vorgenannten Strategien vor, z. B. getrenntes Scheduling für Dialog- und Batch Betrieb.

3.3.7 Scheduling in Echtzeitbetriebssystemen

Echtzeitbetriebssysteme sind durchwegs Multitasking-fähige Betriebssysteme. Alle Tasks können unabhängig voneinander gestartet und ausgeführt werden. Sie werden durch Unterbrechungen (engl. interrupts), Zeitabläufe (engl. timeouts) und Eingabeereignisse

(engl. input events) gesteuert. Diese Tasks müssen miteinander kommunizieren und sich koordinieren, um gemeinsame Aufgaben lösen zu können. Der Scheduler ist die zentrale Instanz eines Echtzeitbetriebssystems. Er entscheidet, wann eine Task den Prozessor zugesprochen bekommt und wann sie ihn wieder abgeben muss. Hierzu gibt es verschiedene Schedulingstrategien, die unterschiedlich gut für Echtzeitbetriebssysteme geeignet sind.

In marktüblichen Mehrbenutzerbetriebssystemen sind die im vorangegangenen Abschnitt skizzierten Scheduling Strategien FCFS oder Round Robin ausreichend. Dies ist für Echtzeitbetriebssysteme jedoch nicht mehr der Fall. Hier sind besondere Strategien erforderlich, welche den Echtzeitanforderungen besonders Rechnung tragen.

EDF-Verfahren

Ein erster Ansatz führt uns zum *Earliest Deadline First Scheduling-Algorithmus* (kurz EDF). Beim EDF wird derjenige Task mit der nächstliegenden Deadline ausgeführt. EDF ist optimal in dem Sinne, dass er für jedes schedulebare Prozesssystem einen zulässigen Schedule konstruiert (Fränzle, 2002). Dies kann durch einen formalen Beweis belegt werden.

Trotz dieser Optimalitätsergebnisse wird EDF in der Praxis selten eingesetzt, da es deutliche *Nachteile* gegenüber anderen Verfahren hat (Fränzle, 2002): Falls sporadische (also nicht vor der Laufzeit bekannte) Tasks im System enthalten sind, können mit EDF festgelegte Schedules nicht vorab berechnet werden. Stattdessen wird eine dynamische Sortierung der Tasks während der Laufzeit erforderlich, die vergleichsweise aufwändig ist und damit die Annahme unrealistisch macht, dass Kontextwechselzeiten und Scheduling Overhead vernachlässigbar sind. EDF neigt darüber hinaus zu überflüssigen Kontextwechseln, da es einen laufenden Prozess auch dann sofort zu Gunsten eines Prozesses mit kürzerer Deadline unterbricht, wenn die weitere Bearbeitung dieses Prozesses problemlos möglich gewesen wäre. Diese Vorgehensweise erzeugt unnötigen Overhead und wirkt sich somit negativ auf die Laufzeit aus.

Prioritätensteuerung

Um das Problem der dynamischen Prozessumordnung zu verringern, verwendet man in der Regel durch *Prioritäten* gesteuerte Scheduler mit fest vergebenen Prioritäten (siehe auch Abschnitt 3.3.6). In einem solchem System ist jedem Prozess eine Priorität fest zugeordnet und zu jedem Zeitpunkt wird der gerade lauffähige Prozess mit der höchsten Priorität ausgeführt. Manche Tasks und Ereignisse sind wichtiger als andere (haben also eine höhere Priorität) und sind daher vorrangig zu behandeln. Um dieser Anforderung Rechnung zu

tragen, arbeiten Echtzeitbetriebssysteme deshalb mit einem anderen Verfahren, der sogenannten *Prioritätensteuerung*.

Der Programmierer gibt jeder Task eine feste d. h. statische Priorität, und der Scheduler wählt jeweils die lauffähige Task mit der höchsten Priorität für die CPU-Zuteilung aus. Auf diese Weise werden hoch priorisierte Aufgaben vorrangig erledigt und die weniger wichtigen unterbrochen. Nur wenn mehrere Tasks mit gleicher Priorität vorliegen, greift der Scheduler wieder auf ein anderes Verfahren, beispielsweise Round Robin zurück.

Um eine statische Festlegung (also vor der Laufzeit) der Priorität treffen zu können, müssen alle Prozessparameter vorab bekannt sein. Der Vorteil dieses Konzepts liegt in der Ausführungsgeschwindigkeit: Zur Laufzeit ist kein Planungsaufwand erforderlich. Das dynamische Scheduling ist dagegen zwar flexibler, weil alle Planungsentscheidungen erst zur Laufzeit getroffen werden müssen, benötigt aber mehr Planungszeit und ist daher für Echtzeitbetriebssysteme nicht geeignet.

RMS

Die resultierende Problematik besteht darin, eine sachgerechte Prioritätenzuordnung zu finden (Fränzle, 2002). Ein bewährter Algorithmus hierfür ist das *ratenmonotone Scheduling (RMS):*

Die *Rate* eines periodischen Prozesses ist der Kehrwert seiner Periode. Bei sporadischen Prozessen bezieht man sich ersatzweise auf die Maximalrate, die sich als reziproker Wert des Minimalabstandes ergibt. Ein ratenmonotoner Schedule entsteht, wenn in einem präemptiven Multitaskingsystem stets der lauffähige Prozess mit der höchsten Priorität ausgeführt wird, wobei die Priorisierung zwischen den Prozessen der Ratenordnung entspricht. Falls der ratenmonotone Schedule eines Prozesssystems für jede bezüglich der Prozessparameter mögliche Aktivierungssequenz zulässig ist, bezeichnen wir das Prozesssystem als ratenmonoton schedulebar, kurz RMS-schedulebar. Der Scheduler kann immer effektiv prüfen, ob ein Prozesssystem RMS-schedulebar ist. Ein Nachweis ist durch formalen Beweis möglich.

3.3.8 Speicherverwaltung

Im Multitaskingbetrieb brauchen alle verwalteten Tasks einen zugeteilten Speicher, um lauffähig zu sein. Zusätzlich zur Prozessverwaltung (Scheduling) ist also eine Speicherverwaltung erforderlich. Wie sich der Leser vielleicht noch erinnern mag, kommt hier in der Regel eine Speicherhierarchie (Register, Cache, Hauptspeicher und Festplatte) zum Einsatz. Auch im Falle des

Speichermanagements können Standardverfahren anderer Betriebssysteme nicht ohne weiteres übernommen werden.

Paging, Swapping

Die meisten Betriebssysteme ermöglichen es Applikationen, mehr (virtuellen) Speicher zu verwenden als tatsächlich physikalisch vorhanden ist. Die hierzu verwendeten Techniken heißen Paging und Swapping. Beim *Paging* werden im Bedarfsfall Teile (Blöcke) des Hauptspeichers, Seiten (engl. pages) genannt, auf eine Festplatte transferiert und bei Zugriff einer Applikation wieder zurück geschrieben. Das Ein- und Auslagern ganzer Prozesse bezeichnet man als *Swapping*. Dieses Konzept ist heute allerdings nur noch selten in Benutzung. Aktuelle Betriebssysteme verwenden das Paging. Beiden ist allerdings gemein, dass ihr Einsatz zu einem nichtdeterministischen Systemverhalten führen kann.

Sollen Echtzeitanforderungen erfüllt werden, so kann es ggf. zu lange dauern, wenn erst zur Laufzeit Programmcodes oder Daten von der Festplatte in den Hauptspeicher nachgeladen werden müssen. Dynamisches Nachladen von Speicherbereichen aus einem virtuellen Speicher, wie es z. B. bei Unix oder Linux üblich ist, kann daher im Allgemeinen zur Realisierung von Echtzeitaufgaben nicht verwendet werden. Echtzeitbetriebssysteme, die tatsächlich Swapping oder Paging anbieten, sehen dies nur für zeitunkritische Tasks vor.

In einem Echtzeitbetriebssystem müssen alle (echt-) zeitkritischen Tasks im Hauptspeicher gehalten werden. Somit kommt der Speicherverwaltung eine sehr wichtige Rolle zu. Dabei muss bedacht werden, dass jederzeit ein Task mit einer Speicheranforderung oder Speicherfreigabe an die Speicherverwaltung herantreten kann, dass rekursive Strukturen mit lokalen Datenbereichen möglich und auch üblich sind, dass gleiche Tasks auch mehrmals parallel laufen können (ggf. mit Code-Sharing), dass sogar während der Allokation von Speicher eine höher priorisierte Task „dazwischenfunken“ kann und vieles mehr. Quasi beiläufig muss das Speichermanagement auch noch Lücken im Speicher optimal zusammenfassen und nicht mehr benötigte Daten entfernen (Carbage Collection, vgl. Java). Der Speichermanager bestimmt nicht zuletzt die Reaktionszeit auf ein Ereignis. Bei Echtzeitbetriebssystemen wird die Reaktionszeit umso länger, je mehr Tasks sich gleichzeitig im Hauptspeicher befinden, weil der Speichermanager bei der Allokation mehr Zeit benötigt, um freien Speicher zu finden (auch hier gibt es natürlich Allokationsstrategien, um die Suchzeit so gering wie möglich zu halten).

Caching

Moderne Prozessoren haben Caches eingebaut, durch die die Performanz des Systems erheblich gesteigert wird. Caches induzieren allerdings ein nicht.deterministisches Verhalten. Es lässt

sich nicht vorhersagen, ob der nächste Programmbefehl oder das nächste benötigte Datum sich im Cache befindet oder nicht. Der Entwickler aber muss wissen, dass andererseits das Abschalten von Caches in der Regel zu teilweise erheblichen Performanzverlusten führen kann.

3.4 VxWorks als Beispiel eines Echtzeitbetriebssystems

Im folgenden wollen wir nun das Echtzeitbetriebssystem VxWorks (Windriver, 1993) der Firma Wind River Systems Inc. etwas genauer betrachten. VxWorks wurde unter anderem bei der US-amerikanischen Pathfinder-Mission 1996 zum Mars eingesetzt und wird derzeit in der Formel 1 als Betriebssystem in den Rennfahrzeugen verwendet, um mit Hilfe von Anwendungen die Kommunikation und Interaktion zwischen Fahrer und Rennstall zu realisieren (Quelle: www.about-it.de).

VxWorks ist ein speziell für Steuerungs- und Datenerfassungszwecke entwickeltes Echtzeitbetriebssystem. Anders als bei Windows NT/2000/XP und UNIX-Derivaten, gibt es bei VxWorks keinen Benutzerbereich, d. h. einen Bereich, der deutlich vom System abgegrenzt ist und nur begrenzte direkte Zugriffsrechte auf Systemressourcen erlaubt. Erweiterungen, um mit Hardware zu kommunizieren, werden direkt in den Kernel geladen. Direkte Hardware-Zugriffe stellen somit in VxWorks kein Problem dar (Breuer et al., 2002).

VxWorks ist aus verschiedenen Komponenten aufgebaut; die *drei Hauptkomponenten* sind ein skalierbares *Laufzeitsystem* für ein Zielsystem (siehe Abbildung 3.1), ein *Netzwerkmodul für verteilte Systeme* (mit einer TCP/IP- und Ethernet-Unterstützung) sowie der *Entwicklungsumgebung* Tornado. VxWorks besteht aus einem Kernel, der in seinem Aufbau und seiner Funktionalität einem UNIX Kernel sehr ähnlich ist sowie zusätzlichen Funktionen bzw. Bibliotheken. Dienste befinden sich auf der Taskebene statt im Kernel. VxWorks kann seit 1995 parallel zu Windows auf PCs installiert werden. Dies bietet die komfortable Möglichkeit, Windows für die Bedienoberfläche und Auswertung zu verwenden, während VxWorks die Steuerung und Regelung von Systemen übernimmt.

VxWorks besitzt einen Hochleistungs-Mikrobetriebssystemkern. Dieser Microkernel unterstützt eine Vielzahl von Echtzeitfunktionen einschließlich schnellem Multitasking, Interrupt Support, Pre-

emptive und Round Robin Scheduling. Der besondere Aufbau des Microkernels macht es möglich, die Auslastung des Systems so gering wie möglich zu halten und gleichzeitig auf äußere Ereignisse schnell und deterministisch zu reagieren. VxWorks ist sehr flexibel. Es kann sowohl in einer minimalen Konfiguration eines Kernels mit nur wenigen Modulen für eingebettete Systeme als auch in einer voll ausgebauten Variante, welche viele Benutzer bedient, betrieben werden. Die hierfür tatsächlich benötigten Ressourcen sind zu installieren. Entwickler können VxWorks sehr einfach an die Bedürfnisse ihrer eigenen Applikationen anpassen. Weiterhin besitzt VxWorks einen effektiven Intertask Mechanismus der es unabhängigen Tasks erlaubt, ihre Aktionen innerhalb des Echtzeitsystems selber zu koordinieren.

Auf Grund dieser Eigenschaften und weiterer wie etwa Multiprozessorfähigkeit, Shared Memory, Unterstützung des VME-Bus, Message Queues etc. ist VxWorks besonders gut für den Einsatz im Bereich eingebetteter Systeme geeignet. Ein besonderer Vorteil dieses Betriebssystems ist die Möglichkeit, kompilierte Objektdateien beim Laden dynamisch zu linken. Dies bedeutet, dass zur Laufzeit eine Objektdatei geladen werden kann und alle darin enthaltenen globalen Variablen und Symbole stehen danach betriebssystemweit zur Verfügung. Ein weiterer entscheidender Vorteil von VxWorks ist dessen Skalierbarkeit bzw. Konfigurierbarkeit. Dies macht das Betriebssystem für den universellen Einsatz geeignet. So kann VxWorks sowohl in Applikationen zum Einsatz kommen, die zur Laufzeit nur einen kleinen effizienten Betriebssystem Kernel benötigen, als auch bei großen verteilten Systemen, die UNIX-kompatible Schnittstellen benötigen.

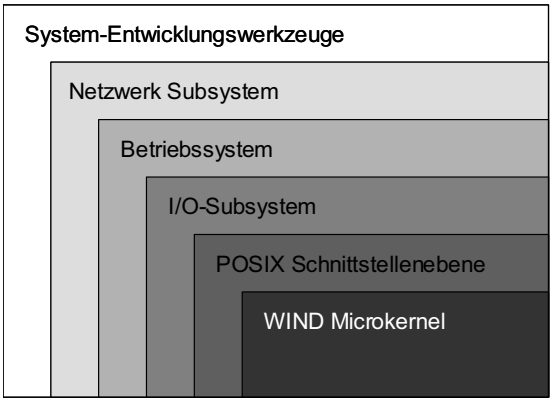


Abb. 3.1: Das skalierbare Laufzeit-System von VxWorks (aus (Berthold, 1996))

3.4.1
Das Laufzeitsystem

Die Basis des VxWorks Betriebssystems ist der sogenannte „Wind"-Microkernel (siehe Abbildung 3.1). Durch diesen Kernel werden eine Multitasking-Umgebung sowie Mechanismen zur Interprozess-Kommunikation und Synchronisation zur Verfügung gestellt. Alle anderen Betriebssystemaufgaben werden auf die Ebene der Tasks verlagert und auf die einfache Kernel-Basis abgebildet. Eine wichtige Aufgabe des Kernels besteht in der Bereitstellung einer Umgebung zur Verwaltung nebenläufiger Tasks. Der Echtzeit-Kernel ist für die deterministische Zuteilung von Rechenzeit auf der CPU für die verschiedenen Tasks zuständig. Dies wird im Gegensatz zu Unix-Systemen mit einem prioritätsbasierten, unterbrechenden Multitasking realisiert. Insgesamt stehen 256 unterschiedliche Prioritätsstufen zur Verfügung. Jeder Stufe können prinzipiell beliebig viele Tasks zugeteilt werden. Zur Synchronisierung und Kommunikation zwischen den verschiedenen Tasks wird das Konzept der (binären sowie zählenden) Semaphore und Message Queues verwendet. Um Plattform-Unabhängigkeit und Portabilität zu gewährleisten, ist der Kernel über eine Schnittstelle nach POSIX 1003.1b (vgl. www.posix.com) zugänglich gemacht.

3.4.2
Exkurs: Der POSIX Standard

POSIX, das „Portable Operating System Interface", ist eine Spezifikation, die vom IEEE erarbeitet und von ANSI und ISO standardisiert wurde. Aufgabe des POSIX Standards ist es, die Portabilität von Applikationen auf Quellcodeebene zu gewährleisten. Im Rahmen von POSIX werden eindeutige Schnittstellen in Form von Funktionen definiert, die ein Betriebssystem bereitstellen muss, um diesem Standard zu genügen. Ein POSIX konformer Quellcode ist daher auf allen POSIX Betriebssystemen kompilierbar. Beim *POSIX Substandard IEEE 1003.1b* handelt es sich um echtzeit-spezifische Erweiterungen sowie I/O-Erweiterungen. POSIX 1003.1b erlaubt die Verwendung eines Betriebssystems in *weichen* Echtzeit-Situationen. Ferner können als Scheduling Strategien Round Robin sowie ein statisches Prioritätenverfahren verwendet werden.

3.4.3 Das I/O-Subsystem von VxWorks

Für eingebettete Systeme ist der Zugriff auf Hardwarekomponenten wesentlich. In VxWorks kann Speicher direkt adressiert werden (engl. Direct Memory Access, kurz: DMA). Somit kann auch unmittelbar auf Hardware zugegriffen werden, was die Verwaltung selbst sehr zeitkritischer Hardwarekomponenten ermöglicht. Darüber hinaus stellt VxWorks ein komplettes Unix-kompatibles I/O-System zur Verfügung. Treiber können so nach bekannten Mechanismen in das System – sogar dynamisch während der Laufzeit – eingebunden werden. Ferner können weitere Unix-Treiber leicht nach VxWorks portiert werden.

3.4.4 Unterstützung verteilter Systeme in VxWorks

Eine stets anwachsende Zahl eingebetteter Systeme ist Teil komplexer verteilter Systeme, bei denen die Kommunikation untereinander eine entscheidende, weil zeitkritische Rolle spielt. Klassische gängige Kommunikationsprotokolle erfüllen jedoch keine Echtzeitanforderungen. VxWorks stellt echtzeitfähige, aber auf Standard-Unix basierende Kommunikationsprotokolle zur Verfügung.

3.4.5 VxWorks Entwicklungswerkzeuge

Um echtzeitfähige Anwendungen entwickeln zu können, müssen dem Entwickler passende Entwicklungswerkzeuge und -konzepte zur Verfügung gestellt werden. Das Cross-Entwicklungspaket Tornado für VxWorks ist auf die speziellen Anforderungen der Softwareentwicklung im Bereich der Echtzeitsysteme maßgeschneidert. Die Programmentwicklung (Spezifikation, Codierung, Kompilierung und Debugging) erfolgt auf einem UNIX-Rechner. Abbildung 3.2 verschafft einen Überblick.
Um Portabilität eines Programms auf verschiedensten eingebetteten Systemen zu gewährleisten, sollte stets der gleiche Compiler verwendet werden. Im Falle von VxWorks kommt hier der GNU-C (vgl. www.gnu.org) bzw. C^{++}-Compiler zum Einsatz. Der Cross-Debugger GNU-GDB ermöglicht dann ein Source Level Debugging. VxWorks bietet ferner über eine Shell (eine Shell ist eine flexible,

Kommandozeilen-basierte Benutzerschnittstelle des Betriebssystems, welche vorzugsweise zum Start von Prozessen verwendet wird) auf dem Entwicklungsrechner einen interaktiven Zugriff auf den Zielrechner. Auf diese Weise können alle C-Funktionen, die der Anwender mit seinen Anwendungsmodulen auf den Zielrechner überträgt, aufgerufen und getestet werden. Darüber hinaus besteht die Möglichkeit, mit Hilfe der Shell Betriebssystemroutinen aufzurufen und auszuführen. VxWorks bietet weiterhin die Möglichkeit, interaktiv Softwaremodule auf das Zielsystem schrittweise nachzuladen. Ein neues Modul muss lediglich neu kompiliert, nicht aber neu gelinkt werden. Offene Referenzen werden dabei über Symboltabellen auf dem Ziel- und dem Hostsystem aufgelöst. Mit einem System-Browser erfolgt eine graphische Darstellung des resultierenden Echtzeitsystems. Dabei lassen sich Speicherbereiche oder Informationen über Systemobjekte (Tasks, Semaphore, Warteschlangen) abfragen und darstellen.

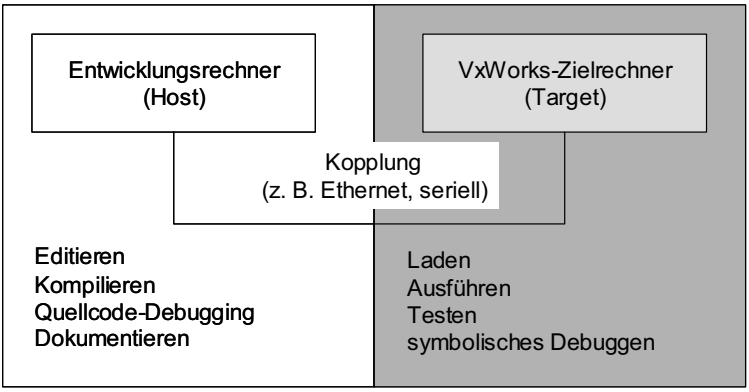


Abb. 3.2: Das Konzept der Cross-Entwicklung in VxWorks (aus (Berthold, 1996))

Programmierer können bei der Entwicklung ihrer Programme mehrere Methoden des Betriebssystems nutzen. Unter anderem gehören Shared Memory, Message Queues, Semaphoren, Events und Pipes (für die Intertask Kommunikation innerhalb einer CPU), Sockets und Remote Methoden dazu. Außerdem steht dem Benutzer eine Vielzahl von Industriestandards zur Verfügung wie z. B. POSIX, TCP/IP, UDP, ARP, RIP v1/v2, SLIP, CSLIP, BOOTP, DNS, DHCP, TFTP, NFS, SUN RPC, FTP, rlogin, rsh, telnet, SNTP usw. Als Programmiersprache steht neben C und C++ auch Java zur Verfügung.

Ständig aktualisierte Informationen über VxWorks finden sich im Internet auf den Seiten der Firma WindRiver Systems Inc. (www.wrs.com).

3.5 Weitere Beispiele eingebetteter Betriebssysteme

Derzeit gibt es über 100 verschiedene eingebettete Betriebssysteme; aber nicht jedes von ihnen ist für einen bestimmten Mikroprozessor geeignet, da hierzu Betriebssystem-spezifisch einige Grundvoraussetzungen erfüllt sein müssen. So benötigt z. B. das Betriebssystem Windows CE 5.0 eine 32 Bit CPU (ARM, SHx, MIPS oder x86), 250 KByte Flash und mindestens 6 bis 8 MByte RAM. Hat man sich mal für einen bestimmten Mikroprozessor entschieden, kann man meist immer noch zwischen mehr als einem Dutzend Betriebssystemen auswählen.

Mobile Systeme, mobile Betriebssysteme

Die nachfolgende Auswahl von eingebetteten Betriebssystemen erhebt folglich keinen Anspruch auf Vollständigkeit. An dieser Stelle sei darauf hingewiesen, dass es sich bei vielen dieser eingebetteten Betriebssysteme um keine Echtzeitbetriebssysteme im engeren Sinne, sondern um Betriebssysteme für mobile Rechnersysteme handelt. Unter dem Terminus *„mobile (Rechner-)Systeme“* wollen wir hier nur solche Systeme zusammenfassen, die von ihrem Benutzer mitgeführt und unabhängig von anderen Systemen verwendet werden (z. B. Mobiltelefone, PDAs, Handhelds, Smartphones, Wearables usw.). Sie alle teilen die Eigenschaft, nur weichen, aber keinen harten Echtzeitanforderungen genügen zu müssen. Folglich müssen die meisten mobilen Betriebssysteme ebenfalls keine harten Echtzeitanforderungen erfüllen. Unter einem *mobilen Betriebssystem* wird im Folgenden ein Programm für die Verwaltung der Ressourcen eines mobilen Rechnersystems verstanden.

Nichtsdestoweniger müssen auch mobile Betriebssysteme teilweise restriktiven Anforderungen an ihre Leistungsfähigkeit genügen. So unterliegen mobile Systeme oft Beschränkungen wie leistungsschwachen Prozessoren, geringen Speicherressourcen, eingeschränkten Displaygrößen und Eingabemöglichkeiten sowie einer begrenzten Stromversorgung. Durch diese Restriktionen ist der Einsatz von UniversalBetriebssystemen selten möglich. Darüber hinaus werden an mobile Betriebssysteme Anforderungen gestellt, die sich an dem Mobilitätsgrad und dem gewählten Endgerätetyp orientieren (Schiefer, 2004), (Devooght, 2003):

- Ausfallsicherheit
- Schnelle Betriebsbereitschaft (kurze Ladezeiten)
- Ergonomie (Benutzerfreundlichkeit, intuitive Bedienbarkeit)
- Einhaltung weicher Echtzeitanforderungen

- Erweiterbarkeit und Anpassbarkeit für neue Funktionen und Kokmmunikationskanäle
- Hohe Inormationssicherheit
- Geringer Energieverbrauch durch eine ausgefeilte Energieversorgung (engl. power management), z. B. das Abschalten nicht verwendeter Systemkomponenten.

In diesem Abschnitt sollen einige weit verbreitete Betriebssysteme für mobile Systeme vorgestellt werden. Es wurden Symbian OS, Palm OS und Windows CE ausgewählt. Zur Gegenüberstellung mobiler Betriebssysteme und Echtzeitbetriebssysteme schließt dieser Abschnitt mit einer kurzen Betrachtung der beiden Echtzeitbetriebssysteme QNX und Embedded Linux.

3.5.1 Symbian OS

PDA

Symbian OS ist ein Betriebssystem für PDAs (Personal Digital Assistants) und Smartphones. Es stammt von Psions 32 Bit EPOC Plattform ab. Zahlreiche Kommunikationsprotokolle werden von ihr unterstützt. Schwerpunkte setzt Symbian dabei vor allem in der drahtlosen Kommunikation (WAP, Bluetooth usw.), im Netzwerkbereich und in der Telekommunikation. Symbian wurde 1998 gegründet und besteht aus folgendem Konsortium: Arima, benQ, Fujitsu, Lenovo, LG Electronics, Mitsubishi Electric, Nokia, Motorola, Panasonic, Sanyo, Sendo, Sony Ericsson, Psion und Siemens.

Robustheit

Das Symbian Betriebssystem wurde so konzipiert, dass Abstürze die einen Reboot hervorrufen, so gut wie unmöglich sind. Um dies zu erreichen, läuft jeder Prozess in einem eigenen, geschützten Adressbereich; so ist es einer Anwendung nicht möglich, den Adressbereich einer anderen Anwendung zu überschreiben. Auch der Kernel läuft analog in einem geschützten Adressbereich, so dass es für keine Anwendung möglich ist, den Stack oder den Heap des Kernels zu überschreiben und dadurch einen systemweiten Absturz zu verursachen.

Speicher

Symbian unterscheidet zwischen Arbeitsspeicher (RAM) und Festspeicher (Flash-Speicher, Erweiterungskarten). Um Arbeitsspeicher zu sparen, wird die sogenannte Execute-in-Place Methode verwendet: Anwendungen lassen sich direkt aus dem Flash-Speicher ausführen. Im Gegensatz zu vielen anderen vergleichbaren Betriebssystemen unterstützt Symbian präemptives Multitasking und

Multithreading, wodurch allerdings wiederum der Arbeitsspeicherbedarf erhöht wird.

Entwicklung

Anwendungen können in folgenden Sprachen implementiert werden: C++, Java, OPL und .NET. Hilfreiche Informationen für Entwickler u. a. finden sich unter www.symbian.com.

3.5.2 Palm OS

PDA

Das Palm OS gehört zu den am weitesten verbreiteten Vertretern von eingebetteten Betriebssystemen, da es v. a. in zahlreichen PDAs Einsatz findet. So wurden bisher rund 21 Millionen Geräte, welche auf Basis dieses Betriebssystems betrieben werden, ausgeliefert. Ferner stehen rund 13.000 Applikationen für das Palm OS zur Verfügung. Obwohl es vorzugsweise in PDAs verwendet wird, gibt es durchaus auch andere Gerätetypen, vor allem medizinische Instrumente und Smartphones, die es als Betriebssystem einsetzen.

Kernel

Das Palm OS ist wie viele andere Betriebssysteme auch um einen Kernel herum aufgebaut. Dieser Kernel stellt die Grund-funktionalität zur Verfügung. Fast alle höheren Funktionen werden dann durch einen sogenannten „Manager“ zur Verfügung gestellt. Will eine Applikation beispielsweise einen Alarm auslösen, so muss sie den entsprechenden Befehl an den Manager weiterleiten – erst dieser löst dann den Alarm aus. Dem Applikationsentwickler wird durch die Überwachungsfunktion des Managers einerseits eine hohe Abstraktionsebene geboten, andererseits bietet sie Schutz vor fehlerbehaftetem Code.

Multitasking

Verwirrung gibt es in den einschlägigen Foren oftmals hinsichtlich der Multitasking-Eigenschaften des Palm OS, darum sei hier folgendes klargestellt: Das Palm OS ist ein 32 Bit, ereignisgesteuertes Betriebssystem mit eingeschränkten Multitasking-Fähigkeiten. Auf Betriebssystemebene können nebenläufig mehrere Tasks ausgeführt werden, auf Anwendungsebene jedoch nicht.

Speicher-management

Das Palm OS benutzt den RAM als Arbeitsspeicher sowie zum permanenten Speichern von Daten und Programmen. Dabei wird der RAM in zwei Segmente unterteilt, in die sogenannte Storage Area sowie in die Dynamic Area. Die Größe der einzelnen Areas hängt vom Betriebssystem und natürlich vom verfügbaren Speicher im Gerät ab, kann also nicht eingestellt werden. Das Palm OS läßt eine Speichererweiterung durch Speicherkarten zu. Um Arbeitsspeicher zu sparen, wird wie beim Symbian OS die Execute-in-Place Methode verwendet.

3.5.3 Windows CE

Im Gegensatz zu den bis hierher vorgestellten Betriebssystemen hat man bei der Entwicklung von Microsoft Windows CE versucht, möglichst nahe an den bereits bestehenden Universalbetriebssystemem der Windows-Familie zu bleiben. Das Design ist daher zwangsläufig nicht so konsequent auf den Bereich eingebetteter Systeme zugeschnitten, wie dies bei den anderen Betriebssystemen der Fall war. So ist Windows CE beispielsweise ressourcenintensiver als VxWorks oder QNX. Andererseits findet der Benutzer eine vertraute Oberfläche vor. Windows CE besitzt eine offene Architektur und unterstützt daher eine Vielzahl von Hardwarekomponenten. Microsoft sieht das Hauptanwendungsgebiet von Windows CE im Informations-, Kommunikations- und Unterhaltungsbereich (Internet-TV, Set-Top-Boxen) (Quelle: www.microsoft.com).

Speicher-management

Speichermanagement:
Das 32 Bit Betriebssystem Windows CE verwaltet den Speicher ähnlich wie das Palm OS. Der RAM wird in zwei Teile aufgeteilt, ein Teil als Arbeitsspeicher, der andere als permanenter Speicher. Das System selber ist im ROM abgespeichert. Im Gegensatz zu den vorher vorgestellten Betriebssystemen wird bei Windows CE nur das Betriebssystem „in place“ ausgeführt. Applikationen werden in den Arbeitsspeicher kopiert und von dort ausgeführt. Ein weiterer sehr wichtiger Unterschied etwa zu Palm OS oder Symbian OS ist folgender: Daten, die im permanenten Speicher von Windows CE gehalten werden, sind komprimiert. Windows CE unterstützt eine große Anzahl von Speichererweiterungen, so dass auch größere Anwendungen bzw. Datenmengen auf einem Windows CE System abgelegt werden können.

Multitasking

Multitasking:
Windows CE arbeitet wie Windows NT, 2000 und XP mit präemptiven Multitasking. Um das System nicht zu überlasten, wird die Anzahl von gleichzeitig laufenden Applikationen auf maximal 32 limitiert. Wird diese Grenze erreicht, kann Windows CE eine Applikation schließen.

Programmieren

Programmieren unter Windows CE:
Das Studium der Entwicklungsumgebungen für Windows CE zeigt, dass Microsoft den Bereich der Programmiersprachen C/C++ und Basic mit seinen Embedded Visual Tools dominiert. Dies hat dem Anschein nach vornehmlich folgende Gründe. Zum einen sind die

Embedded Visual Tools ohne Aufpreis erhältlich, zum anderen sind sie sehr umfangreich. Sie bieten neben einem Source Code bzw. Ressource Editor sowie einer umfangreichen Sammlung von Klassen und einem detaillierten Hilfesystem auch Debugging Tools sowie einen Windows CE Emulator. Softwareentwickler, die bereits mit Visual Studio vertraut sind, sollten sich auch mit den Embedded Visual Tools schnell zurecht finden.

3.5.4 QNX

POSIX

QNX ist ein 32 Bit Multiuser System und wurde von der kanadischen Firma QNX Software Systems Ltd. entwickelt. Es kann für Echtzeit-Applikationen verwendet werden und unterstützt Multitasking und prioritätsgesteuertes, verdrängendes Scheduling mit schneller Kontextumschaltung. Durch die Orientierung am POSIX Standard fällt Benutzern, die bereits andere UNIX-Derivate, wie Linux, HP-UX, IRIX, Solaris oder SunOS kennen, die Bedienung sehr leicht. Eine flexible Architektur erlaubt sowohl den Einsatz in kleinen eingebetteten Systemen mit minimaler Konfiguration und nur wenigen Kernelmodulen, als auch in großen, im Netzwerk verteilten Anwendungen. Man muss daher nur die Ressourcen installieren, die tatsächlich benötigt werden. Entwickler können QNX auch sehr einfach an die Bedürfnisse ihrer eigenen Applikationen anpassen.

Die Effizienz, Modularität und Einfachheit erreicht QNX durch zwei fundamentale Prinzipien, seine Microkernel-Architektur und die nachrichtenbasierte Kommunikation zwischen Prozessen. QNX besteht aus einem kleinen Kern mit einer Gruppe kooperierender Prozesse (Prozess-Manager, Geräte-Manager, verschiedene Filesystem-Manager, Netzwerk-Manager, die graphische Bedienoberfläche Photon microGUI usw.). Sie ergänzen je nach Bedarf den Microkernel. Alle Module außer dem Prozess-Manager können während der Laufzeit gestartet oder beendet werden (Quelle: www.qnx.com).

Prozess-Manager

Prozess-Manager:
Der Prozess-Manager ist mit dem Microkernel in einem Modul verbunden. Dieses Modul ist für alle Laufzeitsysteme erforderlich. Der Prozess-Manager benutzt den gleichen Adressraum wie der Microkernel. Er wird vom Microkernel wie ein Prozess gehandhabt und nutzt zur Kommunikation mit anderen Prozessen im System das Message Passing (Nachrichtenaustausch) des Microkernels. Der

Prozess-Manager ist für die Erzeugung neuer Prozesse im System verantwortlich und verwaltet außerdem die grundlegenden Prozess-Ressourcen. Diese Dienste werden alle über Nachrichten angeboten. Will z. B. der Prozess-Manager einen neuen Prozess erzeugen, verschickt er die Informationen in Form einer Nachricht. Da Nachrichten netzwerkweit funktionieren, kann sie zum Erzeugen eines neuen Prozesses einfach an den Prozess-Manager des entfernten Netzwerkknotens versandt werden.

Geräte-Manager

Geräte-Manager:
Der Geräte-Manager von QNX bildet die Schnittstelle zwischen Prozessen und zeichenorientierten Terminal-Geräten. Weiterhin reguliert er den Datenfluss zwischen einer Anwendung und dem Gerätetreiber.

Dateisystem-Manager

Dateisystem-Manager:
QNX bietet eine Auswahl an Dateisystemen an, die gleichzeitig nebeneinander laufen können. Wie die meisten Dienste des Betriebssystems, werden sie außerhalb des Kernels ausgeführt. QNX unterteilt sie in fünf Kategorien: Image, Block, Flash, Network und Virtual Filesysteme.

Power-Manager, Power Management Framework

Power-Manager:
Als die zentrale Komponente innerhalb des Power Management Frameworks, ist der Power-Manager verantwortlich für den Energiemodus (engl. power mode) jeder einzelnen vom Framework verwalteten Komponente. Das Framework ermöglicht eine feingranulare Kontrolle der Leistungsaufnahme jeder einzelnen Systemkomponente. Besonders hervorzuheben ist die Eigenschaft des Frameworks, es dem Anwendungsentwickler und nicht etwa dem Betriebssystem zu erlauben, individuelle Strategien für die Leistungsaufnahme (engl. power policies) für jedes System zu definieren. Dies ermöglicht letztendlich schnellere Systemantwortzeiten genauso wie längere Batterielebensdauern.

GUI

Photon microGUI:
Viele eingebettete Systeme benötigen für die Interaktion mit der Applikation ein User Interface (UI). Für komplexe Applikationen oder für maximale Benutzerfreundlichkeit ist eine graphische Benutzeroberfläche (GUI) mit mehreren Fenstern wünschenswert. Graphische Fenstersysteme wie wir sie von Betriebssystemen im Desktop-Bereich her kennen benötigen allerdings zu viele Systemressourcen als dass sie für den Einsatz in eingebetteten Systemen praktikabel wären. Photon geht bei der Erzeugung einer GUI einen

völlig neuen Weg; es benutzt, im Gegensatz zu anderen monolithischen Fenstersystemen, einen Microkernel und eine Menge kooperierender Prozesse. Als Ergebnis liefert Photon eine GUI mit bemerkenswerten Eigenschaften (Quelle: www.qnx.com):

- Bei knappen Speicherressourcen bietet Photon einen hohen Grad an Fensterfunktionalität in Umgebungen, in denen sonst nur eine kleine Grafikbibliothek Platz hätte.
- Photon bietet eine sehr flexible, durch den Benutzer ausbaufähige und skalierbare Architektur, die es Entwicklern ermöglicht, die GUI an ihre eigenen Applikationen anzupassen.
- Durch die flexible Anbindung verschiedener Plattformen können Photon-Applikationen von jeder virtuell verbundenen Bildschirmumgebung genutzt werden.

Programmieren unter QNX

Programmieren unter QNX:
In den Softwarebibliotheken von QNX findet sich u. a. eine komplette GNU-Werkzeugkette. Mit ihrer Hilfe ist Portierung von Linux-Software auf QNX möglich. Besonders hervorzuheben ist auch der Photon Application Builder: Mit seiner Hilfe lassen sich in kurzer Zeit Applikationsprototypen erstellen.

3.5.5 Embedded Linux

POSIX, GPL

Linux ist ein Multiuser Multitasking 32 Bit und 64 Bit POSIX konformes Betriebssystem, welches hauptsächlich in Internetservern und in den letzten Jahren auch zunehmend auf Desktops eingesetzt wird. Unter Embedded Linux versteht man eine Zusammenstellung von Linux Kernel und Software, die speziell auf eingebettete Systeme zugeschnitten ist, sich grundsätzlich aber nicht von dem Linux, das auf Servern und Desktops eingesetzt wird, unterscheidet. Genau genommen bezieht sich die Bezeichnung *Linux* lediglich auf den Kernel des Betriebssystems. Werkzeuge und Software, die auf diesem Kernel aufbauen, kommen aus dem *GNU Projekt* und fallen, ebenso wie der Kernel, unter die GNU Public License (GPL). Sie sind nicht Linux spezifisch, sondern können in den meisten Fällen auch auf anderen Unix-artigen, teilweise sogar Windows Betriebssystemen eingesetzt werden.

Linux ist multitaskingfähig. Standard-Distributionen sind jedoch nicht für den Echtzeitbetrieb geeignet. Hierfür wurden insbesondere zwei spezielle Linux-Systeme implementiert, RT-Linux und RTAI

(Real Time Application Interface). Diese Systeme verfügen über einen echtzeitfähigen Kernel.

3.6 Zusammenfassung

Unter der Echtzeitfähigkeit eines Betriebssystems versteht man in erster Linie dessen reale Fähigkeit, in einer gegebenen Betriebsumgebung alle anstehenden Aufgaben und Funktionen unter allen Betriebszuständen immer rechtzeitig und ohne Ausnahme erledigen zu können. „Rechtzeitig“ oder „in Echtzeit“ versteht sich somit nicht als exakte wissenschaftliche Definition, sondern als sehr variable Größe, die sich nach den jeweiligen (Echtzeit-) Anforderungen der spezifischen Anwendungen und deren zeitlichen Rahmenbedingungen orientiert und ausrichtet. Ein Echtzeitsystem ist also ein eingebettetes System, das Echtzeitanforderungen besitzt und dann ggf. mit Hilfe eines Echtzeitbetriebssystems implementiert werden kann.

Ein Echtzeitbetriebssystem führt zwischen der Hardware und den darauf auszuführenden Applikationen eine Abstraktionsebene ein und ermöglicht es Softwareentwicklern auf diese Weise, von der Plattform unabhängige echtzeitfähige Applikationen zu programmieren, welche die geforderten Echtzeitbedingungen einzuhalten versuchen. Das zeitliche Verhalten eines Echtzeitsystems wird so vorhersagbar bzw. deterministisch. Um dies zu unterstützen, enthalten Echtzeitbetriebssysteme spezielle Scheduling Algorithmen für die Einplanung der zeitlichen Reihenfolge der in der Applikation enthaltenen Teilaufgaben. VxWorks von der Firma Windriver Systems ist ein speziell für Steuerungs- und Datenerfassungszwecke entwickeltes Echtzeitbetriebssystem. Es wird im Flugzeugbau genauso verwendet wie im Automobilbau bzw. im Motorsport. Als Open Source Echtzeitbetriebssystem steht beispielsweise eCOS zur Verfügung.

4 Programmierung eingebetteter Systeme

Motivation

Die Programmierung von Software für eingebettete Systeme in Assembler war lange Zeit möglich, da die Größe der Software meist nur einige 100 Bytes umfasste. Das rapide Anwachsen der Softwareumfänge, vor allem aufgrund der stark anwachsenden Leistungsfähigkeit neuer Mikrokontroller führt dazu, dass eine kostengünstige Programmierung in Assembler nicht länger erfolgen kann. Zudem ist die geschätzte Leistung eines Programmierers in Codezeilen pro Tag bei Anwendungssoftware ca. zehnfach höher als bei Systemsoftware. Daran wird deutlich, dass heutige Softwarewerkzeuge immer noch nicht auf den Entwurf eingebetteter Systeme zugeschnitten sind.

Wie bereits diskutiert, sind viele eingebettete Systeme Echtzeitsysteme. Bei ihnen kommt es nicht nur darauf an, dass die berechneten Ergebnisse logisch korrekt sind, sondern pünktlich zu einem bestimmten Zeitpunkt, einer sogenannten Deadline, zur Verfügung stehen. Echtzeitsysteme, die noch akzeptabel funktionieren, obwohl eine Deadline geringfügig versäumt wurde, sind weiche Echtzeitsysteme. Meist wird bei Nichteinhalten der Deadlines bei weichen Echtzeitsystemen die Qualität der Ergebnisse und der erbrachten Dienste schlechter, aber die Funktionalität wird noch bereitgestellt. Ein Beispiel (Pree et al., 2003) ist die Dekompression von Videodaten, die über ein Netzwerk geladen werden. Je langsamer die Dekompression erfolgt, desto schlechter ist die Qualität des angezeigten Videos. Bei sogenannten harten Echtzeitsystemen müssen hingegen alle Deadlines unter allen Umständen eingehalten werden, da sonst möglicherweise eine Katastrophe eintreten könnte. Beispiele für harte Echtzeitsysteme sind Flugzeugsteuerungen, oder im Automobilbereich das Motormanagement im Verbrennungsmotor, Bremsassistent oder der Auslöser eines Airbags.

Die Entwicklung von harten Echtzeitsystemen unterscheidet sich grundlegend von der von weichen Echtzeitsystemen. Ein hartes Echtzeitsystem muss unter allen möglichen Ausnahme- und Fehler-

bedingungen die zeitlichen Restriktionen einhalten. Es ist die Systemumgebung, in der ein hartes Echtzeitsystem eingesetzt wird, die vorgibt, welches exakte Zeitverhalten verlangt wird und welcher Grad an Parallelität beziehungsweise Verteilung auf Seite der Software zu beherrschen ist. Beispielsweise definieren die Charakteristika eines Motors die Anforderungen an die zugehörige Steuerungssoftware, also welche parallel ablaufenden chemischen und mechanischen Prozesse im Motor beobachtet werden müssen, in welchen Zeitabständen, z. B. vorgegeben durch die Motordrehzahl, die Messungen durch Sensoren und die Steuerungsimpulse durch Aktoren erfolgen müssen. Die Steuerungssoftware muss innerhalb der Zeitvorgaben die Berechnungen durchführen, um die richtigen Stellwerte an die Aktoren zu liefern, so dass das gewünschte Verhalten des Motors erzielbar ist. (Pree et al., 2003).

Ein derzeit besonders interessantes Forschungsthema im Bereich harter Echtzeitsysteme ist die automatische Übereinstimmung der Zeitachse der Umgebung (also der realen Welt) mit der Zeitachse des Rechners, auf dem die Software läuft. In der realen Welt ist Zeit kontinuierlich, also reellwertig, während digitale Rechner mit einem diskreten Zeitbegriff arbeiten. Da Rechnerplattformen für eingebettete Systeme zunehmend als verteilte Systeme eingesetzt werden, ist es mühsam und fehleranfällig, die Übereinstimmung der beiden Zeitachsen manuell herzustellen (Broy und Pree, 2003).

Genau diese Vorgehensweise muss aber in Ermangelung besserer Ansätze heutzutage in der Entwicklungspraxis eingebetteter Echtzeitsysteme verwendet werden. Stattdessen wäre es aber wünschenswert, eine automatische Übereinstimmung der Zeitachsen sicherzustellen. Eine tiefgehende formale Betrachtung dieser Problemstellung kann die Grundlage dafür liefern, das ungewollt nichtdeterministische zeitliche Verhalten heutiger Systeme zu bereinigen. Deterministisch bedeutet in diesem Kontext, dass das Ausgabeverhalten der Aktoren bei gegebenem Input an den Sensoren vorhersagbar ist. Das stellt eine Voraussetzung für die formale Verifikation und Implementierbarkeit der Software dar. Hier ist nach (Broy und Pree, 2003) die Entwicklung neuartiger Sprachen und Konzepte erforderlich, „die sich ausschließlich darauf konzentrieren, das gewünschte Zeitverhalten von eingebetteten Systemen auf verteilten Plattformen durch eine automatische Codeerzeugung zu gewährleisten“. Beispiele für brauchbare, aber sehr unterschiedliche Sprachansätze in diese Richtung sind Ansätze wie Esterel, Lustre, Signal und Giotto.

Kapitelübersicht

Dieses Kapitel soll zunächst eine Diskussionsbasis schaffen, inwieweit gängige Programmiersprachen wie C/C++ und Java zur Entwicklung eingebetteter Systeme geeignet sind. Der Leser soll

dann überzeugt werden, dass für die Programmierung eingebetteter Systeme im Vergleich zu gängigen Praktiken völlig neuartige Ansätze erforderlich sind. Im Folgenden wollen wir daher alternative Programmiersprachen zur Entwicklung ggf. nebenläufiger, eingebetteter Echtzeitsysteme überblicksartig betrachten. Danach werden die Sprachen Esterel und Giotto als zwei sehr unterschiedliche Vertreter für mögliche Lösungsansätze kurz vorgestellt. Eine Vermittlung tiefergehender Programmierkenntnisse ist dagegen weder möglich noch nötig.

4.1 Der Einsatz von C/C++ für eingebettete Systeme

Assembler und C im Zusammenspiel

Im Jahr 2000 wurden laut empirischen Erhebungen (Lewis, 2001) noch etwa 80 Prozent der eingebetteten Systeme in C entwickelt. Aus Performance-Gründen wird jedoch C immer noch in folgender Weise in Kombination mit Assemblersprache verwendet:

- Assemblersprache wird zur Realisierung echtzeitkritischer Anforderungen verwendet,
- C wird nach Assembler übersetzt und dann von Hand optimiert.

Gründe für die Verbreitung von C

Gründe für die immer noch weite Verbreitung von C auf diesem Gebiet sind:

- Assemblersprache ist häufig nicht mehr möglich (siehe oben), und C kommt der maschinennahen Programmierung am nächsten,
- Entwicklungszeit und -zyklen werden immer kürzer,
- die Komplexität der eingebetteten Systeme steigt,
- eingebettete Systeme erfüllen mehr und mehr sicherheitskritische Aufgaben,
- leistungsfähigere Hardware ermöglicht Hochsprachen, dennoch liefert die Programmierung in Maschinensprache schnellere und kürzere Programme (bis zu Faktor 10).

Die Anzahl der eingebetteten Systeme, die nicht mehr in C oder Assembler programmiert werden nimmt jedoch stetig zu. Dies ist u. a. mit der steigenden Systemkomplexität begründet. Durch den

Einsatz objektorientierter Hochsprachen in eingebetteten Systemen soll die gestiegene Komplexität kompensiert werden. Objektorientierte Hochsprachen ermöglichen ferner die einfachere Wiederverwendung von Code sowie eine komplett objektorientierte, teilweise automatisierte Entwicklung von der Analyse bis hin zur Realisierung. Auch um die Softwarequalität bei gleichzeitig möglichst geringen Entwicklungskosten zu steigern, werden immer mehr objektorientierte Programmiersprachen wie C++ oder auch Java verwendet. C++ ist wegen der vielen C-Sprachelemente und der damit verbundenen leichteren Eingewöhnung der Entwickler sehr populär. Embedded C++, kurz EC++, soll die Entwicklung erleichtern und zugleich die Kompatibilität zu gängigen Standards sicherstellen. Aber auch Java gewinnt nicht zuletzt wegen der großen Entwicklergemeinde und den verschiedenen direkt für eingebettete Systeme zugeschnittenen Varianten, an Bedeutung.

Sowohl Embedded C++ als auch die Java 2 Micro Edition (J2ME) stellen objektorientierte Programmiersprachen zur Verfügung, die hinsichtlich ihres Sprachumfangs im Vergleich zu C++ bzw. Java eingeschränkt wurden. So wurde z. B. bei EC++ die Mehrfachvererbung bzw. bei der CLDC (Connected Limited Device Configuration) der J2ME die Gleitkommaarithmetik entfernt. Ebenso sind viele Standard-APIs meist nicht vollständig verwendbar.

4.2 Embedded C++

Sprachumfang von EC++

Im Vergleich zu C enthält C++ einige Sprachumfänge, die den Einsatz von C++ zur Programmierung eingebetteter Systeme behindern, so z. B. die Ausnahmebehandlung mittels Exceptions sowie das Konzept der Mehrfachvererbung, die selbst dann schon Code Overhead generieren, wenn die Konzepte im konkreten Programm gar nicht verwendet werden. Aber auch Templates oder die Verwendung mancher Bibliotheksfunktionen wie etwa für Ein- und Ausgabe von Daten kann bei weitem mehr Code erzeugen, als dies vor dem Hintergrund der restriktiven Speicherplatzanforderungen eingebetteter Systeme vertreten werden könnte. Diesen Nachteilen von C++ tritt die Entwicklung von Embedded C++ (EC++) entgegen.

Ziel von EC++

Ziel von EC++ ist es, einen offenen Standard für die Entwicklung kommerzieller eingebetteter Systeme zu definieren (Plauger, 1999). Entwicklern von eingebetteten Systemen wird eine Untermenge von C++ geboten, die für C-Programmierer leicht verständlich und damit nutzbar ist. Diese Untermenge ist aufwärtskompatibel zur vollen

Version von Standard C++ und partizipiert daher an zahlreichen Vorteilen von C++. Insbesondere existiert ein sogenannter Styleguide, der die Programmierung eingebetteter Systeme mit Embedded C++ erleichtert. Embedded C++ wurde auf die speziellen Bedürfnisse bei der Programmierung eingebetteter Systeme konfektioniert. Die Entwicklung des Standards wurde durch das sogenannte „Embedded C++ Technical Comittee“ vorangetrieben. Diesem 1995 in Japan gegründeten Komitee gehören namhafte, vor allem japanische Unternehmen an. Unter www.caravan.net/ec2plus/ ist der Internetauftritt des Komitees zu finden.

Rahmenbedingungen von EC++

Folgende Rahmenbedingungen für die Auswahl einer geeigneten Untermenge von C++ hat dieses Komitee zur Definition von EC++ definiert:

- Die Spezifikation sollte so klein wie möglich sein, ohne die objektorientierten Elemente auszuschließen.
- Es sollten solche Funktionen und Spezifikationen vermieden werden, die nicht die Erfordernisse der Entwicklung eingebetteter Systeme erfüllen. Die drei Haupterfordernisse eingebetteter Systeme sind nach Ansicht des Komitees:
  - Die Vermeidung von exzessivem Speicherverbrauch
  - Keine unvorhersagbaren Reaktionen zu generieren
  - Die Lauffähigkeit des Codes im ROM zu garantieren
- Nichtstandard-Erweiterungen von C++ dürfen nicht in die Untermenge einfließen.
- Zielplattform der mit EC++ programmierten Anwendungen sind 32 Bit RISC MCUs (RISC = Reduced Instruction Set Computer; MCU = Micro Controller Units). Für Anwendungen mit 4 bzw. 8 Bit MCUs stellen nach Meinung des Komitees die Sprache C sowie Assemblersprachen eine ausreichende Umgebung dar.

Embedded C++ ist daher keine neue Sprachspezifikation die eine Konkurrenz zum existierenden C++ Standard darstellen würde, sondern vielmehr eine – bis auf eine Ausnahme – Untermenge von C++. Die Untermengen-Bedingung von EC++ zu C++ wird im folgenden Punkt aufgeweicht: Die speziellen EC++ Datentypen “float_complex” und “double_complex” besitzen kein Pendant im offiziellen C++ Standard.

4.2.1 Einschränkung: Das Schlüsselwort „mutable“

Objekte einer als „const“ deklarierten Klasse werden bei ihrer Realisierung in eingebetteten Systemen üblicherweise im ROM (Read Only Memory) abgelegt. Klassenelemente die als „mutable“ deklariert wurden, können auch dann geändert werden, wenn das Objekt selbst „const“ ist. Folglich können Klassen, welche ein „mutable“-Element besitzen, nicht länger im ROM gespeichert werden. Das technische Komitee hat deshalb entschieden, das „mutable“-Schlüsselwort nicht in die EC++ Spezifikation aufzunehmen.

4.2.2 Einschränkung: Ausnahmebehandlung

Ausnahmebehandlungen (engl. exception handling) sind zur kontrollierten Behandlung von Laufzeitfehlern von Vorteil, bergen jedoch einige Fallstricke für die Programmierer. So erweist es sich zunächst als schwierig, die Zeit, die zwischen dem Auftreten der Ausnahme und ihrer Weitergabe an die Behandlungsroutine vergeht, abzuschätzen. Wenn die Ablaufkontrolle vom Auftrittspunkt zur Behandlungsroutine übergeht, werden die Destruktoren aller Objekte aufgerufen, die seit Betreten des „try“-Blockes automatisch erzeugt wurden. Daher ist die Zeit, die zur Zerstörung der automatisch erzeugten Objekte benötigt wird, nur schwer abzuschätzen. Ausnahme-Mechanismen erfordern Compiler-generierte Datenstrukturen sowie eine Laufzeitunterstützung. Dies kann zu einem unerwarteten Anwachsen der Programmgröße führen. Der genaue Speicherverbrauch zur Abwicklung der Ausnahmebehandlung kann daher nicht exakt bestimmt werden.

Für Entwickler von eingebetteten Systemen ist es wichtig, die Ausführungszeit ihrer Programme zu kennen. Ebenso muss die erzeugte Programmgröße so klein wie möglich sein. Deshalb können die oben genannten Nachteile nicht ignoriert werden.

Aus diesen Gründen hat sich das technische Komitee dazu entschieden, die Ausnahmebehandlung nicht in den EC++ Standard aufzunehmen. Ausnahmebibliotheken werden folglich in der Standard EC++ Bibliothek nicht unterstützt.

4.2.3 Typidentifikation zur Laufzeit

RTTI

Um die Typidentifikation zur Laufzeit (engl. Run-Time-Type Identification, kurz: RTTI) zu unterstützen, wird ebenfalls Code Overhead erzeugt, da Typinformationen für polymorphe Klassen benötigt werden. Der Compiler erzeugt diese Informationen automatisch – selbst dann, wenn ein Programm RTTI gar nicht benutzt. Das technische Komitee hat deshalb RTTI nicht in die EC++ Spezifikation aufgenommen.

4.2.4 Namenskonflikte

Namespaces

Aufgrund der vergleichsweise geringen Dimensionierung des Arbeitsspeichers typischer Zielprozessoren von EC++ Programmen ist der Codeumfang dieser Programme stark limitiert. Aus diesem Grund treten Namenskonflikte selten auf und können ggf. durch die Verwendung statischer Klassenelemente vermieden werden. Die Funktionalität dedizierter Namensräume (engl. namespaces) ist daher nicht relevant für die EC++ Spezifikation und wurde deshalb nicht in sie aufgenommen.

4.2.5 Templates

Templates werden zum Erzeugen generischer Funktionen und Klassen verwendet, erzeugen jedoch wiederum Code Overhead. Da es durch die unachtsame Verwendung von Templates zu regelrechten „Code Explosionen“ kommen kann, hat sich das technische Komitee dazu entschlossen, sie nicht in die EC++ Spezifikation mit aufzunehmen.

4.2.6 Mehrfachvererbung und virtuelle Vererbung

Programme mit Mehrfachvererbung tendieren dazu, weniger lesbar, weniger wiederverwendbar und schwer wartbar zu sein. In den EC++ Standard wurde Mehrfachvererbung deshalb nicht aufgenommen. Da die virtuelle Vererbung (Spezifikationsvererbung,

Schnittstellenvererbung, engl. interface inheritance) nur in Kombination mit der Mehrfachvererbung vorkommt, ist auch sie kein Teil des Standards.

4.2.7 Bibliotheken

STL

Da Templates in EC++ aus vorgenannten Gründen keine Beachtung gefunden haben, werden auch Bibliotheken wie etwa die STL (Standard Template Library), welche Templates benutzen, nicht unterstützt. Einige aus der STL bekannten Klassen werden allerdings als Nicht-Template-Klassen unterstützt. Dazu zählen die Klassen „string“, „complex“, „ios“, „streambuf“, „istream“ und „ostream“; sie sind im EC++ Standard enthalten.

Die Bibliotheken für die Typen „wchar_t“ oder „long double“ werden im Bereich eingebetteter Systeme sehr selten benutzt. Ihre Aufnahme in den EC++ Standard wurde deshalb vom Komitee abgelehnt.

Dateioperationen

Bibliotheken für Dateioperationen sind abhängig vom Betriebssystem und werden daher nicht unterstützt.

Internationalisierung

Internationalisierungs-Bibliotheken benötigen einerseits viel Speicher und sind andererseits für die meisten Embedded Anwendungen nutzlos. Deshalb sind sie nicht Teil der EC++ Spezifikation.

4.2.8 EC++ Styleguide

Teil A

Der Teil A „Migrating from C Language to C++ Language“ des Styleguides beschäftigt sich mit Regeln zur Migration von C nach Embedded C++. Dieser Teil soll zum einen C-Programmierern den Umstieg von C nach EC++ erleichtern, zum anderen das Portieren bereits vorhandener C-Programme durch die dort dargestellten Migrationsrichtlinien beschleunigen.

Teil B

In Teil B „Guidelines for Code Size“ des Styleguides werden Hinweise gegeben, welche Sprachmittel man einsetzen sollte, damit der durch den Compiler generierte Bytecode möglichst klein ist.

Teil C

Teil C „Guidelines for Speed” erläutert Programmiertechniken, die eingesetzt werden sollten, um Code mit möglichst hoher Laufzeit- und Speichereffizienz zu generieren.

Teil D („Guidelines for ROMable Code“) zeigt, welche Programmkonstrukte verwendet werden dürfen, um Objekte im ROM ablegen zu können.

Teil D

4.3 Der Einsatz von Java für eingebettete Systeme

Java für die Programmierung eingebetteter Systeme einzusetzen mag zunächst als ein Widerspruch erscheinen, da es sich bei Java um eine Plattform-unabhängige, sehr aufwändige, objektorientierte Sprache handelt, die zunächst ihre Anwendung im Internetbereich fand und daher meist interpretiert wird. Hierfür kommt die Java Virtual Machine (JVM) zum Einsatz, eine registerlose und damit vergleichsweise langsame, virtuelle Stack-Maschine. Insgesamt wird eine Performanz von nur ca. 5–10% eines Native Compilers erreicht.

Geschwindigkeitseinbußen

Weitere Geschwindigkeitseinbußen gibt es bei Java u. a. auch wegen folgender Eigenschaften:

- Keine Pointer
- Viele Laufzeitüberprüfungen
- Sicherheitsüberprüfung des Bytecodes
- Synchronisation
- Symbolische Namensauflösung
- Langsame Array-Initialisierung
- Automatische Speicherverwaltung (Garbage Collection)

Nachteile

Weitere Nachteile für die Verwendung von Standard-Java im Bereich eingebetteter Systeme sind:

- Es bietet für den Umgang mit begrenzten Ressourcen wenig Unterstützung.
- Es verbraucht viel Speicherplatz (allein die Laufzeitumgebung benötigt schon 26 MByte).
- Hardwarezugriffe sind kaum möglich.
- Mannigfaltige, evtl. sogar unbekannte Ein- bzw. Ausgabeschnittstellen müssten angeboten werden.
- Es ist nicht echtzeitfähig.

Hindernisse für Echtzeitanwendungen

Im Umfeld von Echtzeitsystemen wurde man auf Vorteile von Java wie Sicherheit, Portabilität und die Unterstützung von Multithreading aufmerksam. Folgende Punkte sind aber hinderlich für den Einsatz von Java in Echtzeitanwendungen:

- **Schlechte Performance:** Java ist eine interpretierte Sprache; u. a. entstehen auch Performanceeinbussen bedingt durch die Virtuelle Maschine
- **Nichtdeterministisches Verhalten:** Java bietet keine garantierten Antwortzeiten. Zusätzlich besitzt Java einen Garbage Collection Mechanismus, der in undefinierten Zeitabständen unreferenzierte Objekte (wieder) freigibt. Dieser Mechanismus unterbricht die Programmausführung in unvorhersagbaren Zeitabständen.
- **Scheduling und Synchronisation:** Die Semantik der Synchronisation und des Schedulers in Java genügt den Anforderungen von Echtzeitsystemen nicht. In Echtzeitsystem muss bekannt sein, wenn/wann ein Task gestartet und beendet wird.
- **Speicherverwaltung:** In Java ist kein direkter Zugriff auf den Speicher möglich, alle Objekte werden auf dem Heap gespeichert und die Garbage Collection ist zuständig für das Abbauen von nicht mehr gebrauchten Objekten. Daher hat man keine Möglichkeit, den Lebenszyklus von Objekten zu kontrollieren.

Historie

Dennoch wurde Java ursprünglich für den Einsatz in eingebetteten Systemen entwickelt. 1991 wurde bei Sun von Patrick Naughton, Mike Sheridan und James Gosling das „Green Project“ gestartet. Ziel des Projekts war das Aufspüren der nächsten Trendwelle der Computertechnik. Aufgrund ihrer Vermutung, dass digitale Konsumgeräte und Computer immer mehr zusammenwachsen würden, wurde StarSeven (*7) entwickelt. Diese Multimedia-Fernbedienung für Unterhaltungselektronik war in der Lage, ein umfangreiches Spektrum an Plattformen zu steuern. Die Grundlage dafür bildete eine Programmiersprache namens „Oak“, die später in Java umbenannt wurde.

Obwohl Java also ursprünglich für eingebettete Systeme entwickelt wurde, setzte es sich vor allem wegen seiner flexiblen Verwendung auf unterschiedlichsten Plattformen, Netzwerkanbindungen und dem Konzept der Objektorientierung schnell auf Desktops, Servern und als Applets im Internet durch. Dennoch ist Java für eingebettete Systeme durch speziell für die Anforderungen dieser Systeme ausgerichtete Versionen interessant. Sie ermöglichen es, viele Java Vorteile auch auf *mobilen Systemen* mit geringen Ressourcen zu nutzen.

4.3.1 Java 1

In Java1 existieren zur Anwendungsentwicklung für eingebettete Systeme zwei Plattformen: Personal Java für leistungsstarke und Embedded Java für schwächere Systeme. Allerdings werden beide von Sun im Rahmen des „Sun End of Life (EOL) Process“ nicht mehr weiterentwickelt da neuere Produkte (J2ME) existieren. Online-Dokumentationen sind im Internet unter folgenden Adressen zu finden:

java.sun.com/products/personaljava
java.sun.com/products/imp/overview.html

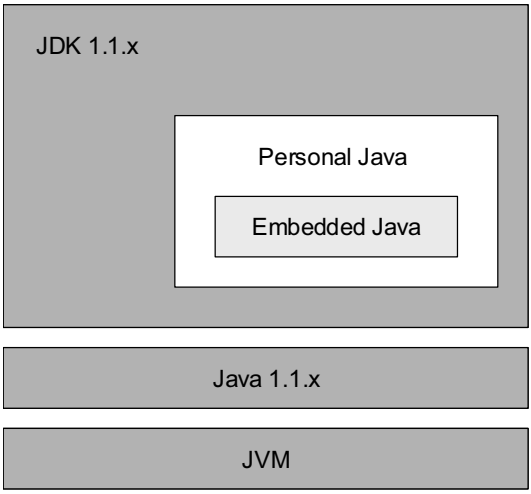


Abb. 4.1: Aufbau von Java 1

4.3.1.1 Personal Java

GUIs, Netzwerke

Diese Java Plattform ermöglicht die Entwicklung von Anwendungen mit graphischer Benutzerschnittstelle (java.awt) und Netzwerkverbindungen. Die Personal Java API (Application Programming Interface) ist von der JDK 1.1 API abgeleitet und nur um wenige neue APIs ergänzt. Sie umfasst eine virtuelle Maschine (z. B. Standard VM) und eine optimierte Version der Java Klassenbibliothek. Um Ressourcen auf den Zielgeräten zu sparen, ist Personal Java auf drei Ebenen konfigurierbar. Es ist möglich auf Paket-, Klassen- und Methodenebene genau die benötigten Elemente auszuwählen. Das frei verfügbare Entwicklungswerkzeug JavaCheck von Sun ermög-

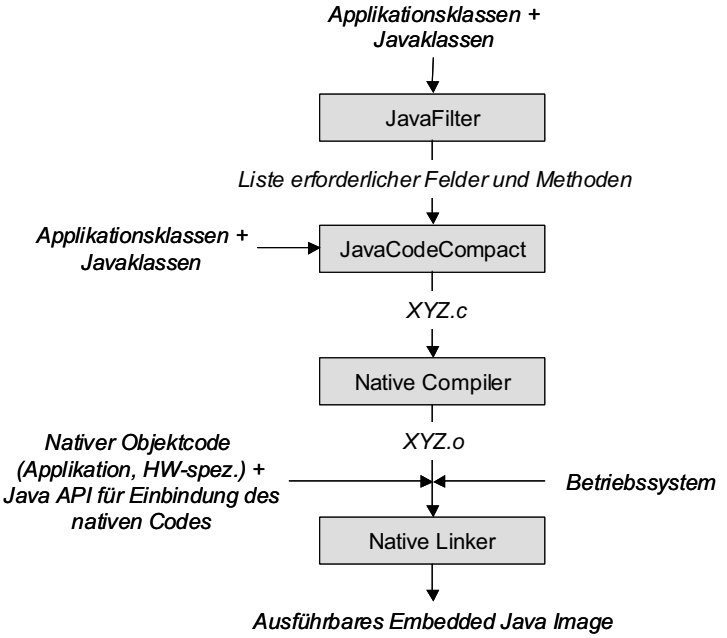
licht eine Überprüfung, ob eine Anwendung bzw. ein Applet auf einer Personal Java Plattform lauffähig ist.

4.3.1.2 Embedded Java

Beschränkte Ressourcen

Embedded Java ist für Geräte gedacht, deren Ressourcen für Personal Java nicht ausreichen. So werden z. B. keine Core APIs benötigt, sondern man kann je nach Anwendungsgebiet konfigurieren welche APIs verwendet werden sollen. Damit kann man fast alle APIs des JDK 1.1.7 bei minimaler Applikationsgröße verwenden. Allerdings ist der Entwicklungsprozess mit den von Sun zur Verfügung gestellten Werkzeugen aufwändiger als bei Personal Java, vgl. Abbildung 4.2.

Abb. 4.2: Entwicklungsprozess mit Embedded Java, Java 1



Optimierung

Nach der Entwicklung der Anwendung in Java wird mit dem JavaFilter Werkzeug eine Liste mit allen verwendeten Feldern, Klassen und Methoden erstellt. Diese Liste wird durch JavaCodeCompact weiter optimiert (beispielsweise werden nicht verwendete Codeteile entfernt) und in Datenstrukturen der Sprache C umgewandelt. Diese können dann mit einem C-Compiler für das jeweilige System in Objekt-Dateien übersetzt werden. Mit

JavaDataCompact ist es möglich, benötigte Daten wie Bilder oder HTML-Dateien umzuwandeln, um diese dann zu den Objekt-Dateien zu verlinken, falls auf dem Zielgerät kein Dateisystem verfügbar ist. Die so entstandenen Embedded Java Images können auf dem jeweiligen Gerät ausgeführt werden.

4.3.2 Java 2 (J2ME)

J2EE, J2SE, J2ME

Mit Java2 wurde die Java Technologie in drei Teilbereiche aufgeteilt: Java 2 Enterprise Edition (J2EE), Java 2 Standard Edition (J2SE) und Java 2 Micro Edition (J2ME). In letzterer greift Sun Personal Java und Embedded Java auf, um Anwendungen dieser Plattformen kompatibel zu halten. Jedoch wurden nicht monolithische Plattformen geschaffen, die dann umfangreich je nach Anwendung mühsam konfiguriert werden müssten, sondern modulare Konfigurationen und Profile. Diese bieten dann für verschiedene Anwendungsbereiche definierte APIs.

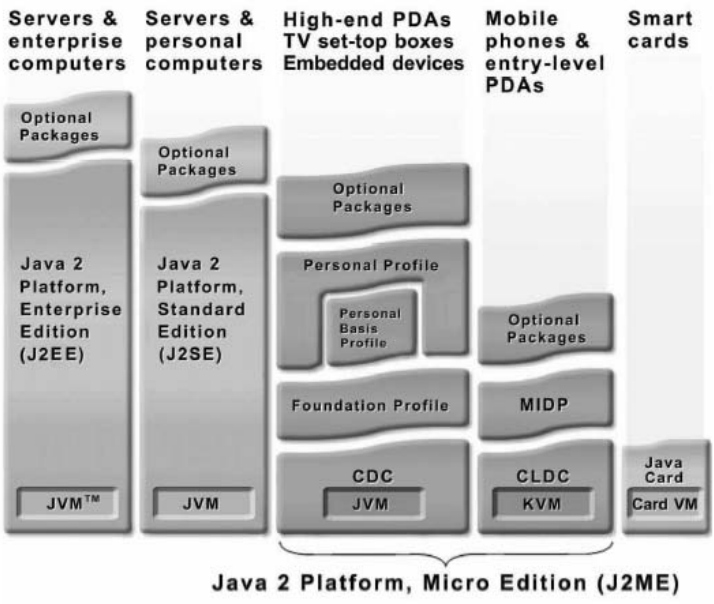


Abb. 4.3: Java 2 Architektur (Quelle: Sun)

Konfigurationen, Profile

Hierbei beschreiben *Konfigurationen* die Grundfunktionen der jeweiligen virtuellen Maschine (VM) für eine Klasse von Geräten mit ähnlicher Leistungsfähigkeit; vgl. Abbildung 4.3. Eine Konfiguration umfasst eine VM, Klassenbibliotheken und (minimale) APIs, die auf die jeweiligen Geräte zugeschnitten sind und stellt damit eine funktionstaugliche Laufzeitumgebung dar. *Profile* setzen auf den Konfigurationen auf und stellen weitere APIs für bestimmte Klassen von Zielgeräten zur Verfügung. Sie können ineinander verschachtelt sein oder aufeinander aufbauen. Allerdings können auch optionale Pakete wie z. B. das „RMI Optional Package“ oder das „JDBC Optional Package“ (beide für CDC) eingebunden werden.

4.3.2.1 Connected Device Configuration (CDC)

Die Connected Device Configuration umfasst eine voll funktionsfähige VM der Java 2 Plattform – die CDC HotSpot Implementation oder auch kurz CVM genannt – sowie eine kleine Anzahl von Klassenbibliotheken und APIs. Erst mit einem darauf aufbauenden Profil ist es möglich, eine Applikation zu entwickeln. Für diese Profile gibt es zusätzliche optionale Pakete, die weitere spezielle Funktionen bereitstellen. Die CDC kommt häufig in Geräten mit einem 32 Bit Prozessor zum Einsatz. Sie benötigt ca. 2 MByte RAM und in etwa 2,5 MByte ROM für die Java Umgebung. Zurzeit existieren die folgenden drei Profile für die CDC.

Foundation Profile

(1) Foundation Profile:
Das Foundation Profile setzt auf dem CDC auf. Es dient als Basis für weitere Profile und stellt deshalb keine Klassen für das Benutzerinterface zur Verfügung. Es dient im Wesentlichen dazu, die Pakete des CDC konzeptuell verfügbar zu machen. Es umfasst Socket-Klassen und ermöglicht Internalization und Localization. Außerdem enthält es Klassen aus den Paketen java.lang, java.lang.ref, java.lang.reflect, java.net, java.security, java.text, java.util, java.util.jar und javax.microedition.io. Eine graphische Benutzeroberfläche (GUI) wird nicht unterstützt. Meist wird dieses Profil nicht unmittelbar selbst verwendet, sondern dient lediglich als Grundlage für weitere Profile.

Personal Basis Profile

(2) Personal Basis Profile:
Dieses Profil baut auf dem Foundation Profile auf und erweitert dieses um die sogenannten „lightweight“ (leichtgewichtigen, schlanken) Komponenten aus dem Paket java.awt. Dies bedeutet, dass eine einzige Instanz von java.awt.Frame erlaubt ist, um

Komponenten aufzunehmen. Im Gegensatz dazu sind „heavyweight“ GUI-Komponenten wie etwa java.awt.Button oder java.awt.Panel nicht vorhanden. Außerdem stehen javax.microedition.xlet sowie javax.microedition.xlet.ixc für xlet-Anwendungen zur Verfügung. Xlets sind von den Java TV APIs abgeleitete Anwendungen, deren Lebenszyklus genau wie der von Applets von der Software, die sie aufgerufen hat, kontrolliert wird.

Personal Profile

(3) Personal Profile:
Dieses Profil schließt das Personal Basis Profile als Submenge (damit folglich auch das Foundation Profile) ein und beinhaltet im Gegensatz zum Personal Basis Pofile das Paket java.awt komplett. Außerdem können Anwendungen, die unter Personal Java (siehe Java1) entwickelt wurden, in dieses Profil überführt werden. Es eignet sich insbesondere für die Softwareentwicklung für PDAs (Personal Digital Assistants).

4.3.2.2 Connected Limited Device Configuration (CLDC)

Mobiltelefon, PDA

Diese Konfiguration wurde speziell für Geräte mit wenig Speicher (zwischen 128 und 512 KByte) und 16 bzw. 32 Bit Prozessoren wie beispielsweise Mobiltelefone, Pager oder kleine PDAs entwickelt. Sie stellt eine Untermenge des CDC (siehe oben) dar. CLDC Programme sind folglich auch auf CDC Plattformen lauffähig, falls die erforderlichen Profile zur Verfügung stehen.

KVM

Die CLDC umfasst eine VM, oft handelt es sich hierbei um die KVM, sowie grundlegende Klassenbibliotheken. Das „K“ der KVM ist hierbei als Abkürzung für „Kilobyte“ zu interpretieren, weil der Speicherplatzbedarf dieser virtuellen Maschine sehr gering ist. Bereits ab 70 KByte sind ausreichend. Datentypen wie „float“ oder „double“ werden nicht unterstützt, da die Prozessoren der Zielplattformen oft keine Gleitkomma-Arithmetik zur Verfügung stellen. Anstelle von java.net steht das Generic Connection Framework zur Verfügung. Mit ihm ist es Herstellern eingebetteter Systeme möglich, Geräte auf die KVM zu portieren und weitere Verbindungsprotokolle wie z. B. Bluetooth oder Irda zu unterstützen. Alle Klassen-Dateien müssen, bevor sie auf der KVM ausgeführt werden können, durch einen Pre-Verifier auf dem Desktop geprüft werden. Neben einigen optionalen Paketen für die CLDC wie beispielsweise die Wireless Messaging API (WMA) oder die Mobile Media API (MMAPI) gibt es zurzeit die beiden folgenden Profile.

MIDP

Mobile Information Device Profile (MIDP):
Dieses Profil ist für mobile Geräte wie Mobiltelefone oder Pager ausgelegt und enthält Pakete für den Zugriff auf persistenten Speicher sowie zur Erstellung von Benutzerschnittstellen. Da letztere geräteunabhängig in Form sogenannter MIDP-Programme (oft auch kurz MIDlets) beschrieben werden und kein konkretes, gerätespezifisches Layout festgelegt wird, können Darstellungsunterschiede auf den Zielgeräten die Folge sein. So können beispielsweise Displayauflösung oder Farbtiefe variieren. Detailinformationen zum MIDP sind im Internet unter java.sun.com/products/midp/overview.html zu finden.

IMP

Information Module Profile (IMP):
Dieses Profil enthält eine Submenge des MIDP. Es wurde im Rahmen des Java Community Prozesses (JCP) im Wesentlichen von den Firmen Siemens und Nokia entwickelt. Die zugehörigen APIs beinhalten Klassen zur Erstellung von „IMLets“ (IMP-Programme), zur Kommunikation und zur Ein-/Ausgabe. Damit ist z. B. durch die unterstützte Untermenge des HTTP (sowohl TCP/IP als auch nicht IP basiert) ein Zugriff auf WAP- (Wireless Access Protocol) oder iMode- (Interactive Media on Demand) Informationen möglich. Das MIP ist vor allem für Geräte, die über keine graphische Benutzeroberfläche verfügen bzw. nur vergleichsweise kleine Displays bieten gedacht (Beispiele: Notrufsäulen, Parkuhren, Alarmsysteme, Ferndiagnose- und Fernwartungssysteme). Weiterführende Informationen können unter java.sun.com/products/imp/overview.html detailliert nachgelesen werden.

4.3.3 JavaCard

Smart Cards

Smart Cards sind ein fester Bestandteil unseres Lebens geworden. Sie werden z. B. als SIM-Karten (SIM = Subscriber Identity Module) in Mobiltelefonen oder als Zugangskarten für gesicherte Bereiche eingesetzt. Eine JavaCard ist eine Smart Card auf der Java-Programme laufen können. JavaCards unterliegen demnach der Standardisierung Smart Cards nach der ISO/IEC 7816 Norm.

ISO/IEC 7816

Diese Norm legt die physikalischen Eigenschaften von Chipkarten und Protokollen für die Kommunikation mit dem Kartenleser fest. Er besteht aus den Teilen ISO/IEC 7816-1 bis ISO/IEC 7816-10. Dort sind unter anderem Anordnung und Größe der Kontakte

(ISO/IEC 7816-2) oder Material und Format der Karte (ISO/IEC 7816-2) standardisiert.

Software

Bei der auf der Karte laufenden Software unterliegen die Hersteller allerdings nur sehr wenigen Einschränkungen. Dies führt neben anderen Nachteilen zu einer großen Vielfalt an Karten, die meist untereinander so gut wie nicht kompatibel sind. Die Entwicklungskosten sind aufgrund nicht portabler Implementierungen vergleichsweise hoch; meist ist eine Anwendung sogar nur auf einer bestimmter Hardware-Version einer Karte lauffähig. Dies birgt den weiteren Nachteil, dass Entwickler sehr detaillierte Kenntnisse über die verwendete Hardware besitzen müssen. Oft ist bereits ein Update der Software im Nachhinein schon nicht mehr möglich.

JavaCard

Die JavaCard Technologie bietet hier viele Vorteile bei der Entwicklung und dem Einsatz von Smart Cards.

Smart Card Typen

Man unterschiedet im Wesentlichen zwei Gruppen von Smart Cards. Karten die nur zum Speichern von Daten verwendet werden (sogenannte Memory Chip Cards) und Karten mit eigenem Mikroprozessor. Erstgenannte verfügen nur über eine sehr eingeschränkte Menge vordefinierter Funktionen, die sich im Wesentlichen auf das Speichern und Verarbeiten der Daten beschränken. Mikroprozessorkarten beinhalten eine CPU sowie mehrere Arten von Speicher: ROM (Read-Only Memory) für Betriebssystem und Programme, EEPROM (Electrically Erasable Programmable Read-Only Memory) für Daten und Programme und RAM (Random Access Memory) als Arbeitsspeicher. Die Größe des Speichers reicht von wenigen Bytes bis hin zu mehreren MByte und wird meist genauso wie die Leistungsfähigkeit der CPU nur durch finanzielle Vorgaben eingeschränkt. Da Smart Cards über keine eigene Stromversorgung verfügen, beziehen sie die benötigte Energie von den Lesegeräten. Die Kommunikation kann hierbei sowohl drahtlos als auch über Kontakte erfolgen.

Hardware Minimalanforderungen

Die Minimalanforderungen von JavaCard an die Hardware sind mindestens 16 KByte ROM, 8 KByte EEPROM sowie 256 KByte RAM (Quelle: www.javawold.com). Die Architektur dieser Karten kann man wie in Abbildung 4.4 (aus www.javaworld.com) dargestellt in fünf Schichten einteilen.

Architektur

Wie Abbildung 4.4 zeigt, setzt die JavaCard VM auf einer speziellen CPU und deren Betriebssystem auf und abstrahiert von diesen, so dass der Zugriff der Applets auf die Hardware nur über die VM möglich ist. Die Aktivierungsdauer der VM ist mit jener der Smart Card selbst identisch. Falls die Smart Card nicht mit Strom versorgt wird, wird die VM nur temporär angehalten. Das JavaCard

Framework und andere (optionale) Klassen (sogenannte Industry Add-on Classes) stellen den Applets zusätzliche Funktionen zur

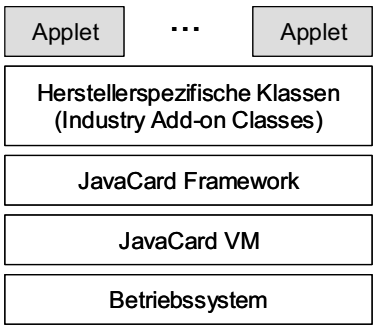


Abb. 4.4: Architektur JavaCard 2.0

Verfügung. Das JavaCard Framework ab 2.0 beinhaltet im Wesentlichen:

- javacard.framework: definiert Klassen wie Applet, PIN, System, Util und APDU (Application Protocol Data Units)
- javacardx.framework: objektorientiertes Design für ein ISO/IEC 7816-4 kompatibles Dateisystem. Es unterstützt „Elementary Files“ (EF), „Dedicated Files“ (DF) und dateiorientierte ADPUs.
- javacardx.crypto und javacard.cryptoEnc: zur Verschlüsselung von Daten

JavaCard Applets

JavaCard Applets:
JavaCard Applets sind Instanzen von Klassen, die von der Klasse javacard.framework.Applet abgeleitet wurden. Da mehrere Applikationen auf eine Smart Card geladen werden können, ist jedes Applet eindeutig durch seine *AID* (Application Identifier, ISO/IEC 7816-5) gekennzeichnet. Die vier Methoden *install*, *select*, *deselect* und *process* stellen die Schnittstelle von JavaCard Applets dar. Aus Sicherheitsgründen steht für Applets eine Sandbox zur Verfügung. Daten können je nach Bedarf in flüchtigen Objekten im RAM oder dauerhaft im EEPROM gespeichert werden.

Für die Kommunikation zwischen der Smart Card und der Lesestation (engl. Card Acceptance Device, kurz CAD) wird ein Protokoll benötigt. Smart Card und CAD verwenden zu diesem Zweck APDUs (Application Protocol Data Units). Hier wird grundsätzlich zwischen Befehlen (Command APDU) und Antworten (Response APDU) unterschieden. Dabei übernimmt die Smart Card die Rolle des passiven Kommunikationspartners. Sie wartet auf den Empfang

einer APDU vom CAD, verarbeitet diese und sendet dem CAD eine Antwort zurück.

Der Byte-Code, der nach seiner Übersetzung aus Java-Code im „Class File Format“ vorliegt, ist in diesem Format allerdings nicht direkt auf die Karte ladbar. Dazu muss der Byte-Code zunächst in das sogenannte CAP-Format konvertiert werden. Neben der Übersetzung nimmt der Konverter auch eine digitale Signierung der Datei vor. Dies verhindert das Umgehen von Sicherheitsüberprüfungen der VM durch manipulierte Applets.

Da viele JavaCard-Hersteller auf proprietäre CAP-Formate zurückgreifen, ist eine Kartenapplikation nach der Überführung in das CAP-Format eines Kartenherstellers auf Karten anderer Hersteller nicht mehr ladbar.

Zusammenfassung JavaCard

Zusammenfassung:
Bei herkömmlichen Smart Cards sind Betriebssystem und Anwendungen nicht klar getrennt. Daher sind die Chipkarten der unterschiedlichen Hersteller untereinander nicht kompatibel. Mit Hilfe von JavaCard entwickelte Anwendungen sind Plattformunabhängig; sie sind auf jeder Smart Card, die über eine VM verfügt, lauffähig. Dank der Multiapplikationsfähigkeit beschränkt nur der Speicherplatz auf der Karte die Anzahl der installierten Programme. Applikationen können, anders als bei herkömmlichen Karten, nachträglich installiert und auf den neuesten Stand gebracht bzw. auch ggf. wieder deinstalliert werden.

Allerdings können JavaCard Anwendungen, was die Geschwindigkeit und Leistungsfähigkeit der Hardware betrifft, nicht mit herkömmlichen Smart Cards Applikationen konkurrieren. Dies macht sich vor allem bei den für Smart Cards relevanten, teilweise rechenintensiven kryptographischen Algorithmen bemerkbar. Vor dem Hintergrund der stetig anwachsenden Rechenleistung der Chipkarten-Prozessoren, ist hier zukünftig jedoch mit einer Entschärfung der Situation zu rechnen. Allerdings schränkt derzeit das Herstellerspezifische CAP-Formart die Portabilität der Anwendungen auf Byte Code Ebene immer noch ein.

4.3.4 Echtzeiterweiterungen für Java

Eine Echtzeitanwendung unterscheidet sich von einer herkömmlichen durch die zusätzlichen Anforderungen an ihr zeitlichen Verhaltens. Dies bedeutet, dass Abläufe zeitlichen Einschränkungen bzgl. ihres Start- und Endzeitpunktes unterliegen. Da bei der Ent-

wicklung von Java auf solche Aspekte nur sehr wenig Rücksicht genommen wurde, eignet es sich (ohne zusätzliche Erweiterungen) nicht für Echtzeitanwendungen. Oft werden deshalb zeitkritische Komponenten immer noch in C oder C++ geschrieben.

Echtzeit-anforderungen

Wie bereits diskutiert, unterscheidet man aufgrund ihres Einsatzgebietes und der zeitlichen Anforderungen zwischen „harten Echtzeitanwendungen“ und „weichen Echtzeitanwendungen“. Bei ersteren muss a priori analysiert und nachgewiesen werden, dass alle Echtzeitanforderungen eingehalten werden können. Dies ist einerseits sehr aufwändig und erfordert andererseits detaillierte Kenntnisse über die verwendete Hardware (z. B. Cachestrategien, Pipelining usw.). Bei weichen Echtzeitanwendungen genügt es in den meisten Fällen, das zeitliche Verhalten empirisch zu bestimmen. Sämtliche bis dato bekannten Java-Echtzeitvarianten eignen sich lediglich für weiche Echtzeitanwendungen. Derzeit wird Java im Bereich der eingebetteten Systeme daher überwiegend in Multimediakomponenten bzw. mobilen Geräten (Mobiltelefone, PDAs usw.) eingesetzt. Hier findet vor allem die KVM (Kilobyte Virtual Machine) der J2ME (Java 2 Micro Edition) Anwendung (siehe oben).

Ansätze zum Einsatz von Java für Echtzeitsysteme

Mittlerweile gibt es bereits mehrere Ansätze, Java für den Einsatz in echtzeitkritischen eingebetteten Systemen aufzubereiten. Verbesserungsmöglichkeiten, die hierfür in Betracht kommen sind:

- Entwicklung einer Echtzeitspezifikation für Java – *RTSJ* (real-time specification for Java) definiert Standards für echtzeitfähiges Java (siehe www.rtj.org)
- Der Einsatz spezieller, leistungsoptimierter, *echtzeitfähiger JVMs* (Beispiel: Die JamaicaVM des AJACS Projektkonsortiums mit Beteiligung der Universität Karlsruhe (TH), siehe www.ajacs.de)
- Verwendung sogenannter *Just-in-Time Compiler,* die zur Laufzeit jeweils genau den Teil des Java Byte Codse in die Maschinensprache des jeweiligen Zielrechners übersetzen, der gerade gebraucht wird
- Verwendung des *JNI* (Java Native Interface) zur Einbindung sogenannter nativer Methoden (etwa aus C/C++); derzeit aber noch leistungsschwach

Um die Unzulänglichkeiten von Java im Zusammenhang mit der Echtzeit-Verarbeitung zu lösen, fand sich im Juni 1998 unter der Federführung des NIST (National Institute of Standards and Technology) eine Gruppe zusammen, die Anforderungen für ein

echtzeitfähiges Java erarbeiten sollte. An dieser Gruppe waren insgesamt 37 Firmen beteiligt. Da die gemeinschaftliche Umsetzung der erarbeiteten Ergebnisse misslang, bildeten sich zwei Gruppen, die jeweils ihre eigenen Vorstellungen in die Spezifikation einbringen wollten.

RTSJ, Real-Time Core Erweiterungen

Eine Gruppe wurde von Lizenznehmern der Firma Sun gebildet, die andere bestand aus Herstellern, die unabhängig bleiben wollten. Sun lieferte ihren Beitrag mit der „Real-Time Specification for JAVA“, kurz RTSJ. Die unabhängigen Hersteller lieferten ihren Vorschlag mit den „Real-Time Core Extensions for the JAVA Platform“.

Beide Vorschläge basieren auf der Definition von Objekten, die nicht der globalen Speicherplatzverwaltung (Heap, Garbage Collection) unterliegen und darauf arbeitenden echtzeitfähigen Routinen, die nicht von der Garbage Collection beeinflusst werden und diese bei Bedarf unterbrechen können. Beide unterscheiden sich im Wesentlichen in der Form, wie garantiert wird, dass Echtzeitroutinen keine Heap-Objekte manipulieren können und darin, wie der direkte Zugriff auf Hardware erfolgt.

4.3.4.1 Real-Time Core Erweiterung

Die Real-Time Core Erweiterung wurde im September 2000 vom sogenannten J-Consortium in der Version 1.0.14 veröffentlicht (siehe www.j-consortium.org). Sie enthält zwei verschiedene APIs, zum einen die sogenannte Baseline Java API für Nicht-Echtzeit-Threads in Java und zum anderen die Real-Time Core API für Echtzeit-Tasks. Ein weiterer, auf dem NIST-Dokument basierender Ansatz, ist die Basic Real-Time Java Specification, die eine sehr einfache Lösung darstellt und als Alternative oder Ergänzung zum Real-Time Core gedacht ist.

4.3.4.2 Real Time Specification for Java (RTSJ)

Die RTSJ wurde im Rahmen des Java Community Process (JCP) Ende 2001 veröffentlicht (siehe www.rtj.org). Im Folgenden gehen wir auf einige Aspekte der RTSJ genauer ein. Zur Definition der RTSJ wurde zunächst ein Anforderungskatalog erstellt, der die Anforderungen an eine Echtzeiterweiterung für Java spezifizierte. Diese Anforderungen sind im Einzelnen:

Anforderungen

- RTSJ soll keine Spezifikationen beinhalten, welche die Anwendung auf spezielle Java-Umgebungen beschränken.
- RTSJ soll abwärtskompatibel sein. Nicht-Echtzeit Java-Programme sollen uneingeschränkt auch unter RTSJ-Implementierung ausführbar sein.
- Die Portabilität von Java muss weiterhin möglich sein.
- RTSJ soll den momentanen Stand der Echtzeittechnik abdecken, aber auch für zukünftige Entwicklungen offen sein.
- RTSJ soll als erste Priorität deterministische Ausführung ermöglichen.
- Es sollen in Java keine syntaktischen Erweiterungen, wie z. B. neue Schlüsselwörter, eingeführt werden.

Lösungen

Dieser Anforderungskatalog wurde wie folgt umgesetzt:

- **Threads und Scheduling:** Die RTSJ fordert mindestens 28 Prioritätsstufen für Echtzeit-Threads und weitere 10 Stufen für „normale“ Threads. Sollten mehrere Threads die gleiche Priorität haben, werden diese nach dem FIFO Prinzip abgearbeitet. Es stehen NoHeapRealtimeThreads und RealTimeThreads zur Verfügung, wobei RealTimeThreads nur eine kleinere Priorität als der Garbage Collector haben können und sich somit im Gegensatz zu den NoHeapRealtimeThreads nur für „weiche“ Echtzeitanforderungen eignen. Zusätzlich können Scheduler dynamisch geladen werden, wenn das Interface Schedulable implementiert ist. Neben dem minimalen Priority-Scheduler können Implementierungen der Spezifikationen weitere willkürliche Scheduling-Algorithmen beinhalten.
- **Speicher:** Da, wie bereits erwähnt, die Garbage Collection Probleme bei der Synchronisation und dem Scheduling mit sich bringt, wurden neue Speicherbereiche außerhalb des Heaps (und damit auch außerhalb der Garbage Collection) definiert: der sogenannte *Scoped Memory* existiert erst nach dem er betreten (durch Thread oder „enter()“) wurde und dann nur solange Threads mit einer Referenz darauf existieren. Danach wird er wieder freigegeben. Objekte, die im sogenannten *Immortal Memory* erzeugt wurden, haben eine mit der Applikation identische Lebensdauer. Für diesen Speicher wird keinerlei Garbage Collection durchgeführt. Beim *Heap Memory* handelt es sich um den „normalen“ Speicher mit Garbage Collection. *Physical Memory* und *Raw Memory* sind Speicher mit verschiedenen Ansprechzeiten bzw. für Speicherzugriffe auf Byte-Ebene.

- **Synchronisation:** Falls mehrere Threads gleicher Priorität rechenbereit sind, werden diese nach dem FIFO Prinzip abgearbeitet. Die RTSJ fordert hierfür die Einrichtung einer Warteschlange für jedes Prioritätslevel. Falls ein Thread während der Ausführung blockiert wird, wandert er in der Prioritätswarteschlange wieder nach hinten, muss sich also neu „anstellen". Ferner wird mindestens die Implementierung des Priority Inheritance Protocol gefordert. Hier wird zu einem bestimmten Zeitpunkt die Priorität eines gesperrten Threads angehoben, um die Ausführung zu beschleunigen. Um eine nicht blockierende Kommunikation zwischen Threads (sowohl NoHeapRealTimeThreads als auch RealTimeThreads) zu gewährleisten, führt die RTSJ die sogenannten „free queues" ein.
- **Zeit und Timer**: Um relative oder absolute Zeit mit einer Genauigkeit im Nanosekundenbereich realisieren zu können, existieren spezielle Klassen.
- **Asynchrone Ereignis Behandlung:** Um die bestimmte zeitliche Ausführung einer Aktion nach einem Ereignis (sowohl selbst programmierte als auch externe wie z. B. Interrupts vom Betriebssystem) zu ermöglichen, wurden die Klassen AsyncEvent und AsyncEventHandler eingeführt. Hierbei repräsentiert AsyncEvent das zu überwachende Ereignis und der AsyncEventHandler enthält den nach Eintritt des Ereignisses auszuführenden Code.

Implementierungen

Bisher existieren nur wenige Implementierungen der RTSJ, die sich darüber hinaus größtenteils noch in der Entwicklung befinden. Neben RI, der Referenzimplementierung der RTSJ von TimeSys gibt es JRate, eine Open Source Lösung mit GNU Java Comiler sowie AJile, eine Implementierung der RTSJ auf Basis der CLDC 1.0 für aJ-80 und aJ-100 Chips.

JamaicaVM

Die JamaicaVM stellt eine der verbreitetsten Implementierungen der RTSJ dar. Ihre Erfinder, Teilnehmer des Projekt-Konsortiums AJACS (Applying Java to Automotive Control Systems, siehe www.ajacs.org bzw. www.ajacs.de), mittlerweile in einem Spinn-Off der TH Karlsruhe organisiert (ww.aiacs.com), bieten neben der VM auch spezielle Entwicklungswerkzeuge an (z. B. ein Plugin für Eclipse). Von der JamaicaVM werden mehrere Plattformen, darunter auch Echtzeitbetriebssysteme wie VxWorks, QNX oder verschiedene Linuxvarianten, unterstützt.

4.4 Synchrone Sprachen

Anwendungsgebiete

Synchrone Sprachen wie Lustre, Esterel, Signal usw. dienen zur Spezifizierung, Modellierung, Validierung und Implementierung von eingebetteten Systemen. Da ihre mathematische Grundlage formale Beweise ermöglicht, können damit Systeme mit hohen Sicherheitsanforderungen entwickelt werden.

Synchrone Sprachen haben enormes Interesse bei vielen führenden Firmen, die automatische Steuerungen für sicherheitskritische Systeme entwickeln, geweckt. So wurde z. B. Lustre von Schneider Electric zur Entwicklung von Steuerungen für Atomkraftwerke und von Aerospatiale zur Entwicklung von Steuerungen für den neuen Airbus verwendet. Esterel wurde erfolgreich von Dassault Aviation zur Entwicklung von Flugzeug-Steuerungen für den Rafale-Fighter eingesetzt und Signal von Snecma zur Entwicklung von Flugzeugmotoren benutzt. Weiterhin interessieren sich ST Microelectronics, Texas Instrument, Motorola und Intel für Esterel in Bezug auf Chip-Design mithilfe dieser Sprache (vgl. www-sop.inria.fr/cma/slap/). Der Hauptvorteil, den diese Firmen hervorheben, ist die formale semantische Fundierung der synchronen Sprachen; sie erlaubt eine formale Verifikation.

Synchrone Sprachen zeichnen sich dadurch aus, dass ihr Ein-/Ausgabeverhalten unabhängig von tatsächlichen Ausführungszeiten und -reihenfolgen beschrieben wird. Modellierungswerkzeuge, welche teilweise eine synchrone Semantik implementieren, dienen ebenfalls dazu, eine bestimmte Funktionalität auf einer höheren Abstraktionsebene zu beschreiben als dies mit klassischen imperativen (C/C++, Java) oder funktionalen Sprachen möglich ist. Beide Konzepte haben zunehmende Bedeutung für den Entwurf reaktiver Systeme, welche – oft sicherheitskritische – Steuerungsaufgaben wie z. B. die Steuerung von Aufzügen oder Flugzeugen übernehmen.

Historie

Das wesentliche Konzept dieser Sprachen ist, dass die meisten Anweisungen keine Zeit verbrauchen und ein Zeitverbrauch explizit zu programmieren ist. Synchrone Sprachen entstanden Anfang der 1980er Jahre basierend auf dem Modell der „perfekten Synchronie“. Anstoß war die Erkenntnis, dass herkömmliche Technologien nur noch bedingt für die Entwicklung von eingebetteten Systemen geeignet waren (Berry, 1998). Sie konnten in der Regelungstechnik schnell Fuß fassen, da sie sich nicht stark von den dort implizit verwendeten Modellen unterschieden. Synchrone Sprachen fassten in den 1990-ern auch Fuß im Hardware-Entwurf, nachdem man große Ähnlichkeiten des synchronen Modells mit den Modellen des

Schaltungsentwurfs entdeckte. Seit dem Advent synchroner Sprachen wächst das Interesse der Industrie an ihnen kontinuierlich.

Synchrone Sprachen werden auch heute noch ständig weiterentwickelt und daran wird sich aller Voraussicht nach auch in absehbarer Zukunft nichts ändern. Die Entwicklung der synchronen Sprachen bedient sich vielerlei Techniken aus verschiedensten Bereichen der Informatik (Berry, 1998): Regelungstechnik, Automatentheorie, binäre Entscheidungsbäume, konstruktive Logik sind nur einige davon.

Esterel

Esterel ist eine imperative Sprache, welche derzeit eine industrielle Verbreitung erfährt. In der Industrie werden jedoch hauptsächlich graphische synchrone Sprachen wie synchrone Varianten von Statecharts (μ-Charts) verwendet.

Perfekte Synchronie

Synchrone Sprachen verwenden das sogenannte *„perfekt synchrone"* Kommunikationsprinzip. Es unterstellt auf der Ebene der Spezifikation zunächst, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen, ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Zeit vergeht lediglich in stabilen Systemzuständen.

Die Annahme der Rechtzeitigkeit hat mehrere Vorteile, die wir in Abschnitt 4.5.2 eingehend diskutieren werden. Bei den sogenannten „synchronen Sprachen" wie etwa Esterel wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt.

Modellierungstechniken

Darüber hinaus müssen diese Konzepte von der Ebene der Programmiersprachen auf die Ebene der graphischen Modellierungstechniken erweitert werden. Hierzu gibt es ebenfalls schon Lösungen für die Modellierung zustandsbasierter Systeme, z. B. für die an Statecharts angelehnte Techniken SyncCharts Argos oder μ-Charts (Scholz, 1998), (Scholz, 2001). Letztere gibt dem Entwickler methodische Hilfestellungen bei der Verwendung von Statecharts, um ein eingebettetes System von der ersten, gänzlich architekturabhängigen Idee bis hin zur ggf. verteilten Implementierung zu konstruieren und dann zu partitionieren.

4.5 Ereignisbasierter Ansatz am Beispiel von Esterel

Esterel ist eine textuelle, perfekt synchrone Sprache die für die Programmierung reaktiver Systeme auf der Ebene des Systementwurfs (engl. system design) entwickelt wurde. Nach einem knappen historischen Einblick wird die Syntax der Sprache

beschrieben. Esterel besitzt eine mathematisch definierte und damit formale Semantik, die in diesem Rahmen nur in Ansätzen und lediglich informell erläutert werden kann.

4.5.1 Historie

Esterel wurde in einem gemeinsamen Projekt von der Ecole Nationale Supérieure des Mines de Paris (ENSMP) und vom Institut National de Recherche en Informatique et Automatique (INRIA) entwickelt. Grund hierfür war, dass sich „herkömmliche" Programmiersprachen für die Ansteuerung eines Roboters, den J.-P. Marmorat und J.-P. Rigault bauten, nicht geeignet waren, da sich die Struktur des reaktiven Systems „Roboter" nicht als Programm wiedergeben ließ. Das Esterel ist ursprünglich eine Bergkette zwischen Cannes und St. Raphaël, der Name erinnerte die Entwickler an „Echtzeit" (frz. temps réel). Gérard Berry vom INRIA entwickelte später eine formale Semantik für die Sprache, basierend auf der Hypothese der perfekten Synchronie. Mittlerweile gibt es von der Firma Esterel Technologies Inc. (www.esterel-technologies.com) eine industrielle Unterstützung, so dass bereits zahlreiche Firmen aus der Luftfahrt- und Automobilindustrie teilweise in Esterel entwickeln.

4.5.2 Hypothese der perfekten Synchronie

Hypothese

Synchrone Sprachen verwenden das sogenannte *„perfekt synchrone"* Kommunikationsprinzip (Berry, 1998). Es handelt sich hierbei um eine Hypothese, die auf der Ebene der Systembeschreibung zunächst unterstellt, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Ein System arbeitet also perfekt synchron, genau dann wenn. es ohne Zeitverzögerung auf externe Ereignisse reagiert. Ausgaben erscheinen somit zum gleichen Zeitpunkt wie die zugehörigen Eingaben. Zeit vergeht lediglich in stabilen Systemzuständen.

„Keine Zeit zu verbrauchen" darf hier aber keinesfalls missinterpretiert werden; es handelt sich dabei nur um eine Abstraktion der Realität, die annimmt, dass die tatsächliche Zeitspanne, die das später realisierte System verbrauchen wird, um seinen Zustand zu

verändern, kürzer sein wird, als die Zeit, die zwischen dem Eintreffen aufeinanderfolgender Signale der Sensorik vergeht. In der Praxis ist die Aussage „ohne Zeitverzögerung“ daher eher durch den Begriff der „Rechtzeitigkeit“ zu ersetzen. Berechnungen müssen somit abgeschlossen sein, bevor die nächste Berechnungsanforderung eintrifft. Die Zeitpunkte der Interaktionen müssen dabei nicht periodisch sein. Ein synchrones System kann sich synchron gegenüber seiner Umgebung verhalten, wenn es seine Berechnungen für eine Reaktion immer abgeschlossen hat, bevor neue Ereignisse eintreffen. Wenn dies sichergestellt werden kann, so ist gleichzeitig die Gültigkeit der Synchronitätshypothese gegeben oder anders ausgedrückt: Das komplexe Vibrationsmodell kann in das einfache Newtonmodell abstrahiert werden (Gunzert, 2003). Dazu müssen jedoch folgende Zeiten in Erfahrung gebracht werden:

1. Die minimale Zeit zwischen zwei Ereignissen (die aufeinander folgen können) sowie
2. die maximale Ausführungszeit der Reaktion.

Falls (2) größer als (1) ist, kann versucht werden durch Modifikation des Programms (z. B. Struktur, Übersetzungsoptionen usw.) oder der Laufzeitumgebung (Prozessor, Speicher usw.) die Bedingung für die Gültigkeit der Synchronitätshypothese trotzdem zu erfüllen (Gunzert, 2003).

Vorteile der Hypothese

Diese Annahme hat mehrere Vorteile. Die Reaktionszeiten sind in der Phase des Entwurfs noch unabhängig von der konkreten, ggf. verteilten Implementierung. Künstliche, vielleicht sogar falsche, zusätzliche Verzögerungszeiten werden nicht eingeplant; und schließlich kann auf diese Weise jede Reaktion in beliebig viele Sub-Reaktionen zerlegt werden, ohne das zeitliche Verhalten der Spezifikation zu beeinflussen. Bei den synchronen Sprachen wie etwa Esterel wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt.

Die perfekte Synchronie unterstellt also ein ideales reaktives System, bei dem jede Reaktion durch eine quasi unendlich schnelle Maschine ausgeführt wird. Zeit vergeht nur, wenn das System einen stabilen Zustand erreicht hat. Zustandsübergänge verbrauchen dagegen keine Zeit.

Das synchrone Modell macht es einfacher, korrekte Systeme zu entwickeln und aufzubauen (Gunzert, 2003). Es verbirgt zeitliche Details und vereinfacht die Synchronisierung von Teilen des Systems. Die Funktionsweise eines Systems ist einfacher zu spezifizieren und zu verstehen, da das Verhalten vereinfacht wird.

Darüber hinaus wird durch diese Technik weniger Kontrolle über das Verhalten von einzelnen Komponenten eines Systems benötigt. Ihre genaue Geschwindigkeit spielt keine Rolle solange sie über einer bestimmten Schwelle liegt.

Die Synchronitätshypothese vereinfacht das modellierte zeitliche Verhalten und ermöglicht den synchronen Sprachen präzise definierte Semantiken (Gunzert, 2003). Dadurch ist es möglich, sowohl ausführbaren Code aus den Spezifikationen zu generieren als auch formale Methoden zur Verifikation einzusetzen. Synchrone Sprachen sind somit gleichzeitig Spezifikations- und Programmiersprachen.

In der Realität hat man häufig genauere Vorstellungen, in welchen Zeitabständen externe Ereignisse kommen können (z. B. welche Datenrate das System verarbeiten können muss). Dies erlaubt eine bessere Optimierung der zu synthetisierenden Systemkomponenten. In so einem Fall bietet sich ein zeitbehaftetes Modell an.

Alternative Lösungsansätze zur Modellierung der Zeitspanne zwischen Ein- und Ausgabe

Bei der Mehrzahl von reaktiven Systemen, insbesondere bei Echtzeitsystemen, ist es andererseits für deren tatsächliche Implementierung sehr wichtig zu wissen, wie viel Zeit zwischen Ein- und Ausgabe vergeht (Huizing und Gerth, 1991). Auf der Ebene der Beschreibung solcher Systeme sind zunächst mehrere Ansätze zur Lösung dieser Problematik denkbar:

1. Zunächst könnte man für jeden Systemschritt die exakte Zeit angeben, die hierfür seitens der Implementierung nötig sein wird. So eine Abschätzung ist aber zu aufwändig und zwingt den Designer der Software schon zu Beginn seiner Entwicklungsarbeiten, alle Vorgänge zeitlich exakt zu quantifizieren. Wollen wir davon abstrahieren, so müssen alternative Lösungswege gefunden werden.
2. Eine erste Näherung wäre, jeder Systemreaktion eine konstante endliche Zeitdauer zuzuschreiben. Obwohl dieser Ansatz schon sehr viel einfacher als der erste ist, ist er immer noch nicht abstrakt genug und besitzt darüber hinaus weitere Nachteile: In der Praxis stellt die Angabe einer fixen Zeitvorgabe zur Reaktion in der Regel eine obere Grenze für die Reaktionszeit der Implementierung dar, was dazu führen kann, dass einzelne Vorgänge ggf. künstlich verzögert werden müssen. Darüber hinaus kann eine Reaktion mit einer festen oberen Zeitschranke nicht weiter durch eine Folge von (Teil-)Reaktionen verfeinert, also konkretisiert werden. Schrittweise Konkretisierungen von Systembeschreibungen sind aber ein probates Mittel zur

konstruktiven Gewährleistung einer hohen Softwarequalität und sind daher im Zuge einer methodisch ordentlichen und fehlerfreien Softwareentwicklung wünschenswert.

3. Eine nächste Lösungsalternative könnte darum sein, jeder Reaktion eine beliebige, unterschiedliche (also nichtkonstante) positive Zeitdauer zuzuordnen. Dieser Ansatz wäre mit Sicherheit abstrakt genug, weil Reaktionszeiten beliebig ausgetauscht und Reaktionen damit verfeinert werden können. Allerdings lässt er durch dieses hohe Maß an Abstraktion so viel Freiraum (genauer: Nicht-Determinismus), dass es nahezu unmöglich ist, interessante Systemeigenschaften in dieser Entwicklungsphase nachzuweisen.
4. Es bleibt also nur die Lösung, jeder Systemreaktion die Zeitdauer 0 Zeiteinheiten zuzuschreiben und damit die Annahme zu verbinden, die Dauer einer Systemreaktion des implementierten Systems sei stets schneller als die Rate der eingehenden Ereignisse oder Signale. Eine vergleichbare Annahme wird bei der Entwicklung von Mikrocomputern, also von Hardware, übrigens schon seit langem verwendet: Hier geht man stillschweigend davon aus, dass viele komplexe arithmetische und logische Operationen von der arithmetisch-logischen Einheit (ALU) der CPU (Central Processing Unit) ausgeführt werden können, bevor das nächste Mal ein Taktsignal, beispielsweise mit einer Frequenz von 4 GHz, erzeugt wird. In diesem Fall bleiben der ALU nur vier Nanosekunden zur Reaktion. Dass diese Zeitspanne immer eingehalten werden kann, wird von den Prozessorherstellern stets akribisch nachgewiesen, bevor ein neuer Prozessor auf dem Markt kommt.

Vorteile der perfekten Synchronie

Diesen letzten Ansatz bezeichnet man wie eingangs schon erwähnt als „perfekte Synchronie“. Er besitzt mehrere Vorteile:

- Reaktionszeiten sind bereits in der frühen und damit noch abstrakten Phase der Systementwicklung bekannt.
- Reaktionszeiten hängen nicht von der konkreten Implementierung ab.
- Reaktionszeiten sind so kurz wie möglich, da künstliche Verzögerungen zur Überbrückung von zeitlichen Obergrenzen nicht nötig sind.

- Jede Systemreaktion kann durch eine Folge von Reaktionen verfeinert, also konkretisiert werden, da die folgende zeitliche Gleichung immer gilt: 0 + 0 = 0.

Esterel

Esterel basiert auf der Hypothese, dass Signalaustausch und elementare Berechnungen keine Zeit benötigen, so dass Systemausgaben mit ihren Eingaben synchron ablaufen („perfect synchrony hypthesis"). Diese „perfekte Synchronisation" macht nebenläufiges, deterministisches Verhalten in Esterel möglich. Weitere Hauptmerkmale von Esterel sind umfangreiche Konstrukte zur Zeitdarstellung und die Möglichkeit zur Modularisierung.

4.5.3 Determinismus

Ein Programm in Esterel verarbeitet Ströme (potentiell unendlich lange Sequenzen) von Ereignissen. Ereignisse dienen zur internen und externen Kommunikation, wobei ein *Ereignis* aus mehreren Elementarereignissen (z. B. Signalen) bestehen kann, die nicht unterbrechbar sind. In Esterel wird auf jedes Eingabeereignis durch Ändern des internen Zustandes sofort reagiert und diese Reaktion läuft perfekt synchron mit der Eingabe ab. Die verarbeitende Einheit ist genügend schnell, so dass während der internen Berechnung keine neue Eingabe auftritt. Dieses interne Berechnungsintervall wird *„Moment"* oder *„Augenblick"* (engl. instant) genannt und ist unendlich kurz (engl. instantaneous).

Esterel nimmt an, dass reaktive Systeme deterministisch sind. Beim Nichtdeterminismus handelt es sich um ein ebenso vielschichtiges wie komplexes Themengebiet, dessen ausführliche Behandlung mit einem Anspruch auf eine nur annähernde Vollständigkeit den Rahmen dieses Dokuments bei weitem sprengen würde. Auf informeller Ebene können wir ein System dann als nichtdeterministisch betrachten, wenn es auf ein gegebenes Eingabeereignis (engl. input event) mehr als ein (eindeutiges) Ausgabeereignis (engl. output event) produzieren kann. Es wählt sozusagen eine von mehreren möglichen Systemantworten aus, ohne dass ein Beobachter des Systems von außen die Möglichkeit besäße, diese Antwort vorauszusagen.

Nehmen wir beispielsweise an, ein Aufzug befände sich im dritten Stockwerk eines Gebäudes und Personen im ersten und fünften Stock forderten ihn per Knopfdruck gleichzeitig an. Falls das Verhalten des für diese Funktionalität zuständigen Softwaremoduls

nichtdeterministisch beschrieben wäre, könnte sich der Aufzug für eine der beiden Möglichkeiten entscheiden (ohne dass vorher jemand wüsste wie). Wie man ebenfalls anhand dieses Beispiels sieht, hat Nichtdeterminismus auch nichts mit Wahrscheinlichkeitstheorie zu tun, sondern kann als allgemeinerer Begriff betrachtet werden.

Für Esterel gilt: Alle Anweisungen bzw. Konstrukte sind (vom Compiler) garantiert deterministisch. Diese Aussage muss im weiteren Verlauf nochmals genauer betrachtet werden: Mit Hilfe der Parallelkomposition zweier ursprünglich deterministischer Anweisungen kann der Entwickler ungewollt nichtdeterministisches Verhalten einführen. Um dies zu vermeiden, sind ausgefeilte Algorithmen erforderlich, die ein fertiges Programm auf Nichtdeterminismen untersuchen und nur deterministische Programme für die Übersetzung freigeben. Da es sich bei Esterel um eine Sprache handelt, die in der Regel direkt in einen ausführbaren Code übersetzt wird, ist diese Analyse unbedingt erforderlich. Schließlich kann auf der Ebene der realen Hardware Nichtdeterminismus nicht realisiert werden.

4.5.4 Allgemeines

Mit Hilfe von Esterel können im Wesentlichen Zustandsmaschinen (Mealy Maschinen, sequentielle Automaten) beschrieben werden, bei denen die Zustandswechsel durch externe Events (englisch für Ereignisse, daher der Name „ereignisbasierte Sprache“) ausgelöst werden und wiederum zum Aussenden von Events führen können. Diese Events am Eingang eines Modules werden *nicht* gespeichert, sondern führen zur sofortigen Aktivierung des Moduls zwecks eines Zustandswechsels. Danach sind alle Events am Eingang verschwunden, auch diejenigen, die im aktuellen Zustandswechsel nicht abgefragt wurden.

Esterel-Prinzip „Write Things Once“ (WTO)

Esterel verfolgt das *„Write Things Once“ (WTO) Prinzip:* Die Sprache erlaubt eine Wiederholung von Programmcode durch Verschachtelung syntaktischer Strukturen. Mealy Maschinen dagegen erlauben dies z. B. nicht und erfordern statt dessen eine Wiederholung von Codesequenzen. Das WTO Prinzip wird in den meisten „herkömmlichen“ Programmiersprachen ebenfalls verwendet, z. B. wird dort durch Schleifen vermieden, den Schleifenrumpf mehrfach zu schreiben. Es führt zu kompakten Programmen, die weniger fehlerträchtig und damit besser wartbar sind.

Orthogonalität

Ein zweites in Esterel umgesetztes Paradigma ist das der *Orthogonalität:* Jedes Programmkonstrukt kann beliebig in andere Konstrukte verschachtelt werden.

4.5.5 Parallelität

In Esterel gibt es einen Operator für die Parallelkomposition von Programmen: Seien P1 und P2 zwei Esterel-Programme, dann ist auch P1 || P2 ein Esterel-Programm und zwar mit der folgenden Charakteristik:

Eigenschaften

- Alle Eingaben, die von der Systemumgebung empfangen werden, sind gleichermaßen für P1 und P2 verfügbar.
- Alle Ausgaben, die von P1 (bzw. P2) generiert werden, sind im gleichen Augenblick (engl. „instantanteously", „in the same instant") für P2 (bzw. P1) sichtbar (= perfekte Synchronie).
- P1 und P2 werden nebenläufig ausgeführt und ihre Parallelkomposition P1 || P2 terminiert, wenn beide, P1 und P2, terminieren.
- P1 und P2 teilen sich keine gemeinsamen Variablen.

Broadcasting

Da es weder gemeinsame Variablen noch Konstrukte für den gezielten Austausch von Werten gibt, muss die Kommunikation zwischen P1 und P2 auf andere Weise erreicht werden. Es handelt sich hierbei um das Prinzip des „Broadcasting" (ein Sender, viele Empfänger; vgl. Funk und Fernsehen), bei dem aufgrund der Annahme der perfekten Synchronie übertragene Werte sofort den Empfängern zur Verfügung stehen. Man spricht hier darum auch vom „Instantaneous Broadcasting". Beim Broadcasting erfolgt die Kommunikation nicht gerichtet, d. h. ein dedizierter Empfänger kann nicht spezifiziert werden.

Ein Esterel-Programm besteht im Wesentlichen aus Deklarationen und Instruktionen.

4.5.6 Deklarationen

Esterel besitzt nur sehr wenige Datentypen und kann daher auch nicht als vollständige Programmiersprache bezeichnet werden.

Vielmehr müssen benötigte Datentypen, Konstanten, Signale sowie darauf aufbauende Funktionen und Prozeduren wie etwa das Lesen von Eingaben und die Produktion von Ausgaben in einer sogenannten „Host Language“, in diesem Fall die Sprache C oder Ada, implementiert und dann in Esterel importiert werden. In Esterel selbst wird lediglich die Prozesskontrolle programmiert. Fertige Esterel-Programme werden dann nach C zurückübersetzt und sind auf Mikrocontrollern usw. lauffähig.

Module

Ein Esterel-Programm kann aus einem oder mehreren *Modulen* bestehen. Ein Modul beginnt mit dem Schlüsselwort module, gefolgt vom Modulnamen und einem Doppelpunkt. Das Konstrukt wird mit dem Schlüsselwort „end module“ (in älteren Versionen lediglich durch einen Punkt) abgeschlossen. Dazwischen stehen der Deklarations- und der Instruktionsteil. Zeilenweise Kommentare beginnen mit „%“ und enden mit dem Zeilenabschluss:

```
module ErstesEsterelBeispiel :
       % Deklarationen
       % Instruktionen
end module
```

Basisdatentypen

Esterel kennt drei vordefinierte *Basisdatentypen,* nämlich „integer“, „boolean“ und „string“. Komplexere Datenstrukturen wie Arrays oder Records müssen in Esterel vom Benutzer deklariert und extern in C oder Ada definiert werden.

Konstanten

Konstanten werden in Esterel folgendermaßen deklariert:

```
constant MIN, MAX: integer;
```

Jeder Konstanten bzw. jeder Gruppe von Konstanten muss ein Typ zugewiesen werden. Im obigen Beispiel handelt es sich hierbei um den Basistyp „integer“. Die Initialisierung muss dann extern in der Hostsprache (C oder Ada) geschehen, da eine Zuweisung von Konstanten im Esterel-Modul nicht erlaubt ist.

Funktions-deklarationen

Beispiele für die *Funktionsdeklaration* in Esterel sind:

```
function TOP(): integer,
         FAKULTAET (integer): integer,
         MULT (float, float): float;
```

Wie man unschwer erkennen kann, können Funktionen keine, eine oder mehrere Argumente besitzen. Die Definition der Funktion findet dann in der Hostsprache statt. Die Funktionsdeklaration in

Esterel bestimmt den Typ und die Anzahl der Eingabeargumente sowie den Typ des Ergebnisses. Nach dem Funktionsnamen folgt die Liste der Argument-Typen, nach dem Doppelpunkt der Ergebnistyp. Eine Funktion liefert immer einen Wert als Ergebnis. Dabei ist die Angabe des Resultat-Typs obligatorisch, da Funktionen nicht überladen (engl. overloaded) werden dürfen.

Prozedur-deklarationen

Prozedurdeklarationen sind Funktionsdeklarationen sehr ähnlich. Allerdings finden sich in Prozedurdeklarationen statt Argument- und Resultat-Typen Listen mit Referenz- und Wertangaben. Eine Prozedur gibt kein Ergebnis zurück, sondern ändert einige der Referenzparameter, also Variablen, die durch die Prozedur gelesen und verändert werden können (engl. call-by-reference). Die Wertparameter (engl. call-by-value) können wie die Argumente der Funktion nur gelesen werden. Die Definition von Prozeduren findet dann wiederum in der Hostsprache statt.

Signale

Signale dienen der Kommunikation per Broadcasting, die als Input-, Output- oder Input-Output-Signale deklariert werden können. Input-Signale dürfen nur in das umschließende Modul eingelesen, Output-Signale nur vom Modul ausgegeben werden und Input-Output-Signale dürfen beides. Signale müssen in der Hostsprache durch Prozeduren definiert sein.

Reine Signale

Drei Arten von Signalen werden unterschieden. *Reine Signale* (engl. pure signals) übertragen keine Daten und tragen keinen Wert; sie dienen ausschließlich der Synchronisation und benötigen daher auch keine Angabe von Datentypen:

```
input A, B, C;
```

Einfache Signale

Einfache Signale (engl. simple signals) übermitteln einen Wert. Sie brauchen deshalb neben dem Signalnamen (wie etwa A, B oder C oben) zusätzlich eine Typangabe:

```
input A(integer), B(integer), C(integer);
output D(string);
inputoutput E(boolean);
```

Mehrfache Signale

Bei *mehrfachen Signalen* werden die Werte von simultanen Sendern durch eine sogenannte Kombinationsfunktion zusammengeschlossen, die binär, assoziativ und kommutativ sein muss. Nach dem Schlüsselwort „combine“ wird der Typ der Eingabesignale, nach dem Schlüsselwort „with“ der Name der Kombinationsfunktion angegeben. Ein Beispiel wird durch

```
output VALUE (combine FLOAT with FLOAT_ADD),
function FLOAT_ADD (FLOAT, FLOAT): FLOAT;
```

gezeigt. Obige Kombinationsfunktion „FLOAT_ADD“ muss, wie geschehen, als Funktion deklariert werden.

Signale besitzen zu jedem Zeitpunkt einen eindeutigen Status, nämlich „present“ oder „absent“, der zu jedem Zeitpunkt aufs Neue bestimmt werden muss, da er nicht gespeichert wird. Gespeichert wird lediglich ggf. der Signalwert.

Variablen

Variablen besitzen dagegen keinen Status, sondern nur einen Wert eines beliebigen Typs (s.o.), der gespeichert wird. Variablen können während eines Zeitpunkts ihren Wert mehrmals ändern.

Sensoren

Darüber hinaus kennt Esterel noch *Sensoren,* die vom zugehörigen Modul nicht ausgeschickt, sondern nur empfangen werden können. Die einzige Operation, die auf einem Sensor zulässig ist, ist die Abfrage seines Wertes durch den „?“-Operator. Da Sensoren im Folgenden eine untergeordnete Rolle spielen, wollen wir hier nicht im Detail auf sie eingehen.

4.5.7 Instruktionen

Im Instruktionsteil eines Esterel-Programmes werden Anweisungen durch folgende Konstrukte aufgebaut:

Konstrukte

- Deklaration von lokalen Signalen und Variablen
- elementare Anweisungen wie Zuweisungen und Prozeduraufrufe
- Verzweigungen und Schleifen
- Eingabe, Ausgabe und Testen von Signalen
- zeitliche Anweisungen
- Ausnahmebehandlung

Syntax der Anweisungen

Gültige *Anweisungen* in Esterel sind:

- **nothing** (leere Anweisung)
- **pause** (Zeitverbrauch)
- **emit** x(τ) (Signalemission)
- x := τ (Variablenzuweisung)
- **present** σ **then** S1 **else** S2 end (Fallunterscheidung)
- S1 || S2 (Parallelkomposition)
- S1; S2 (Sequentielle Komp.)

- **loop** S **end** (Schleife)
- **trap** t in S **end** (async. Prozessabbruch)
- **exit** t (async. Prozessabbruch)
- **abort** S **when** σ (synchr. Prozessabbruch)
- **suspend** S **when** σ (Prozesssuspension)
- **signal** x : α in S **end** (lokale Signale)
- **var** x : α **in** S **end** (lokale Variable)

Hierbei bezeichnet σ ein Signal, τ eine gültige Typangabe und α eine beliebige Anweisung. Durch die Klammerung von Anweisung mittels eckiger Klammern kann die Bearbeitungsreihenfolge modifiziert werden.

Auf informeller Ebene kann diesen Basis-Anweisungen folgende Bedeutung zugewiesen werden (wir verzichten hier auf das Studium der formalen Semantik von Esterel):

Informelle Semantik der Anweisungen

- **nothing** terminiert sofort und emittiert nichts; es ist notwendig, da es keine leere Instruktion in Esterel gibt
- **pause** verbraucht genau eine Zeiteinheit; emittiert nichts
- **halt** verbraucht die restliche Zeit; emittiert nichts
- **emit** x(τ) setzt den Status von x auf „present“ und ändert evtl. den Wert von x auf τ (ohne Zeitverbrauch)
- x := τ ändert den Wert von x auf τ (ohne Zeitverbrauch)
- **present** σ **then** S1 **else** S2 end testet auf Präsenz des Signals σ. Im positiven Fall wird sofort S1, sonst S2 ausgeführt. Insgesamt besitzt die Anweisung ein Verhalten wie S1 *oder* S2.
- S1; S2 führt S1 aus und startet sofort nach dessen Terminierung S2.
- S1 ‖ S2 startet die synchrone nebenläufige Ausführung von S1 und S2 und terminiert, wenn S1 *und* S2 terminieren.
- **loop** S **end** startet S und startet S erneut, falls S terminiert (S muss hier Zeit > 0 verbrauchen, ansonsten ergibt das Konstrukt eine Endlosschleife in Nullzeit).
- **abort** S **when** σ startet S unabhängig von der initialen Präsenz von σ. Falls während der Ausführung von S σ präsent wird, wird S abgebrochen und Signale in S zu diesem Zeitpunkt nicht mehr emittiert (starker Abbruch, engl. strong preemption).

- **suspend** S **when** σ startet S unabhängig von der initialen Präsenz von σ. Falls während der Ausführung von S σ gilt, wird S unterbrochen, wenn σ nicht mehr präsent ist, wird die Ausführung von S fortgesetzt, Signale in S zu diesem Zeitpunkt aber nicht mehr emittiert (vgl. **abort**).

Durch die Verschachtelung verschiedener „abort"-Anweisungen können nebenläufige Kontrollfäden (engl. control threads) mit statischen Prioritäten versehen werden.

Neben den oben genannten Basis-Anweisungen gibt es in Esterel erweiterte Anweisungen wie etwa „halt", „await" und „sustain", die aber durch Kombination von Basiskonstrukten auf rein syntaktischer Ebene erzeugt werden können und darum als „syntaktischer Zucker", also als syntaktischer Abkürzungsmechanismus betrachtet werden können. Wir gehen auf sie daher im Folgenden nicht näher ein.

4.5.8 Beispiel: Die sogenannte ABRO-Spezifikation

Eine Ausgabe O soll immer dann erfolgen, wenn zwei Eingaben A und B eingetroffen sind. Dieses Verhalten soll mit dem Reset-Signal R zurückgesetzt werden. Der vergleichsweise simple Esterel-Code hierfür lautet:

ABRO-Beispiel

```
module ABRO:
       input A, B, R;
       output O;
       loop
              [ await A || await B ];
              emit O
       each R
end module
```

4.5.9 Semantik

Die formale Semantik zur Verhaltensbeschreibung von Esterel-Programmen von Berry (Berry, 1998) definiert die Sprache durch *Zustandsübergänge*. Eine Erweiterung ist die *Ausführungssemantik*, ebenfalls von Berry (Berry, 1998), die durch eine terminierende Folge von elementaren Mikroschritten angegeben wird. Auf dieser

Ausführungssemantik beruht auch der Compiler von *Esterel V3* (Version 3). Dadurch wird eine Spezifikation zuerst in endliche Zustandsautomaten und anschließend in eine klassische Sprache wie C oder Ada übersetzt. Der *Esterel V4* Compiler basiert auf der sogenannten *elektrischen Semantik* (Berry, 1998), die ein Programm erst in Schaltbilder mit Verbindungen und logischen Ebenen übersetzt. Anschließend kann das Programm. in einer Standardsprache oder als endlicher Automat ausgegeben werden. Ausführungssemantik und elektrische Semantik unterscheiden sich v. a. in ihrem Umgang mit Kausalitätsproblemen.

4.5.10 Kausalitätsprobleme

Dynamische Analyse

Durch die perfekte Synchronie können sogenannte Kausalitätsprobleme entstehen, d. h. dass Signale wie etwa in „present S then emit S end“ zyklisch voneinander abhängen. In einer *dynamischen Analyse* werden alle möglichen Folgeanweisungen berechnet. Dadurch wird festgestellt, ob es zu einem Deadlock (Verklemmung) kommt. Dabei werden auch Relationen zwischen Eingabesignalen berücksichtigt. Der Compiler V3 arbeitet nach diesem Modell.

Statische Analyse

Im Compiler V4 wird dagegen eine *statische Analyse* durchgeführt; dabei wird die Korrektheit unabhängig von möglichen Folgeanweisungen unter der Annahme bestimmter Bedingungen geprüft. Diese statische Überprüfung ist zwar schneller als die dynamische, kann aber auch dazu führen, dass ein eigentlich korrektes, will heißen deterministisches, Programm abgelehnt wird. Die elektrische Semantik übersetzt ein Esterel-Programm auf Gatterebenen in digitale Schaltbilder (Gatter) mit Verbindungen. Da Gatter durch boolesche Ausdrücke und Schaltnetze durch boolesche Gleichungen definiert sind, können in periodischen Abständen diese booleschen Gleichungen, die auch sämtliche Eingaben umfassen, gelöst und der neue Wert jeder elektrischen Verbindung berechnet werden. Dieses Vorgehen beruht auf der „Haltemengen-Theorie“. Dabei wird an jeder Stelle im Programm, an der auf weitere Ereignisse gewartet wird, die Anweisung „halt“ als Kontrollpunkt eingeführt. Dadurch kann ein Zustand durch die Menge seiner Kontrollpositionen, also Haltemengen, angegeben werden, was einen deterministischen Code zur Folge hat.

4.5.10.1 Logische Korrektheit

Fester Bestandteil aller Semantiken ist die logische Korrektheit. Ein logisch korrektes Esterel-Programm ist für jedes mögliche Eingabesignal reaktiv und deterministisch. Mit den gegebenen syntaktischen Konstrukten und dem Prinzip des Broadcastings ist es aber leicht möglich, syntaktisch korrekte, aber semantisch sinnlose Programme zu schreiben, die nicht reaktiv und/oder nicht deterministisch sind. Solche Programme sollen vom Compiler abgelehnt werden. Die logische Korrektheit kann durch umfangreiche Fallunterscheidungen der möglichen Eingabesignale geprüft werden.

Um Fallunterscheidungen der möglichen Eingabesignale durchzuführen, müssen wir zuerst festlegen, welchen Status Signale standardmäßig haben. Der Default-Status wurde auf „absent“ festgelegt. Wann ein Signal „present“ sein kann, wird formal durch das *Kohärenz-Prinzip* ausgedrückt.

Definition (Kohärenz)

Definition (Kohärenz):
Ein Ausgangssignal bzw. ein lokales Signal x ist zu einem Zeitpunkt präsent genau dann, wenn zu diesem Zeitpunkt die Anweisung „emit x“ im Sichtbarkeitsbereich von x ausgeführt wird.

Das Kohärenzprinzip verbietet also die Annahme, dass ein Signal „present“ ist, wenn zu diesem Zeitpunkt keine Emission dieses Signals stattfindet. Im Allgemeinen kann der Status eines Signals selbstverständlich vom Status eines anderen Signals abhängen. Beispielsweise hängt in „present y then emit x end“ x kausal von y ab. Die Kausalitätsrelation kann bei synchronen Sprachen durch Einsatz des Broadcasting Mechanismus Zyklen aufweisen, so z. B. in

```
signal x in
     present x then emit x end
end signal
```

Alternativen zur Behandlung solcher Kausalitätszyklen sind:

- *Alle zulassen:* Da Deadlock und Nichtdeterminismus ähnlich wie bei Statecharts möglich sind, wird diese Variante bei Esterel nicht verwendet.
- *Alle ablehnen, nur azyklische Programme zulassen:* Stellt eine zu starke Einschränkung dar. Es würden viele logisch korrekte Programme abgelehnt werden.

- *Zulassen, wenn Programm logisch korrekt:* Wird bei der logische Transitionssemantik in Esterel verwendet. Bei der Analyse des Verhaltens der Programme müssen die Status von Signalen zuerst angenommen und dann im weiteren Verlauf geprüft werden, ob die getroffenen Annahmen richtig waren. Die Prüfung der logischen Korrektheit ist algorithmisch jedoch sehr aufwändig.
- *Zulassen, wenn Programm logisch korrekt und dies einfach zu prüfen ist:* Wird bei der konstruktiven Semantik und damit in der Praxis verwendet.

Beispiele: Esterel

Im Folgenden werden zwei Beispiele logisch inkorrekter Esterel-Programme angegeben, bevor der Abschnitt mit einem korrekten Programm schließt.

Beispiel 1:
Nachstehender Programmtext stellt ein nichtreaktives Esterel-Programm dar, da keine logische Annahme über die Reaktion dieses Programmes gemacht werden kann. Die Annahme o sei „absent" bestätigt sich nicht, da dann im Else-Zweig eine Emission stattfindet (Verletzung des Kohärenzprinzips). Die Annhame o sei „present" bestätigt sich ebenfalls nicht, da dann keine Emission stattfindet (Verletzung des Kohärenzprinzips).

```
module P1:
  output o;
  present o else emit o end
end module
```

Beispiel 2:
Im Folgenden handelt es sich um ein nichtdeterministisches Esterel-Programm, weil zwei gültige Annahmen über die Reaktion dieses Programms existieren. Die Annahme o sei „absent" bestätigt sich, da dann keine Emission stattfindet. Die Annahme o sei „present" bestätigt sich gleichermaßen nicht, da dann eine Emission stattfindet.

```
module P2:
  output O;
  present O then emit O end
end module
```

Beispiel 3:
Das folgende Beispiel zeigt ein Esterel-Programm, das möglicherweise für den Leser auf den ersten Blick etwas sonderbar anmuten mag, tatsächlich aber logisch korrekt ist:

```
module P3:
  output o1, o2;
  present o1 then emit o1 end
  ||
  present o1 then
    present o2 else emit o2 end
  end
end module
```

Der erste Zweig ist die Kopie unseres nichtdeterministischen Beispiels 2. Der zweite Zweig ist die Kopie unseres nichtreaktiven Beispiels 1 eingeschlossen in ein „present“ Statement. Überraschenderweise ist dieses Programm reaktiv und deterministisch. Die einzige gültige Annahme ist o1 absent und o2 absent, welche sich dann bestätigt. Alle anderen Annahmen verletzten das Kohärenzprinzip.

4.5.10.2 Konstruktive Semantik

Beispiel 3 aus dem letzen Abschnitt, ein sonderbares, aber logisch korrektes Programm zeigt, dass eine unnatürliche Informationsverarbeitung in logisch korrekten Programmen möglich ist:

- Es werden Annahmen getroffen, die sich später selbst bestätigen müssen.
- Die logische Korrektheit ist in diesem Beispiel nur gegeben, weil keine andere Annahme sich bestätigt.

Bewertung der logischen Korrektheit:
Aus folgenden Gründen eignet sich die logische Korrektheit nicht als Basis für die Sprache:

- Die Prüfung eines Esterel-Programmes auf Reaktivität und Determinismus braucht exponentielle Zeit (NP-vollständiges Problem).
- Der Ansatz der logischen Korrektheit ist intuitiv für den Programmierer nicht nachvollziehbar, da der Kontrollfluss nicht

dem Zeitfluss entspricht. Da aber Esterel als eine imperative Sprache konzipiert wurde, soll auch in „present s then p end“ der Status von s nicht davon abhängen, was in p gemacht wird. Oder in anderen Worten: Bei der Ausführung dieses Statements möchte der Programmierer, dass zuerst a ausgewertet werden soll und dann in Abhängigkeit davon p. In „present s then p end“ wird durch „then“ eine kausale Reihenfolge ausgedrückt.

Dieser Missstand wird bei der *konstruktiven Semantik* ausgeschlossen. Sie berücksichtigt die Kausalität der Informationsverarbeitung, also einen gerichteten Informations- und Auswertungsfluss und lässt sich mit polynomiellen Aufwand effizient nachweisen. Dabei wird jedoch das Zurückweisen einiger logisch korrekter Programme in Kauf genommen.

Bei der konstruktiven Semantik werden keine Annahmen über den Status von Signalen gemacht, sondern das Programm und der Status der Signale werden einer Analyse unterzogen. Dabei wird anhand des bisherigen Programmablaufs berechnet ob ein Signal gesetzt werden muss oder nicht gesetzt werden kann. Aus diesen Informationen wird der weitere Programmablauf berechnet. Ein Programm wird abgelehnt, falls eine weitere Analyse aufgrund unbekannter Zustände von Signalen nicht fortgesetzt werden kann. Für weitere Informationen wird der Buchentwurf von Gérard Berry mit dem Titel „The Constructive Semantics of Pure Esterel“, im Internet unter www-sop.inria.fr/esterel.org/home.htm beziehbar (Stand: Dezember 2004), empfohlen.

4.5.11 Codegenerierung und Werkzeuge

Bei der Codegenerierung wird die gesamte Kommunikation zwischen einzelnen Prozessen oder Modulen vom Compiler aufgelöst. Das bedeutet, dass ein paralleles, aus vielen Esterel-Modulen bestehendes Programm in ein sequenzielles Programm umgewandelt wird. Die Verwaltung von Prozessen, Scheduling und Interprozesskommunikation wird zur Übersetzungszeit durchgeführt. Folglich sind dafür keine Aktionen zur Laufzeit mehr notwendig. Es kann zum großen Teil auf Echtzeitbetriebssysteme verzichtet werden, was wiederum zu einer Speicherplatzersparnis führt. Folgende Optionen stehen bei der Codegenerierung zur Verfügung:

Optionen bei der Code-generierung

- Erzeugung von *Automatencode* (explizite Automaten-Darstellung): Beim Automatencode werden alle Zustände und Transitionen des Zustandsautomaten explizit numeriert. Dieser Code ist sehr schnell, kann jedoch bei komplexen Systemen sehr groß werden. Im ungünstigsten Fall besteht ein exponentielles Verhältnis zur Spezifikationsgröße (Zustandsexplosion). In der Praxis stellt sich jedoch heraus, dass die generierten Automaten in vielen Fällen bereits minimal sind.
- Erzeugung von *Gleichungscode* (implizite Automaten-Darstellung): Der Gleichungscode ist eine implizite Darstellung eines Zustandsautomaten durch einen Satz boolescher Gleichungen. Hier wird die Reaktion auf Ereignisse erst zur Laufzeit durch Lösen des Gleichungssystems berechnet. Die Größe des Gleichungscodes wächst nur linear mit der Größe der Spezifikation. Die Ausführungszeiten sind dafür etwas länger.

Zielsprachen

Folgende Zielsprachen stehen zur Verfügung:

- C
- C++
- (Structural) VHDL – nur mit Compiler V7
- (Structural) Verilog – nur mit Compiler V7

Die letzten beiden Übersetzungsmöglichkeiten ermöglichen *Hardware/Software Codesign* (siehe Abschnitt 5.6) mit Esterel.

Werkzeug Esterel Studio

Esterel Studio ist ein kommerziell vertriebenes Tool, das von der graphischen Spezifikation über Simulation bis zur Verifikation von Programmen alles für eine professionelle Soft-/Hardware Entwicklung in Esterel anbietet. Esterel Technologies entwickelte die zurzeit (Dezember 2004) öffentlich verfügbare V5.91 Spezifikation von Esterel weiter und ergänzte sie um datenverarbeitende Konstrukte. Bei Esterel Studio bekommt man die Auswahl mit welchem Compiler man arbeiten möchte, mit dem V5.91 Compiler oder der Inhouse-Weiterentwicklung, dem V7 Compiler. Praktische Erfahrungen mit Studentengruppen (in 2004 und 2005) weisen darauf hin, dass das Tool noch etwas instabil ist. Dennoch kann nach Rücksprache mit den Entwicklern davon ausgegangen werden, dass dies in Kürze abgestellt sein wird.

Weiterführende Literatur

Im Internet finden sich ferner unter www.esterel-technologies.com weitere Informationen sowie freie Software (z. B. Compiler), Demos und White Papers zu Esterel.

4.6 Synchrone Datenflusssprachen am Beispiel von Lustre

Charakteristik

Lustre ist eine deklarative, synchrone, datenflussorientierte Sprache zur Entwicklung reaktiver Systeme. Es handelt sich um eine deklarative Sprache, da jeder Ausdruck in einem Programm stets durch ein Gleichungssystem repräsentiert ist, das durch Programmvariablen beschrieben wird. Dieser Ansatz wurde von technischen Formalismen, wie Differentialgleichungssystemen oder synchronen Anwendernetzen (Blockdiagrammen) inspiriert. Programme in Lustre können sowohl auf rein textueller Ebene als auch graphisch entwickelt werden. In Lustre repräsentiert jede Variable und jeder Ausdruck einen Datenfluss.

Historie

Die Entwicklung von Lustre begann im Jahr 1984. Seitdem wurde Lustre besonders auf den Gebieten der Compilation, Verifikation und des automatischen Testens kontinuierlich weiterentwickelt (siehe www-verimag.imag.fr).

Datenfluss

Im Gegensatz zu der imperativen Sprache Esterel handelt es sich bei Lustre und auch Signal um Datenfluss-orientierte Sprachen. Hierbei beschränken sich diese Sprachen auf jene Datenflusssysteme, welche mit fest definierter, d. h. limitierter Speichergröße implementiert werden können. In Lustre können alle syntaktischen, textbasierten Sprachmittel auch in Form graphischer Datenflussdiagramme repräsentiert werden. Lustre wird auf diese Weise zu einer für Softwareingenieure leicht verwendbaren Sprache und rückt damit in deren Interessensbereich. Da ihre Semantik im Wesentlichen auf temporaler Logik (siehe Abschnitt 6.4.3) basiert, bietet sie andererseits eine formale Basis, die für automatische Codegenerierung und Sicherung der Softwarequalität (Verifikation) genutzt werden kann. Vorgenannte Kausalitätsprobleme, wie sie etwa in Esterel auftauchen, würden in Lustre als zyklische Datenflussdefinitionen modelliert. In diesen Fällen würde eine (Datenfluss-) Variable von sich selbst abhängig sein.

Aufbau

Lustre stellt Datenoperatoren, beispielsweise Addition, Subtraktion oder Multiplikation zur Verfügung. Diese Operatoren arbeiten auf Operanden, welche alle mit einem gemeinsamen Takt (engl. clock) durch das Datenflussnetzwerk „gepumpt“ werden – kurzum: Alle Operanden besitzen den selben Takt.

4.6.1 Datenfluss und Clocks

Basic Clock

Jeder Datenfluss in Lustre wird durch eine unendliche Sequenz von Werten eines bestimmten Typs repräsentiert, welche durch eine Clock in Sequenzschritte unterteilt wird. Dadurch entsteht eine Taktung des Datenflusses. Jedes (Unter-)Programm besitzt ein zyklisches Verhalten. Ein derartiger Zyklus definiert eine Sequenz von Zeitpunkten, die sogenannte „Basic Clock“ (BC), in denen das Programm aktiv ist. Nur zu diesen Zeitpunkten finden Aktionen statt. Die Länge der Zeitabstände zwischen den Aktionen wird durch die Basic Clock festgelegt. Somit ergibt sich eine Sequenz von Zeitpunkten an denen Eingabedaten aus dem Eingabedatenfluss bearbeitet werden.

Die Basic Clock ist die feinste (diskrete) Zeitrasterung des Systems. Teilprogramme können ihre eigenen Clocks besitzen, allerdings stets nur mit gröberen Zeitrasterungen, also größeren diskreten Zeitintervallen als die Basic Clock. Unterprogramme werden in Lustre als sogenannte „Nodes“ bezeichnet. Insgesamt kann es in einem Lustre-Programm darum mehr als einen einzigen Takt geben. Alle Takte können aber stets auf die Basis Clock herunter „gesampelt“ (engl. down sampled) werden. Dadurch entstehen andere, langsamere Clocks. Im Programm geschieht dies durch Kombination der Basic Clock mit einem booleschen Datenfluss (BDF), welcher den gewünschten Sample definiert; vgl. Tabelle 4.1.

Tab. 4.1: Entstehung einer neuen Clock aus der Basic Clock und einem boolschen Datenfluss

BC	1	2	3	4	5	6
BDF	true	false	true	true	false	true
Ergebnis	1		2	3		4

Wie die Tabelle 4.1 zeigt, muss dieses Clock-Konzept nicht an die physikalische Zeit gebunden sein; Intervallschritte einer Clock müssen folglich nicht äquidistant sein. Soll ein physikalischer Zeitbegriff im Programm Einzug halten, kann als Eingabe ein boolescher Datenfluss benutzt werden, der beim Auftreten eines beispielsweise „millisecond“ Signals den Wert true liefert (Halbwachs, 1991).

4.6.2 Variablen, Konstanten und Gleichungen

Variablen

Eine Variable besteht aus einer unendlichen Sequenz von Werten eines Typs. Dabei werden von Lustre die Basistypen „boolean“, „int“ (Integer), „real“ für Gleitkommazahlen und ein Tupelkonstruktor für karthesische Produkte unterstützt. Komplexe Datentypen können bei Bedarf ähnlich wie in Esterel von anderen Hostsprachen importiert werden und als abstrakte Typen benutzt werden.

Gleichungen

Eingabevariablen werden bei der Schnittstellendeklaration mit ihrem Typ deklariert. Alle anderen Variablen dürfen nur einmal in Form einer Gleichung definiert werden. Die Gleichung „X = E;“ (E sei hier ein beliebiger Ausdruck) definiert X als identisch zu E. Dies hat zur Konsequenz, dass X dieselbe Sequenz von Werten, den gleichen Typ und die gleiche Clock wie E besitzt. Die Variable kann nach ihrer Definition nur noch gelesen, jedoch nicht mehr verändert werden. Dadurch wird ein wichtiges Prinzip der Sprache realisiert, das Substitutionsprinzip (X kann überall im Programm durch E ersetzt werden und umgekehrt). Folglich können die Gleichungen auch in beliebiger Reihenfolge geschrieben werden, ohne das Programmverhalten zu beeinflussen.

Konstanten

Konstanten können sowohl aus den Lustre-eigenen, als auch den portierten Datentypen bestehen. Ihre Sequenz von Werten ist dann konstant; ihnen liegt die Basic Clock zu Grunde. In Lustre gibt es zwei Arten von Operatoren: Datenoperatoren und temporale Operatoren.

4.6.3 Operatoren und Programmstruktur

Daten-operatoren

Lustre stellt nachstehende *Datenoperatoren* zur Verfügung:

- *arithmetische:* +, -, *, /, div, mod
- *boolsche:* and, or, not
- *relationale:* =, <, <=, >, >=
- *konditionale:* if-then-else
- *importierte Funktionen*

Diese Operatoren dürfen aber nur bei solchen Variablen angewendet werden, die dieselbe Clock besitzen.

Neben den vorgenannten Datenoperatoren stellt die Sprache vier *temporale Operatoren* zur Verfügung:

Temporale Operatoren

- **Pre** (Previous): Mit dem „Previous"-Operator kann auf den Wert einer Variablen zur Zeit der vorherigen Clock zugegriffen werden. Dieser Operator liefert den Wert eines Ausdrucks, den er bei der vorherigen Clock hatte. Sei E ein Ausdruck mit (e_1, e_2, ..., e_n, ...) als Sequenz von Werten, so liefert pre(E) die Sequenz (nil, e_1, e_1,.., e_{n-1},..).
  Vorsicht: Bei der Verwendung des pre-Operators entsteht für den ersten Wert der Sequenz ein undefinierter Ausdruck, kurz „nil" (als Kurzform für „not in list").
- **Fby** bzw. **->** (Followed by): Der „Followed by"-Operator dient zur Initialisierung von Variablen, um den Wert „nil" zu vermeiden, der beim Zugriff auf den Vorgänger der ersten Variablen einer Sequenz vorliegt (vgl. „Pre"). Seien E und F Ausdrücke mit der Wertesequenz (e_1, e_2, ..., e_n, ...) und (f_1, f_2, ..., f_n, ...), dann ergibt E->F die Wertesequenz (e_1, f_2, f_3, ..., f_n, ...). Mit anderen Worten setzt sich E->F aus dem ersten Wert von E gefolgt vom zweiten bis n-ten Wert von F zusammen.
- **When** (Sample): Dieser Operator passt einen Ausdruck an eine langsamere Clock an. Seien E ein Ausdruck und B ein boolscher Ausdruck mit derselben Clock, dann ist „E when B" eine Sequenz von Werten aus E an der Stelle, an der B true war mit der Clock, die durch die true werte von B bestimmt wird.
  Vorsicht: Alle Werte von E an der Stelle an der B false ist, werden weggeworfen. Benutzen Sie diesen Operator nur, wenn sichergestellt ist, dass keine wichtigen Werte verloren gehen.
- **Current** (Interpolation): Der „Current"-Operator ist das Pendant zum „When"-Operator und besitzt die entgegengesetzte Wirkung. Er passt einen Ausdruck mit einer langsameren Clock an eine schnellere an. Dies wird durch Interpolation realisiert. Sei E ein Ausdruck mit einer langsameren Clock und B ein boolscher Ausdruck der die schnellere Clock darstellt, dann hat current E dieselbe clock wie B. Die „leeren" Werte in der Sequenz werden mit dem Wert der vorherigen Clock aufgefüllt.

Während die ersten beiden temporalen Operatoren auf (Takt-) synchronen Datenflusselementen operieren, welche den gleichen Takt besitzen, werden die letzten beiden verwendet, um Datenflüsse unterschiedlicher Taktfrequenzen anzupassen.

Beispiel: Lustre

Beispiel (Lustre):

```
Node Zaehler (a, b: int; reset: bool)
returns (c: int);
Let
c = a -> if reset then a
         else pre(c) + b;
Tel
```

Zur Strukturierung des Programms dienen Knoten (engl. nodes). Sie sind Unterprogramme, welche die Funktionalität kapseln und wiederverwendbar gestalten. Eine Knotendeklaration besteht aus einem Knotennamen und einem Eingabe- und Ausgabeparameter mit ihren Datentypen. Im Knotenrumpf befinden sich die lokalen Variablendeklarationen, die Gleichungen und die Assertions (Halbwachs, 1991).

4.6.4 Assertions (Zusicherungen)

Um den Compiler bei der Codeoptimierung zu unterstützen, besteht die Möglichkeit, Assertions anzugeben. Durch sie kann dem Compiler mitgeteilt werden, dass bestimmte Inputereignisse nicht vorkommen, beispielsweise dass zwei Variablen nie den gleichen Wert besitzen können oder eine Variable nicht zweimal hintereinander den gleichen Wert annehmen kann. Assertions sind boolsche Ausdrücke, die immer wahr sind.

Sollte es z. B. der Fall sein, dass zwei Eingabevariablen A und B niemals gleich sein sollen, so schreibt man:

```
Assert not (A and B);
```

Assertions nehmen bei der Verifikation von Lustre-Programmen eine zentrale Rolle ein (Halbwachs, 1991).

4.6.5 Compilation

Überprüfungen

Neben den üblichen Überprüfungen eines Compilers muss der Lustre-Compiler zusätzlich folgende Eigenschaften überprüfen:

- Ist jede lokale Variable sowie jede Outputvariable durch genau eine Gleichung definiert? (Gebot)
- Kommen rekursive Knotenaufrufe vor? (Verbot)
- Werden alle Operationen nur auf Operanden mit erlaubten Clocks angewandt? (Gebot)
- Arbeitet jeder Datenoperator mit mehr als einem Operanden auf Operanden mit gleicher Clock? (Gebot)
- Werden uninitialisierte Ausdrücke verwendet? (Verbot) Es darf bei Clocks, Ausgabevariablen und Assertions keine undefinierten Werte geben, wie sie bei der Benutzung des „Pre"-Operators entstehen
- Kommen im Code zyklische Definitionen (Deadlocks) vor? (Verbot) Jeder Zyklus sollte mindestens einen „Pre"-Operator enthalten.

Die Vermeidung von Deadlocks verdient besondere Aufmerksamkeit, die an dieser Stelle durch ein Beispiel untermauert werden soll:

```
X = X+1;                  // nicht erlaubt
X = pre(X)+1;             // erlaubt
```

Der Compiler lässt keine strukturellen Deadlocks zu, auch wenn diese wie im nachstehenden Beispiel nie eintreten können:

```
X = if C then Y else Z;
Y = if C then Z else X;
```

Nach diesen Überprüfungen erzeugt der Compiler einen erweiterten endlichen Automaten. Dieser liegt dann, wie in Esterel in Form einer Datei im sogenannten OC-Format vor. Mit entsprechenden Codegeneratoren kann dann Code in Sprachen wie C, ADA oder Le-Lisp erzeugt werden.

Werkzeuge

Für Lustre stehen mehrere kostenlose Tools zur Verfügung, die zur Spezifikation, Modellierung und Verifikation benutzt werden können. Ihre kommerzielle Verwendung ist jedoch ausgeschlossen. Für kommerzielle Zwecke kann beispielsweise das Softwarewerkzeug „Scade" käuflich erworben werden. Scade (früher SAO+/SAGA), das von Esterel Technologies (www.esterel-technologies.com) entwickelt wurde, ermöglicht den graphischen Entwurf eines datenflussorientierten, reaktiven Systems und stellt die hierfür notwendigen Tools zur Verfügung.

4.6.6 Verifikation und automatisches Testen

Formale Verifikation

Bei sicherheitskritischen Systemen ist es besonders wichtig, dass ein Programm verifiziert werden kann. Durch die *formale Verifikation* kann gezeigt werden, dass bestimmte Zustände nie eintreten bzw. das Programm korrekt arbeitet. Zur Verifikation eines Programms werden in Lustre sogenannte synchrone Beobachter eingesetzt. Synchrone Beobachter sind Assertions, die die Eigenschaften des Systems beschreiben. Diese werden wie das Programm auch in der Sprache Lustre implementiert. Nachdem alle synchronen Beobachter in das Programm eingebunden sind, kann mit dem Verifikationstool „Lesar“ (Model Checker, vgl. Abschnitt 6.4.3) geprüft werden, ob die Spezifikation erfüllt wird. Ist dies nicht der Fall, wird ein Gegenbeispiel erzeugt.

Automatisches Testen

Der *automatische Test* ist gewissermaßen das Gegenteil der Verifikation. Es wird hierbei nicht versucht nachzuweisen, dass das Programm den Spezifikationen entspricht wie es bei der Verifikation gemacht wird, sondern es wird versucht ein Szenario zu finden das die Spezifikation verletzt. Dazu wird ein Testdurchlauf generiert, der aus zwei Schritten besteht:

- Erzeugen einer zufälligen Eingabe, die der Systemumgebung entspricht.
- Überprüfung ob die Spezifikation mit dieser Eingabe eingehalten wird.

Lurette

Dieser Testdurchlauf wird solange wiederholt, bis ein Fall gefunden wird, der die Spezifikation verletzt oder ausreichend Durchläufe gemacht sind, so dass die Korrektheit angenommen werden kann. Solche automatischen Tests können mit dem Tool „Lurette“ durchgeführt werden.

Vergleich Esterel, Lustre

Esterel und Lustre sind nicht für alle Problembereiche gleichermaßen einsetzbar. Bei einigen eignet sich der imperative Ansatz von Esterel besser, bei anderen jedoch bietet der Datenfluss-Ansatz von Lustre Vorteile. Da beide Sprachen von mehreren Softwarewerkzeugen zusammen unterstützt werden, ist es möglich, durch gemischsprachliche Programme die Vorteile beider zu vereinen. Dabei sind allerdings ggf. semantische Unterschiede beider Sprachen zu bedenken: Lustre bietet u. a. auch eine Semantik an, welche die perfekte Synchronie, anders als Esterel, nicht zwingend voraussetzt (Bergerand, 1986).

4.6.7 Lustre im Vergleich zu Signal

Auch Signal ist eine Datenfluss-orientierte, synchrone Sprache. Wie bei Lustre ist die Angabe mehrerer Clocks zur Taktung der Datenflüsse möglich. Im Gegensatz zu Lustre besitzen Signal-Programme allerdings nicht nur eine, sondern mehrere Basic Clocks. Anders als bei Lustre ist es somit nicht möglich, Taktintervalle in kleinere Intervalle aufzuteilen. Abgesehen von zu Lustre vergleichbaren Datenoperatoren stellt Signal drei temporale Operatoren zur Verfügung: Einen Verzögerungsoperator, einen „Sample"-Operator sowie einen deterministischen „Merge"-Operator zum Vereinigen zweier Clocks.

Obwohl es sich bei Lustre und Signal um zwei deklarative Sprachen handelt, unterscheiden sie sich doch erheblich. Lustre ist eine *funktionale Sprache,* wobei jeder Operator eine Funktion repräsentiert, welche Eingabesequenzen in Ausgabesequenzen transformiert. Signal dagegen ist eine relationale Sprache. Ein Signal-Programm definiert eine *Relation* zwischen Ein- und Ausgabeflüssen, wodurch – anders als bei Funktionen – Ausgabedatenflüsse auch einen Einfluss auf Eingabedatenflüsse haben können. In anderen Worten könnte man sagen, ein Signal-Programm induziert seine eigenen Bedingungen (engl. constraints). Folglich stellt die Überprüfung von Vollständigkeit und Konsistenz eine größere Herausforderung an den Compiler dar, als dies bei Lustre der Fall ist. Bei streng mathematischer Sichtweise muss hier die Existenz eines eindeutigen Fixpunktes bewiesen werden. Dieser existiert genau dann, wenn die Konjunktion aller Bedingungen eine eindeutig erfüllbare Aussage darstellt. Da das zeitliche Modell aller Clocks eine signifikante Rolle in Signal spielt, liegt dem Compiler ein komplexer temporaler Kalkül zu Grunde.

4.7 Zeitgesteuerter Ansatz am Beispiel von Giotto

Motivation

Bei der Softwareentwicklung eingebetteter, ggf. verteilter Systeme wird derzeit in den meisten Fällen in der Industrie, vor allem in der Automobilindustrie, folgende Vorgehensweise verwendet: Das mathematische Systemmodell und damit die Funktionalität des Systems wird von einem Regelungstechniker mit Hilfe von CASE-Tools wie etwa Matlab oder MatrixX erstellt und simuliert. Soll

dieses Modell dann hinsichtlich seines Echtzeitverhaltens getestet und ggf. optimiert werden, bzw. auf Zuverlässigkeit getestet werden, wird es in der Regel an einen Programmierer weitergegeben, der dann für die Programmierung des tatsächlichen Codes, der auf der realen Plattform ausgeführt wird, verantwortlich ist. Der Programmier muss dabei ggf. periodische Aufgaben wie z. B. das Holen von Sensordaten selbst implementieren und diese Sensorwerte einzelnen Prozessoren zuweisen. Dabei müssen natürlich alle Echtzeitanforderungen eingehalten werden, damit das Verhalten des Systems vorhersagbar bleibt. Hierfür hat der Programmierer zahlreiche unterschiedliche Freiheitsgrade, die ihn in der Regel überfordern. Diese Vorgehensweise ist sehr fehlerbehaftet, und so muss der Programmierer seinen Code immer wieder testen und ggf. verbessern. Wenn der letztendlich entstandene Code dann lauffähig ist, so wurde er für genau eine Zielplattform optimiert und funktioniert daher nur auf dieser einen Plattform vorhersagbar, für die er geschrieben wurde.

Um diese langwierige, mühsame und fehlerträchtige Vorgehensweise zu verbessern, wurde Giotto entwickelt. Giotto ist ein Programmieransatz für eingebettete Systeme, der auf einer möglicherweise verteilten Hardware bzw. verteilten Plattformen zum Einsatz kommen kann. Er besteht aus einer zeitgesteuerten (engl. time-triggered) Programmiersprache, einem Compiler sowie einem Laufzeitsystem (engl. runtime system). Giotto ist besonders gut für die Softwareentwicklung für Systeme mit harten Echtzeitanforderungen und periodischem Verhalten geeignet (Henziger, 2003).

Historie, Anwendungen

Im Gegensatz zu Esterel handelt es sich bei Giotto um einen zeitgesteuerten Ansatz, der an der University of California, Berkeley in den USA im Team des Österreichers Thomas Henzinger entwickelt wurde. Ein Giotto-Programm spezifiziert die genaue Echtzeitinteraktion zwischen Softwarekomponenten und deren Systemumgebung. Unlängst durchgeführte Fallstudien und damit Beispiele für Anwendungen von Giotto sind die Ansteuerung einer Drosselklappe (Stieglbauer, 2003) im Automobil oder Controller für einen autonom fliegenden, unbemannten Helikopter (Henziger, 2003).

Giotto trägt also dazu bei, die komplexe Aufgabe der Entwicklung von Software für Steuerungssysteme zu strukturieren: Domänen-spezifische Aspekte werden von Plattform-spezifischen und zeitbezogene Aspekte werden von Daten-bezogenen Aspekten getrennt (Pree et al., 2003).

Die Giotto-Programmiersprache stellt für den Programmierer ein abstraktes Modell zur Verfügung, indem sie die Plattform-

unabhängigen funktionalen und zeitlichen Belange bzw. Anforderungen strikt von den Plattform-abhängigen Scheduling- und Kommunikationsfragen trennt. Das zeitliche Verhalten eines Giotto-Programms ist dadurch stets vorhersagbar. Damit ist Giotto besonders gut für die Entwicklung sicherheitskritischer Applikationen mit harten Echtzeitanforderungen geeignet.

Compiler

Die Aufgabe des Compilers ist es, das Giotto-Programm in ausführbaren Code zu übersetzen. Dazu muss ein Scheduling für nebenläufige Tasks bestimmt werden. Außerdem entscheidet der Compiler im Falle einer verteilten Plattform, welche Tasks von welchem Prozessor ausgeführt werden. Zu diesem Zweck benötigt der Compiler die sogenannte Worst-Case-Execution-Time eines jeden Tasks, sowie Informationen über die Leistungsfähigkeit der Plattform. Anhand dieser Informationen erstellt der Compiler dann entweder den ausführbaren Code oder – falls das auf der gegebenen Plattform nicht möglich ist – gibt eine entsprechende Fehlermeldung aus.

Das Giotto-Laufzeitsystem setzt sich aus zwei virtuellen Maschinen zusammen (vgl. Abbildung 4.5). Die erste wird als sogenannte Embedded Machine (E-Machine) bezeichnet und ist für die (reaktive) Interaktion mit der Systemumgebung zuständig. Sie interpretiert den eingebetteten Code (kurz: E-Code), der die Ausführung von Software Tasks als Reaktion auf physikalische Ereignisse aus der Umgebung überwacht.

Bei der zweiten virtuellen Maschine handelt es sich um die sogenannte Scheduling Machine (S-Maschine), welche das (proaktive) Zusammenspiel mit der Plattform regelt. Sie interpretiert den Scheduling Code (kurz: S-Code), der die zeitliche Reihenfolge der Ausführung der einzelnen Taks spezifiziert. Damit der Leser einen ersten exemplarischen Eindruck der Sprache gewinnen kann, sei auf folgendes Beispiel verwiesen.

Beispiel: Giotto

Beispiel (Giotto-Programm):
Das folgende Beispiel zeigt den Giotto-Code einer simplen, diskreten Aufzugssteuerung. Ziel dieser Steuerung ist es, den Aufzug zu demjenigen Stockwerk zu fahren, auf dem er angefordert wurde sowie dann die Tür zu öffnen und zu schließen. Der Aufzug kann durch Drücken der entsprechenden Tasten auf dem jeweiligen Stockwerk angefordert werden. Das Giotto-Programm zur Lösung dieser Aufgabe ist im Folgenden abgebildet (übernommen aus www-cad.eecs.berkeley.edu/~fresco/giotto/):

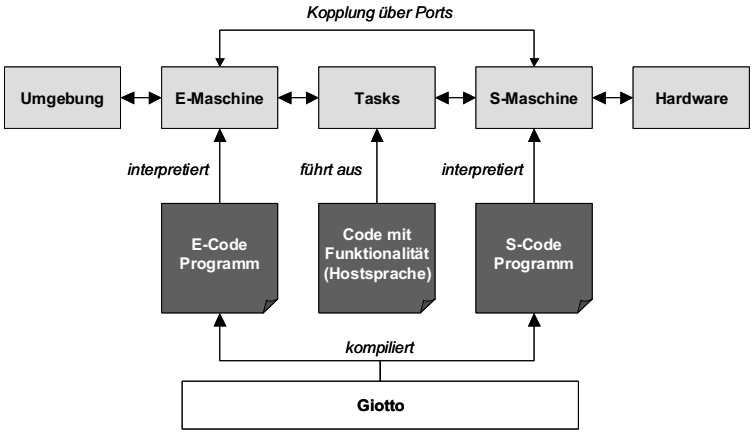


Abb. 4.5: Das Giotto-Laufzeitsystem mit E- und S-Maschine (nach Henzinger, 2003)

```
sensor
   elevator.PortButtons  buttons  uses elevator.GetButtons;
   elevator.PortPosition position uses elevtor.GetPosition;

actuator
   elevator.PortMove     motion  uses elevtor.PutMoveMotor;
   elevator.PortDoor     door    uses elevtor.PutDoorMotor;

output
   elevator.PortMove     tmotion := elevator.InitPortMove;
   elevator.PortDoor     tdoor   := elevator.InitPortDoor;
   bool_port             openwin := elevator.Elevator

task Idle(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskIdle(b, tmotion, tdoor)
}

task Up(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskUp(b, tmotion, tdoor)
}

task Down(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskDown(b, tmotion, tdoor)
}

task Open(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskOpen(b, tmotion, tdoor)
}

task Close(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskClose(b, tmotion, tdoor)
}

// Actuator driver

driver Move(tmotion) output (elevator.PortMove m) {
  if constant_true() then copy_elevator.PortMove(tmotion,
m)
}
```

```
driver Door(tdoor) output (elevator.PortDoor d) {
  if constant_true() then copy_elevator.PortDoor(tdoor, d)
}

// Input driver

driver getButtons (buttons) output (elevator.PortButtons b)
{
  if constant_true() then
copy_elevator.PortButtons(buttons, b)
}

// Mode switch driver

driver PGTC(buttons, position) output () {
   if elevator.CondPosGTCall(buttons, position) then
dummy()
}

driver PLTC(buttons, position) output () {
   if elevator.CondPosLTCall(buttons, position) then
dummy()
}

driver PEQC(buttons, position) output () {
   if elevator.CondPosEQCall(buttons, position) then
dummy()
}
driver True() output () {
   if constant_true() then dummy()
}

start idle {

          mode idle() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do up(PLTC);
                    exitfreq 1 do down(PGTC);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Idle(getButtons);
          }
          mode up() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Up(getButtons);
          }
          mode down() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Down(getButtons);
          }

          mode open() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do close(True);
                    taskfreq 1 do Open(getButtons);
          }

          mode close() period 500 {
                    actfreq 1 do motion(Move);
```

```
                    actfreq 1 do door(Door);
                    exitfreq 1 do idle(True);
                    taskfreq 1 do Close(getButtons);
            }
}
```

Zunächst werden also die für das eingebettete System erforderlichen Sensoren und Aktoren definiert, dann die einzelnen Tasks und Treiber. Die Abarbeitung der Tasks erfolgt in sogenannten Modes (siehe weiter unten). Dieses Beispiel soll lediglich dazu dienen, einen Eindruck zur Syntax zu gewinnen. Sowohl eine detaillierte Erläuterung des Beispiels als auch eine umfangreiche Einführung in Syntax oder gar Semantik von Giotto würde hier den Rahmen deutlich sprengen. Zur Betrachtung des aus diesem Giotto-Code generierten E-Code sei der Leser auf die Seiten www-cad.eecs.berkeley.edu/~fresco/giotto/ verwiesen. Wir geben im Folgenden eine informelle Einführung in die zum Verständnis des obigen Beispiels notwendigen Konzepte.

Ports

Die gesamte Datenkommunikation wird in Giotto über *Ports* abgewickelt. Jeder Port repräsentiert dabei eine Variable des Programms. Jede Variable ist von einem bestimmten Typ und einmalig. Ferner sind Ports persistent, d. h. die Variablen beinhalten stets einen definierten Wert, bis dieser explizit verändert wird. Hierdurch wird indirekt ein Puffer modelliert. Es gibt drei Arten von Ports: *Sensor Ports, Actuator Ports* und *Task Ports*. Erstere enthalten – wie der Name bereits andeutet – Daten, die von Sensoren generiert werden. In die Actuator Ports werden Daten für die ausführenden Elemente des Systems geschrieben. Die Task Ports dienen zur Kommunikation der Tasks untereinander und werden auch zum Datenaustausch zwischen unterschiedlichen Modes verwendet.

Tasks

Bei einer *Task* handelt es sich in Giotto um einen Codeausschnitt, der eine bestimmte Aufgabe ausführt. Jede Task besteht aus Eingabe-, Ausgabe- und lokalen (engl. private) Ports, sowie einer Implementierung mit definierter WCET (Worst Case Execution Time). Die WCET ist die Zeitspanne, welche die Task maximal zur Ausführung benötigt. Jeder Eingabeport darf nur einer Task zugeordnet werden, während die Ausgabeports von mehreren Tasks aktualisiert werden können (wenn die Tasks in verschiedenen Modes sind).

Der funktionale Code der Task kann in einer beliebigen Programmiersprache, also z. B. C, C++, Java, usw. implementiert werden, da der Giotto-Compiler lediglich für die zeitliche Abfolge der Tasks verantwortlich ist. Der funktionale Teil der Task wird vom Compiler der jeweiligen Programmiersprache übersetzt.

Jede Task wird in festen Perioden aufgerufen, deren Dauer ebenfalls im Programm angegeben wird. Dabei können Daten aus dem vorherigen Aufruf in den lokalen Ports für den nächsten Aufruf gespeichert werden. Bei jedem Aufruf der Task werden die Eingabe- und lokalen Ports gelesen und neue Werte für die Task Ports und Actuator Ports berechnet.

Driver

Um die Daten von den Ports zu laden, verwendet eine Task sogenannte *Driver*. Sie wickeln den Datentransport vom bzw. zur Task ab und nehmen, falls nötig, Umrechnungen vor. Das Ausführen eines Drivers geschieht nach der Semantik von Giotto in Null-Zeit. Natürlich wird in der Realität auch für diesen Datentransport tatsächlich Zeit benötigt, welche aber in der zugehörigen WCET der Task subsumiert wird.

Normalerweise ist die für eine Task spezifizierte Ausführungsperiode WCET deutlich länger, als die Ausführung der Tasks tatsächlich dauert (da es sich um eine Wort-Case-Betrachtung handelt). Die Giotto-Semantik legt nicht fest, *wann* die Task in dem verfügbaren Zeitraum ausgeführt wird. Giotto garantiert lediglich, *dass* zur Berechnung diejenigen Werte verwendet werden, die zu Beginn der Periode in den Ports gespeichert waren. Ferner wird garantiert, dass am Ende der Periode die Task vollständig ausgeführt wurde und neue Daten an den Ports anliegen. Die Ausführung einer Task ist also immer zu einem bekannten Zeitpunkt beendet und kann auch nicht vorzeitig abgebrochen werden. Aus diesem Grund ist Giotto für die Entwicklung sicherheitskritischer Echtzeitsysteme mit harten Echtzeitanforderungen geeignet. Das Ergebnis eines Giotto-Programms ist stets deterministisch, weil mit dieser Art der Synchronisation immer vorhergesagt werden kann, welche Werte zu einem bestimmten Zeitpunkt zur Verarbeitung in die Tasks eingelesen werden.

Modes

Die Ausführung eines Tasks kann vom konkreten Betriebszustand des eingebetteten Systems abhängig sein. Beispielsweise darf in einem Flugzeug die Task „Schubumkehr“ auf keinen Fall aktiv sein, wenn sich die Maschine im normalen Flugbetrieb befindet. Hierfür hat man in Giotto sogenannte *Modes* eingeführt. Jeder Mode besteht aus einem festen Satz von Tasks. Ein Giotto-Programm setzt sich dann aus der Menge aller Modes inklusive deren Transitionen zusammen. Zu einem beliebigen Zeitpunkt kann dabei immer nur ein Mode aktiv sein. Jeder Mode ist definiert durch

- eine Zeitperiode (period),
- seine Mode Ports,

- Task Invocations (taskfreq),
- Actuator Updates und
- Mode Switches (exitfreq).

Die *Zeitperiode* gibt an, wie lange die einmalige Ausführung des Modes dauert. Die *Mode Ports* dienen zur Datenkommunikation zwischen verschieden Modes und werden bei einem Wechsel des Modes aktualisiert. Die *Task Invocation* gibt an, welche Tasks in diesem Mode ausgeführt werden. Im Mode wird auch der Transport von Ergebnissen der Tasks zu den ausführenden Teilen des Systems definiert. So wird beispielsweise der Driver, der für den Transport von Daten eines Actuator Ports zum zugehörigen Task verantwortlich ist, vom Mode gestartet. Um zwischen verschiedenen Modes wechseln zu können, gibt es sogenannte *Mode Switches*, die zu den laufenden Tasks weitere hinzufügen bzw. andere entfernen. Diese Mode Switches prüfen in festen Abständen (die sogenannte Switch Frequency), ob der Mode gewechselt werden soll. Dazu überprüft ein Driver den boolschen Wert eines festgelegten Eingabe Mode Ports und entscheidet, ob ein Wechsel des Modes angefordert wurde. Im Mode Switch ist außerdem der Mode festgelegt, in den gewechselt werden soll (der sogenannte Target Mode). Ein Mode Switch ist nur dann zulässig, wenn durch den Wechsel keine laufenden Tasks unterbrochen werden.

Reaktivität

Ein Giotto-Programm spezifiziert die Reaktivität, also innerhalb welcher Zeit welche Funktionen periodisch auszuführen sind und wann die berechneten Ergebnisse weiterzugeben sind. Die Frequenz, mit der eine bestimmte Funktion auszuführen ist, wird spezifiziert. Beispielsweise wird ein Filter mit einer Frequenz von 200Hz ausgeführt, also alle fünf Millisekunden. Funktionen sind in Giotto in sogenannten Modes zusammengefasst. Zu einem Zeitpunkt ist ein Mode aktiv. An den Übergängen zwischen Modes kann auf graphische Weise durch Pfeile und deren Beschriftung angegeben werden, unter welchen Bedingungen zwischen den durch den Pfeil verbundenen Modes gewechselt wird.

Damit wird aber nicht das Scheduling bzw. die Verteilung spezifiziert. Aus Sicht eines Giotto-Entwicklers sind alle Funktionen atomare (nicht unterbrechbare) Ausführungseinheiten, ohne Prioritäten oder interne Synchronisationspunkte. Die Software-Prozesse, die von einem Giotto-Programm angestoßen werden, sind also reine Sub-Routinen (Funktionen) und keine Co-Routinen (Threads). Dem Giotto-Compiler ist es hingegen überlassen, Code für das Zeitverhalten zu generieren, der anderen Code unterbricht. Von dieser Möglichkeit wird in aller Regel auch Gebrauch gemacht,

um das Scheduling und Optimierungen zu bewerkstelligen. Damit sind die einzelnen Code-Stücke, die vom Timing Code angestoßen werden, auf Plattform-Ebene tatsächlich Co-Routinen und nicht Sub-Routinen. Diese Zweiteilung illustriert die Eigenschaften eines guten Software-Modells: Der Programmierer muss sich nicht selbst um die komplexe Aufgabe kümmern, zu welcher Zeit welche Threads nebenläufig ausgeführt werden müssen. Der Giotto-Compiler sorgt für die Effizienz des Plattform-spezifischen, ausführbaren Codes. Damit wird die Fehleranfälligkeit und somit die Zuverlässigkeit erheblich verbessert und aufgrund der Plattform-Unabhängigkeit dennoch die Basis für Komposition und Wiederverwendung geschaffen (Pree et al., 2003).

E-Machine

Die sogenannte Embedded Machine (E-Machine) in Giotto ist eine virtuelle Maschine für die Ausführung von Instruktionen, die das Echtzeitverhalten repräsentieren. Die E-Machine ergänzt das Echtzeit-Betriebssystem und repräsentiert diejenige Komponente innerhalb der Giotto-Middleware, welche zur Ausführung eines Programms benötigt wird. Der Giotto-Compiler übersetzt Plattform-unabhängigen Timing-Code in Plattform-abhängigem Code. Fallstudien (Henzinger, 2003), (Stieglbauer, 2003) haben gezeigt, dass Giotto als Embedded Software Middleware für Steuerungssysteme mit sehr engen Zeitvorgaben geeignet ist. Der durch die zusätzliche Abstraktion entstandene Zusatzaufwand hält sich nach Angaben der Entwickler in einem akzeptablen Rahmen (Pree et al., 2003).

Giotto-basierte Anwendungen profitieren von der Middleware in mehrerlei Hinsicht: Aufgrund der Semantik von Giotto, auf die im Rahmen dieses Studienhefts nicht näher eingegangen wird (vgl. (Henziger et al., 2001)), sind Giotto-Anwendungen deterministisch – das Output-Verhalten der Aktoren bei gegebenem Input an den Sensoren ist also eindeutig vorhersagbar. Das stellt eine Voraussetzung für die formale Verifikation dar. Darüber hinaus wird der Entwicklungsaufwand signifikant reduziert, da der Code für das Zeitverhalten durch einen Compiler erzeugt wird. Dies schließt eine häufige Fehlerquelle aus. Die gute Trennung von Timing Code und Functionality Code erlaubt eine leichte Änderbarkeit des Zeitverhaltens und ein einfaches Hinzufügen oder Modifizieren von Funktionalitäten. Einzelne Giotto-Programme können als Komponenten betrachtet werden; Die formale Semantik von Giotto wurde so definiert, dass bei jeder Komposition die Einhaltung der vom Programmierer einzeln spezifizierten Echtzeitanforderungen gewährleistet ist. Schließlich kann das System auf verschiedene Plattformen portiert werden, wobei lediglich die E-Machine portiert werden muss. Einschlägige Fallstudien hierzu (Henziger, 2003)

haben gezeigt, dass der Portierungsaufwand für die E-Machine etwa eine Personenwoche beträgt. Der Quellcodeumfang hierfür ist ebenfalls sehr gering (Pree et al., 2003).

Der Ansatz von Giotto löst *zwei für die Entwicklung echtzeitkritischer eingebetteter Softwaresysteme typische Probleme*, (1) erstens der Bruch zwischen dem unabhängigen, mathematischen Modell zur Systembeschreibung und des tatsächlich für eine bestimmte Plattform manuell (nach-)optimierten Codes sowie (2) zweitens die enge Verflechtung von funktionalem und zeitlichem Verhalten.

Trennung von Timing und Scheduling

(1) Die Lösung des ersten Problems gelingt durch die *Trennung von Timing (Echtzeitanforderungen) und Scheduling*. Bisher wurde bei der Entwicklung von Software für eingebettete Systeme der Code zwar weitestgehend auf Basis mathematischer Modelle erstellt, dann aber von Hand (nach-)optimiert, um die Zielplattform optimal zu nutzen und alle (harten) Echtzeitanforderungen einhalten zu können. Eingebettete Systeme mit harten Echtzeitanforderungen müssen sich neben der Einhaltung des mathematischen Modells auch um Timing und Scheduling der Aufgaben kümmern. Mit *Timing* ist hierbei die zeitliche Festlegung gemeint, wann welche Task ausgeführt werden soll. Es handelt sich hier also um einen Begriff auf Ebene der Spezifikation, wohingegen das spätere *Scheduling* erforderlich ist, um auf einer gegebenen Plattform diese Tasks so zur Ausführung zu bringen, dass alle Aufgaben gemäss der Echtzeitanforderungen rechtzeitig abgearbeitet werden.

Ein weiterer Punkt ist die starke *Plattformabhängigkeit echtzeitkritischer eingebetteter Software*. Sie kann meist nur auf derjenigen Plattform eingesetzt werden, auf der sie entwickelt wurde und ist auch dort nur schwierig erweiterbar. Giotto führt eine abstrakte Ebene zwischen dem mathematischen Modell und dem von der Plattform ausführbaren Code ein. Diese Ebene wird als eingebettetes Software Modell bezeichnet. In einem eingebetteten Software Modell spezifiziert der Entwickler nur das Timing und die Interaktion (also die Reaktivität) der Prozesse. Es ist dagegen nicht nötig, dass er oder sie sich um das Scheduling und/oder die physische Verteilung (Partitionierung) der Software auf dem eingebetteten System kümmert. In einem Giotto-Programm muss nur festgelegt werden, wann ein Sensor gelesen, seine Werte verarbeitet und die Ergebnisse an den passenden Empfänger weitergeleitet werden. Auf welchem Steuergerät und mit welcher Priorität diese Aufgabe abgewickelt wird, ist im Giotto-Programm nicht spezifiziert (hier stellen lediglich die sogenannten Annotationen in Giotto eine kleine Ausnahme dar). Mit dieser

Trennung muss sich der Programmierer ausschließlich um Plattform-unabhängige Fragen kümmern. Er legt also nur die Reaktivität des Systems auf die Umwelt fest. Die von der Plattform abhängigen Aufgaben des Scheduling und der Partitionierung werden dagegen vom Giotto-Compiler selbst übernommen (für eine detaillierte Beschreibung vgl. Beschreibungen zur E-Machine (Henziger, 2003)).

Verflechtung von funktionalem und zeitlichem Verhalten

(2) Eine zusätzliche Herausforderung stellt bei Echtzeitsystemen die in der Regel *enge Verknüpfung von Timing und Funktionalität* dar. Um das Timing, also alle Echtzeitanforderungen einzuhalten, wird bei der nachgelagerten manuellen Optimierung von Echtzeit-Programmen diese Verknüpfung zwangsläufig vom Programmierer verursacht. Werden Tasks zum Zwecke der Laufzeitperformanz so programmiert, dass sie stets asap (engl. as soon as possible) auf äußere Einflüsse wie z. B. Sensordaten reagieren, wird das Verhalten des Systems in der Regel unvorhersagbar (also nicht-deterministisch) und es entspricht eventuell auch gar nicht mehr dem ursprünglichen mathematischen Systemmodell. Giotto führt eine strikte Trennung von Timing und Funktionalität ein. Ein Giotto-Programm gibt das Timing des Systems exakt vor und überwacht es. Dabei wird in Giotto selbst die eigentliche Berechnung der Funktionalität nicht durchgeführt. Dies geschieht mit Hilfe eines in eine Hostsprache geschriebenen Programms, beispielsweise in C oder Java. Jede dieser Berechnungen wird allerdings als Task gekapselt ausgeführt und kann daher als atomar betrachtet werden: Eine einmal gestartete Taks wird nicht unterbrochen. Ein interner Synchronisationsmechanismus oder Datenaustausch mit anderen Tasks existiert nicht. Die Aufgabe des Giotto-Programmes ist es dabei, fortwährend zu überwachen, dass jede Task innerhalb ihres Zeitfensters ausgeführt wird. Hierin liegt genau die Stärke dieser Trennung von Timing und Funktionalität: Einzelne Tasks können wie Funktionen betrachtet und verifiziert werden, die hintereinander zu bekannten Zeitpunkten ausgeführt werden. Das Verhalten eines auf diese Weise programmierten Systems ist daher nur abhängig von den Sensordaten der Umgebung und somit deterministisch und verifizierbar. Dennoch spielt auch die Optimierung des zeitlichen Verhaltens eine Rolle; der Giotto-Compiler kann dabei das Scheduling eines Betriebssystems noch indirekt durch die Priorisierung der Tasks beeinflussen.

Weiterführende Information

Detaillierte Informationen zu Giotto sowie Softwarewerkzeuge möge der Leser bei Interesse im Internet auf folgenden Seiten nachschlagen: www.eecs.berkeley.edu/~fresco/giotto.

4.8 Zusammenfassung

Programmiersprachen

Wurden im Jahre 2000 noch ca. 80 Prozent aller eingebetteten Softwaresysteme mit der imperativen Programmiersprache C entwickelt, so ist für die Zukunft hier mit einem schrumpfenden Anteil zu rechnen. Mag zwar der direkte Einsatz von C++ bei Embedded Systems nicht sinnvoll sein, so können objektorientierte Sprachen wie Einschränkungen von Java oder die eigens zur Programmierung eingebetteter Systeme entwickelte objektorientierte Sprache Embedded C++ hier schon sehr gute Dienste leisten. Vor dem Hintergrund immer komplexer werdender Systeme mit stetig anwachsenden Qualitätsansprüchen gehört die Zukunft der Softwareentwicklung eingebetteter Systeme jedoch abstrakteren Ansätzen wie etwa graphischen Formalismen (Statecharts, UML, ROOM; siehe Kapitel 5) oder synchronen Sprachen (Esterel, Lustre, Signal).

Esterel

Esterel ist eine textbasierte Entwurfssprache zur ereignisgesteuerten Programmierung eingebetteter Systeme auf Basis komplexer Zustandsübergangssysteme. Esterel wurde unter anderem in Frankreich von Gérard Berry entwickelt und basiert auf dem Grundsatz der sogenannten perfekten Synchronie. Hierbei handelt es sich um die auf den ersten Blick zunächst rein theoretische Annahme, die Reaktion auf ein bestimmtes Eingabeereignis benötige keine Zeit (Nullzeit). Auf den zweiten Blick bzw. in der Praxis bedeutet diese Anforderung, dass ein Ausgabeereignis rechtzeitig, also bevor auf das nächste Eingabeereignis reagiert werden muss, produziert wird. Die Entwurfssprache Giotto dagegen verfolgt nicht die ereignisgesteuerte Philosophie, sondern ist zeitgesteuert.

Giotto

Die mit Hilfe von Giotto erstellten Programme sind portabel, haben ein vorhersagbares Verhalten und entsprechen Echtzeitanforderungen besser als herkömmlich erstellter Code. Die Reaktion auf Ereignisse in Echtzeit geschieht deterministisch, weil das Verhalten der Programme vorhersagbar ist, da dieses nur auf Einflüsse der Umwelt reagiert. Es gibt keinen internen Wettlauf zwischen den Programmteilen, da genau festgelegt ist, wann welche Aufgabe (Task) ausgeführt wird. Der Code ist portabel, weil er weder von der Struktur, dem Scheduling oder der Laufzeitperformanz einer einzelnen Plattform abhängig ist. Nur die Umgebungszeit ist für das Timing wichtig – diese wiederum ist aber von der Plattform unabhängig. Ein mit Giotto erstelltes Programm entspricht exakt dem mathematischen Modell des eingebetteten, möglicherweise verteilten Systems mit harten Echtzeit-

anforderungen. Auf diese Weise sind bei der Systementwicklung keine langwierigen Test- bzw. Optimierungszyklen mehr erforderlich und insgesamt kann die Entwicklungszeit verkürzt werden. Aufgrund der zeitlichen Invarianz bei der Komposition von Giotto-Modulen ist eine Erweiterung der Funktionalität möglich, ohne das zeitliche Verhalten der einzelnen Module zu beeinflussen.

5 Softwareentwurf eingebetteter Systeme

Wie wir bereits in Kapitel 4 gesehen haben, besteht die Entwicklung eingebetteter Systeme nicht nur aus deren Programmierung, sondern erfordert darüber hinaus eine systematische Vorgehensweise, die sich in aufeinander folgende Entwicklungsphasen unterteilen lässt. In der Phase des Softwareentwurfs (also vor der Programmierung) muss sich der Entwickler Spezifikationstechniken bedienen, die eine abstraktere, aber trotzdem möglichst eindeutige und vollständige formale oder semiformale Beschreibung des Systems als Ergebnis haben. Ausgewählte Vertreter dieser visuellen Beschreibungsmöglichkeiten werden wir in diesem Kapitel betrachten. Dabei sei auch erwähnt, dass die Grenzen zwischen dem Entwurf und der Programmierung eingebetteter Systeme zunehmend verschwimmen. So hätte man prinzipiell auch die Beschreibung von Esterel und Giotto in dieses Kapitel mit aufnehmen können. Insgesamt besteht hier langfristig die Bestrebung, eingebettete Systeme überhaupt nicht mehr programmieren zu müssen, sondern ganz auf abstraktere Beschreibungsmöglichkeiten zurückgreifen zu können.

Eingebettete Systeme werden in der Regel nicht von HW- oder SW-Spezialisten sondern von Experten des jeweiligen Fachgebiets entwickelt (z. B. Regelungstechnik). Die verwendeten Formalismen zur Beschreibung des Verhaltens der Funktion sind Domänenspezifisch optimiert (z. B. Zustandsautomaten, Differenzen- oder Differentialgleichungen). Darüber hinaus enthalten diese Systeme immer auch zusätzliche Funktionalitäten, wie Kommunikation oder Diagnose, die mit gänzlich anderen Formalismen adäquat beschrieben werden. Diese heterogenen Beschreibungen sind Ausgangspunkt des Hardware/Software-Codesigns. Nach der Lektüre dieses Kapitels sollte der Leser die wichtigsten Modellierungstechniken zur Entwicklung eingebetteter Systeme kennen und voneinander abgrenzen können.

5.1 Modellierung eingebetteter Systeme

Klassifikation

Die zur Analyse und Modellierung eingebetteter Systeme verwendeten Sprachen können auf mehrere Arten klassifiziert werden:

- Modellierung diskreter und/oder kontinuierlicher Information,
- Zustands-, Aktivitäts- und Struktur-orientierte Verfahren und deren Mischformen,
- mit oder ohne Modellierung zeitabhängiger Informationen.

Viele eingebettete Systeme v. a. in der Automobiltechnik zeichnen sich durch einen *hybriden* Charakter aus: Sie steuern bzw. regeln Vorgänge, in denen diskrete und kontinuierliche Anteile in Wechselwirkung stehen.

Folgende Sprachen werden zurzeit auf breiter, größtenteils industrieller Basis zur Modellierung und Simulation von Software für eingebettete Systeme verwendet. Nachstehende Liste erhebt keinen Anspruch auf Vollständigkeit; vgl. (Broy et al., 1998):

Beschreibungs-techniken

- Statecharts
- Hybrid Statecharts
- ROOM
- VHDL-A
- MSC bzw. EET
- Structured Analysis / SA-RT
- UML
- SDL
- OMT
- Entity-Relationship-Diagramme
- Timed Automata
- Hybride Automaten
- Temporale Logiken (z. B. CTL, LTL)
- mechatronisches Datenmodell
- u.v.m.

5.2 Formale Methoden

Gerade weil eingebettete Systeme zunehmend in sicherheitskritischen Bereichen eingesetzt werden und dadurch die Qualitätssicherung (vgl. Kapitel 6) von besonderer Bedeutung ist, leistet die theoretische Informatik durch Bereitstellung formaler Grundlagen zur *Spezifikation, Verifikation und Codeerzeugung* einen entscheidenden Beitrag zur effektiven Entwicklung eingebetteter Systeme, die damit ein Paradebeispiel für ein interdisziplinäres Forschungs-, Entwicklungs- und Arbeitsfeld der Informatik darstellen (Broy et al., 1998).

Gerade in dem Bereich der *Verifikation* existieren schon viele formale Verfahren und Werkzeuge. Diese werden bisher jedoch kaum im industriellen Softwareentwicklungsprozess für Steuergeräte eingesetzt. Hier ist auch der Aspekt der Skalierbarkeit entscheidend: Beispielsweise entsteht beim Model Checking oft das Problem, dass die zu verifizierenden Systeme, insbesondere beim Auftreten von nebenläufigen Vorgängen, zu komplex für die Bearbeitung sind. In diesem Fall führen Ansätze zur kompositionellen Verifikation und/oder Abstraktionsverfahren zu einer nennenswerten Verbesserung.

In vielen Fällen mangelt es jedoch an der *werkzeugorientierten Umsetzung* in softwaretechnischen Methoden und Vorgehensmodellen für die Praxis: Während formale Methoden für etablierte visuelle Formalismen wie etwa Statecharts mittlerweile in Form von Add-Ons zur Verifikation oder automatischen Test-Generierung für Design-Tools wie Statemate (Firma iLogix) Einzug in die industrielle Praxis gehalten haben, sind formale Methoden für den UML Bereich (UML = Unified Modeling Language) weiterhin ein Gebiet aktiver Forschung.

Um den *Herstellungsprozess für Steuergeräte* bzw. deren Steuerungssoftware möglichst flexibel, kostengünstig und schnell zu gestalten, braucht es Möglichkeiten zur Codegenerierung, d. h. zur automatischen Erzeugung von Code aus einer (formalen) Spezifikation. Hier existieren zwar bereits Lösungen in Form von Methoden und teilweise sogar unterstützenden Werkzeugen, jedoch ergibt sich hier immer noch das Problem, dass der resultierende, automatisch generierte Code zu umfangreich ist (ca. Faktor 2) und/oder eine zu langsame Ausführungszeit besitzt.

5.3 Statecharts

Historie, Prinzip

Statecharts, eine graphische Spezifikationssprache für komplexe, zustandsbasierte Systeme, wurden bereits 1987 von *David Harel* als eine Erweiterung sequentieller Automaten (diese bestehen aus Zuständen und Transitionen) eingeführt. Statecharts kombinieren diese Automaten mit Konzepten für Nebenläufigkeit, Kommunikation und hierarchischer Dekomposition. Sie sind heute Basis vieler Werkzeuge im industriellen Einsatz zur Modellierung eingebetteter Systeme. Statecharts beschreiben das zustandsbasierte Ablaufmodell einer Softwarekomponente.

Von Statecharts existieren heute zahlreiche Varianten mit teilweise sehr unterschiedlicher Syntax und vor allem Semantik. In der *UML* (Unified Modeling Language) werden Statecharts häufig auch eingesetzt, um den Lebenszyklus eines Objekts, also die Abhängigkeiten zwischen Aufrufen von Objektmethoden und dem Zustand des Objekts zu beschreiben. Ein Statechart kann darüber hinaus auch dazu verwendet werden, um das Ablaufmodell einer komplexen Methode darzustellen.

Im Gegensatz zu Sequenzdiagrammen zeigt ein Statechart keine Interaktion zwischen Objekten, sondern modelliert gleichzeitig den Zustand und das Verhalten eines Objekts. Dabei können Statecharts das Verhalten vollständig beschreiben, während Sequenzdiagramme meist exemplarisch sind. Leider wird dadurch die Statechart-Notation komplexer und benötigt einigen Lernaufwand.

Transitionen

Die *Transitionen* (Zustandsübergänge, Pfeile, Kanten) sind im Allgemeinen mit einem Ereignis e bzw. dem Namen einer Methode beschriftet, welche(s) die Transition schaltet. Zusätzlich kann ggf. eine Bedingung b angegeben werden, unter der die Transition erfolgt. Darüber hinaus kann jede Transition mit einer Aktion a attributiert sein, die beim Schalten der Transition ausgeführt wird. Die Schreibweise für die Beschriftung der Transitionen in den meisten Statechartsvarianten lautet damit wie folgt:

„e [b] / a“.

Nebenläufigkeit, Kommunikation, Komposition, Dekomposition

Um die Spezifikation praktisch relevanter Systeme zu ermöglichen, können die Automaten in Statecharts parallel komponiert oder hierarchisch dekomponiert werden. Statecharts erweitern das Modell der sequentiellen Automaten um die Konzepte der *Nebenläufigkeit* durch Komposition, der *Kommunikation* (zwischen Automaten) sowie der *hierarchischen Dekomposition.* Durch das Konzept der

Nebenläufigkeit bzw. Parallelkomposition lässt sich das Verhalten verteilter eingebetteter Systeme beschreiben.

Mit Hilfe der hierarchischen Dekomposition können Gruppen von Zuständen zusammengefasst werden. Bei der *Parallelkomposition* ist die wesentliche Grundannahme, dass zwei auf diese Weise zusammengesetzte sequentielle Automaten im Gleichtakt in Bezug auf einen gemeinsamen Taktgeber ihre Zustandsübergänge vornehmen. Ohne weiteres Zutun interagieren die auf solche Weise komponierten Automaten zunächst überhaupt nicht. Beim Entwurf eingebetteter Software mit Statecharts kann jedoch zusätzlich angegeben werden, dass diese Mealy Maschinen das gegenseitige Verhalten durch den Austausch von Nachrichten wechselseitig beeinflussen können.

Informelle Semantik

Informell lautet die Semantik dann wie folgt: Liegt ein passendes Eingabeereignis im aktuellen Systemschritt bzw. -takt (vgl. e_1 in Abbildung 5.1) an und ist die angegebene Zusatzbedingung (vgl. b_1 in Abbildung 5.1) logisch wahr, so wird der Zustandsübergang vollzogen und gleichzeitig die spezifizierte Aktion (vgl. a_1 in Abbildung 5.1) ausgeführt. Enthält a_1 ein Ereignis, welches das zweite parallel komponierte Statechart im aktuellen Zustand beeinflussen kann, so vollzieht auch dieses alle hierdurch ermöglichten Zustandsübergänge (in Abbildung 5.1 vollzieht sich daher auch der Zustandswechsel von z_2 zu z_3 – vorausgesetzt dass zusätzlich auch die Bedingung b_2 wahr ist). Die (formale) Semantik zahlreicher Statechartsvarianten unterscheidet sich im Übrigen meist genau darin, welches zeitliche Verhalten im Detail diesem kombinierten Verhalten zweier auf diese Art komponierten Statecharts zugrunde liegt.

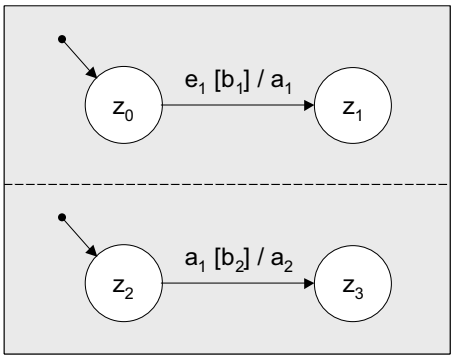


Abb. 5.1: Parallelkomposition bei Statecharts

Parallel-komposition

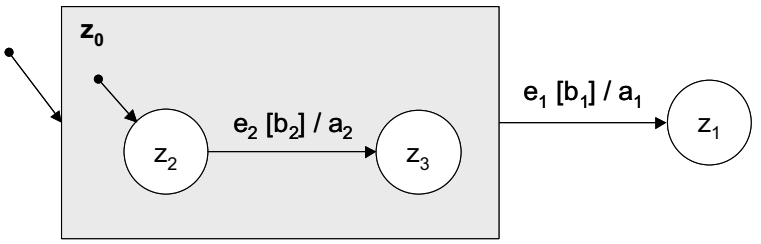
Die Parallelkomposition ermöglicht eine wesentlich kompaktere Darstellung komplexer Zustandsräume als es mit einem einzigen sequentiellen Automaten möglich wäre. Der Zustandsraum einer Parallelkomposition ist das algebraische Produkt der Zustandsräume beider im Rahmen der Komposition verbundenen Automaten, wodurch die Größe der Spezifikation anstelle eines quadratischen Wachstums bezogen auf die Anzahl der Zustände lediglich linear anwächst.

Hierarchische Dekomposition

Reaktive, eingebettete Systeme nehmen meist komplexe Systemzustände ein, so dass einzelne Zustände eines Automaten nicht für deren realistische Abstraktion geeignet sind und somit ausgefeiltere Ansätze gebraucht werden. Neben der Parallelkomposition erlauben daher Statecharts eine weitere Spezifikationstechnik, welche diesem Umstand Rechnung trägt und einzelne Zustände eines Automaten durch *hierarchische Dekomposition* in Subzustände aufteilen lässt: Das Statechart, welches das Systemverhalten in einem spezifischen Zustand in größerem Detail beschreibt, wird auf graphischer Ebene einfach als Subkomponente innerhalb dieses Zustands, der damit zu einem sogenannten „Superzustand“ wird, gezeichnet.

Das Verhalten eines hierarchisch dekomponierten Statecharts ist vergleichbar mit dem eines Prozedur- bzw. Funktionsaufrufs bei einer imperativen Programmiersprache wie etwa Pascal oder C. Ein Automat auf einer hierarchisch höheren Ebene ruft im Sinne dieses Vergleichs „Subroutinen“ auf, nämlich das innerhalb des Zustands modellierte Statechart und zwar immer in genau den Systemtakten, wenn der Superzustand selbst aktiv ist, d. h. sich der Ablauf der Systemreaktion in genau diesem Zustand befindet.

Abb. 5.2: Hierarchische Dekomposition bei Statecharts



UML

In der *UML* (Unified Modeling Language) werden Statecharts häufig auch eingesetzt, um den Lebenszyklus eines Objekts, also die Abhängigkeiten zwischen Aufrufen von Objektmethoden und dem Zustand des Objekts zu beschreiben. Ein Statechart kann darüber

hinaus auch dazu verwendet werden, um das Ablaufmodell einer komplexen Methode darzustellen.

Im Gegensatz zu Sequenzdiagrammen zeigt ein Statechart keine Interaktion zwischen Objekten, sondern modelliert gleichzeitig den Zustand und das Verhalten eines Objekts. Dabei können Statecharts das Verhalten vollständig beschreiben, während Sequenzdiagramme meist exemplarisch sind. Leider wird dadurch die Statechart-Notation komplexer und benötigt einigen Lernaufwand.

5.4 Die Unified Modeling Language (UML)

Aufgrund der stetig wachsenden Komplexität von Softwaresystemen verfolgt man in den letzten Jahren zunehmend das Ziel, den Entwurf und die Programmierung solcher Systeme mit Hilfe objektorientierter (OO) Ansätze zu vereinfachen. Eine Möglichkeit hierfür ist die Unified Modeling Language (UML), eine Sprache mit verschiedenen, meist graphischen Beschreibungsmitteln zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme.

Die UML stellt aktuell einen Standard dar und ist wegen ihrer Struktur- und Verhaltensbeschreibungsmittel und zahlreicher Notationselemente sehr umfangreich.

Historie

Die Entwicklung objektorientierter Ansätze geht in die achtziger Jahre zurück. Die wichtigsten damals verbreiteten dieser OO-Ansätze waren:

- Booch Methode (Grady Booch),
- OOSE (Ivar Jacobsen) und
- OMT (Jim Rumbaugh).

Zwischen diesen Ansätzen gab es jeweils Unterschiede hinsichtlich der verwendeten Notation, so dass ein einheitlicher Standard nicht möglich war, zumal dieser von den jeweiligen Verfechtern der Ansätze auch abgelehnt wurde. Im Oktober 1994 verließ Jim Rumbaugh General Electric und wechselte zur Firma Rational (mittlerweile IBM), wo bereits Grady Booch tätig war. Zusammen gaben sie im Oktober 1995 ihre Veröffentlichung über die Unified Method Version 0.8 heraus und erklärten den „Methodenkrieg“ damit für beendet: „The methods war is over – we won“. Im Herbst 1995 kaufte Rational auch die Firma Objectory, für die der dritte im Bunde, Ivar Jacobsen tätig war. Booch, Jacobsen und Rumbaugh

werden seitdem auch die „drei Amigos“ genannt. Die Arbeiten an der UML begannen 1996, und eine Version 0.9 wurde im Juni 1996 publiziert. Die Unified Modeling Language Version 1.0 wurde im Januar 1997 veröffentlicht und bei der Object Management Group (OMG) als Standardisierungsvorschlag eingereicht. Als die drei Amigos im September 1997 die Version 1.1 einreichten, wurden alle Gegenvorschläge zurückgezogen, da deren Konzepte bereits von dieser UML-Version berücksichtigt wurden. Im November 1997 wurde schließlich die UML 1.1 von der OMG als Standard verabschiedet. (Balzert, 2001). Aktuell ist die UML in der Version 2.0 verfügbar.

Sprache, nicht Methode

Die UML ist eine Modellierungs*sprache*, keine Methode und enthält darum keinerlei Beschreibung für Prozesse. Ihre hauptsächliche Anwendung besteht in der Modellierung von Softwaresystemen und, wenn möglich, anschließender Codegenerierung. Das zu entwerfende System wird in der UML mittels der Diagramme definiert und festgelegt; anschließend wird ggf. ein Codegerüst generiert, das die Basis für die darauf folgende Programmierung darstellt (Oestereich, 1997).

UML 2.0

Die UML ist so konzipiert, dass sie unabhängig von der Anwendungsdomäne funktioniert. Dennoch enthält die UML 2.0 neue Aspekte, welche die Entwicklung von Echtzeitsystemen unterstützen:

- Modellierung interner Strukturen,
- Timing Diagramme (neu),
- bessere Verhaltensmodellierung bei Aktivitätsdiagrammen, Sequenzdiagrammen und Zustandsdiagrammen durch Strukturierungsmöglichkeiten komplexer Diagramme sowie Festlegung formaler(er) Semantik und
- ausführbare Modelle.

Änderungen von UML 1.5 auf UML 2.0

Insgesamt haben sich bei der UML 2.0 im Vergleich zu ihrer Vorgängerversion 1.5 folgende Änderungen ergeben:

- Marginale Änderungen
  - Klassen-Diagramm
  - Use Case-Diagramm
  - Objektdiagramm
- Kleine Änderungen:
  - Verteilungsdiagramm

- Paketdiagramm
- Tiefgreifende Änderungen:
  - Aktivitätsdiagramm
  - Sequenzdiagramm
  - Zustandsautomat
- Vollständig neu sind:
  - Kompositionsstrukturdiagramm
  - Interaktionsübersichtsdiagramm
  - Zeitdiagramm
  - Kommunikationsdiagramm

In der Version 2.0 ist die UML auch in den Interessensbereich von Entwicklern eingebetteter Systeme gerückt – und dies nicht nur aufgrund der immer komplexer werdenden Systeme. Die gegenüber den Vorläuferversionen formalere Spezifikation der UML 2.0 stellt die Ausführbarkeit der mit der UML beschriebenen Modelle sicher. Eine Überprüfung von Systemeigenschaften ohne Verlassen der Modellierungsebene ist somit möglich. Ferner ermöglicht die UML 2.0 exaktere Systembeschreibungen, die eine möglichst präzise Nachbildung der (Kunden-)Anforderungen ermöglichen.

Insgesamt ist vor dem Hintergrund des zurzeit bei eingebetteten Systemen immer weiter verbreiteten Einsatzes objektorientierter Sprachen wie C++ oder Java davon auszugehen, dass die UML nun endgültig auch bei der Entwicklung von Embedded Systems Fuß fassen wird. Immer mehr Modellierungswerkzeuge aus diesem Bereich setzen die UML 2.0 ein. Eine Modellierungstechnik wie etwa die Zeitdiagramme sind in der Lage, eine Verbindung zwischen der Entwicklung zeitkritischer Software einerseits und Model-Driven-Development andererseits herzustellen.

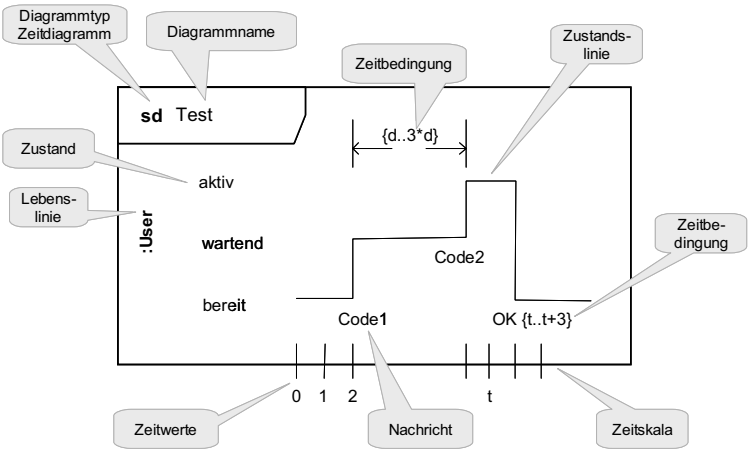
Die Neuauflage der UML mit ihrer Version 2.0 hat einige interessante Aspekte für Entwickler von eingebetteten Systemen bzw. Echtzeitsystemen zu bieten, die im Folgenden exemplarisch anhand der Zeitdiagramme (auch: Timing-Diagramme von engl. timing diagrams) dargestellt werden soll. Eine detaillierte Einführung in die UML kann im Rahmen dieses Buches nicht geschehen. Der interessierte Leser sei beispielsweise auf (Balzert, 2001), (Grässle et al., 2003) und (Oestereich, 2004) verwiesen. Bewährte Konzepte aus „Real-Time Object-Oriented Modeling" (ROOM, vgl. Abschnitt 5.5), einem der bekanntesten Ansätze zur Modellierung von Echtzeitsystemen wurden in die UML 2.0 aufgenommen.

Timing-Diagramme:

Timing-Diagramme

Timing-Diagramme ermöglichen die Modellierung von zeitgenauen Zustandsübergängen. Letztgenannte Beschreibungstechnik wird schon lange erfolgreich von Ingenieuren aus dem Bereich der Elektronik bzw. Elektrotechnik verwendet und wurde nun in die UML 2.0 als eigene Diagrammform adaptiert aufgenommen (siehe auch Abbildung 5.3).

Abb. 5.3: Beispiel Timing Diagramm



Sequenz-Diagramme eignen sich besonders gut, um die Reihenfolge von Methodenaufrufen darzustellen. Sie sind jedoch nicht geeignet, um den exakten zeitlichen Ablauf von Aktionen darzustellen. Um den Zeitaspekt besser darzustellen, wurden die aus *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 1998 bekannten Timing-Diagramme in die UML 2.0 mit aufgenommen.

Ein Timing-Diagramm beschreibt die zeitlichen Bedingungen (Zeitanforderungen, Zeitgrenzen, zeitliche Abhängigkeiten) von Zustandswechseln mehrerer Objekte. Zeitdiagramme finden ihre Verwendung vor allem in der graphischen und dennoch formalen Beschreibung von Echtzeitsystemen.

Ein Zeitdiagramm besteht aus zwei Achsen. Auf der Abszisse verläuft von links nach rechts die Zeit linear. Auf der Ordinate werden die zu betrachtenden Objekte und ihre (diskreten) Objektzustände dargestellt. Mit Hilfe von Linien kann der Zusammenhang verschiedener Zustandswechsel im zeitlichen Verlauf spezifiziert werden. Sowohl signifikante Zeitpunkte also auch Zeiträume können auf der Abszisse benannt werden.

Rational Unified Process (RUP)

Die UML stellt eine Ansammlung von Beschreibungstechniken dar. Sie selbst stellt kein Vorgehensmodell zur Softwareentwicklung zur Verfügung. Dies kann jedoch ergänzend der eigens für die UML entwickelte „Rational Unified Process“ (kurz: RUP) leisten. Das RUP-Modell wurde 1998 durch die Firma Rational entwickelt. Es bietet einen Anwendungsfall-zentrierten Zugang zur Softwareentwicklung. Es bildet das Rahmenwerk für den gesamten Entwicklungsprozess. Aktivitäten, Messkriterien und Ergebnisse werden definiert und gesteuert. Die Sprache des RUP ist die UML; Artefakte sind die Grundlage des Systems. Das Projekt wird durch das RUP-Modell dokumentiert.

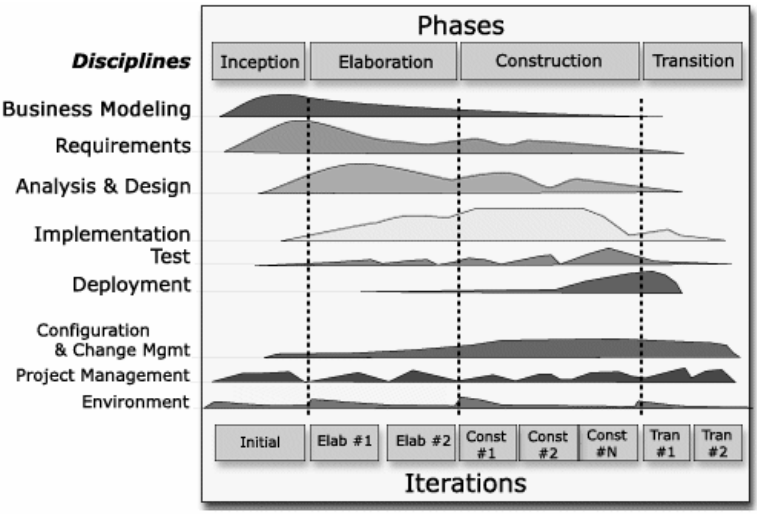


Abb. 5.4: RUP (Quelle: IBM, vormals Rational)

Bei dem Modell wurde darauf geachtet, sich an „Best Practices“ zu orientierten. Es vereint eine iterative Softwareentwicklung und das Anforderungsmanagement, es verwendet Komponenten-basierte Architekturen. Die Software wird visuell modelliert. Die Softwarequalität wird geprüft und das Änderungsmanagement unterliegt einer Kontrolle. Der RUP definiert Workflows für folgende neun Kernaufgaben (engl. disciplines), siehe Abbildung 5.4:

9 RUP Kernaufgaben

- Business Modeling (Geschäftsprozessmodellierung)
- Requirements (Anforderungsanalyse)
- Analysis and Design (Analyse und Design)
- Implementation (Implementierung)
- Test

- Deployment (Softwareverteilung)
- Configuration and Change Management (Konfigurations- und Änderungsmanagement)
- Project Management (Projektmanagement)
- Environment (Umgebung)

4 RUP Phasen

Die vier dazugehörigen Phasen definieren sich wie folgt (siehe auch Abbildung 5.4):

- **Phase 1 (Inception, Konzept)** spezifiziert die Endproduktversion und die wesentlichen Geschäftsvorfälle. Der Umfang des Projekts sowie Kosten und Risiken werden prognostiziert. Ergebnis von Phase 1 ist der „Life Cycle Objective Milestone".
- **Phase 2 (Elaboration, Entwurf)** spezifiziert die Produkteigenschaften, die Architektur wird entworfen, notwendige Aktivitäten und Ressourcen werden geplant. Diese Phase endet mit dem „Life Cycle Architecture Milestone".
- In **Phase 3 (Contruction, Implementierung)** wird das Produkt erstellt und die Architektur entwickelt. Am Ende dieser Phase ist das Produkt fertig und der „Initial Operational Capability Milestone" abgeschlossen.
- **Phase 4 (Transition, Produktübergabe)** beinhaltet die Übergabe des Produkts an die Benutzer, sowie die Überprüfung des Qualitätslevels. Der „Release Milestone" ist damit gesetzt.

Innerhalb jeder Phase erfolgt die Aufteilung der Aufgaben in kleine Schritte (Iterationen), deren Workflow ähnlich wie beim Wasserfallmodell abgearbeitet werden. Es stehen jederzeit Planungshilfen, Leitlinien, Checklisten und Best Practices zur Verfügung.

Mario Jeckle† schrieb zum RUP: „Die konsequente und komplette Nutzung von RUP macht erst bei Teams mit über 10 Personen Sinn. Es müssen über 30 Rollen besetzt werden und über 100 verschiedene Artefakttypen erzeugt, dokumentiert und verwaltet werden." Artefakttypen sind Arbeitsergebnistypen, also Informationseinheiten, die während des Projektes entstehen und gebraucht werden; sie stellen sowohl Ein- als auch Ausgabe von Aktivitäten dar.

Beim RUP-Modell gibt es weitestgehend keine Kontrolle für die gesamte Qualitätsverantwortung. Stattdessen ist jeder Mitarbeiter für die Qualität seiner Arbeitsergebnisse selbst verantwortlich. Erreicht

wird dies durch zwei verschiedene Rollen innerhalb jeder Phase in der der Mitarbeiter entweder für die Qualitätsplanung (dieser Phase) oder für die Qualitätssicherung zuständig ist.

5.5 Der Ansatz ROOM

Architektur- und Detailschicht

ROOM (Real-time Object Oriented Modeling) wurde von Objectime für den Entwurf von Echtzeitsystemen entwickelt. Es teilt sich in eine Architekturschicht sowie eine Detailschicht auf. In der Architekturschicht wird ein Grobentwurf des Gesamtsystems mit Hilfe von graphischen Darstellungsmitteln erstellt. In der Detailschicht werden die Funktionen mit herkömmlichen Programmiersprachen wie C/C++ und Java beschrieben. Besonders hervorzuheben ist die formale semantische Fundierung von ROOM. Auf diese Weise sind bereits Grobentwürfe ausführbar bzw. testbar (Selic, 1994).

5.5.1 Softwarewerkzeuge und Umgebung

ROOM ist insbesondere für die Software-gestützte Anwendung entwickelt worden; denn nur so können seine Stärken in vollem Umfang genutzt werden. Eine dieser Stärken ist die automatische Codegenerierung aus einem Entwurf in ROOM heraus. Es kann jederzeit ein lauffähiger Code erzeugt werden. Dadurch werden Änderungen nur am Entwurf in der Architekturschicht und in der Detailschicht durchgeführt und nicht im Code. Ein „Architekturverfall“, wie er oft bei der Entwicklung mit UML vorkommen kann, wird damit vermieden.

Echtzeitsysteme

Echtzeitsysteme werden meist mit Spezialhardware betrieben. Da es oft nicht möglich ist, die Software direkt auf dem Zielsystem zu entwickeln, gibt es bei ROOM-Werkzeugen die Möglichkeit, das Zielsystem auf einem Entwicklungsrechner (Arbeitsplatzrechner) zu emulieren. Dadurch kann bereits im Vorfeld erkannt werden, ob das Zusammenspiel zwischen Hard- und Software reibungslos funktioniert.

Neben der Funktionalität kann mit der Emulation auch das Echtzeitverhalten überprüft werden. Durch die systematische Abbildung der Modelle auf Programmcode ist die Ausführungszeit für jedes Konstrukt nachvollziehbar. Auf diese Weise kann schon

sehr früh erkannt werden, ob die Hardware genügend Rechenleistung besitzt, um alle Echtzeitanforderungen zu erfüllen. Jedoch können auf diese Weise nicht alle prinzipiell auftretenden Ablaufmöglichkeiten des Kontrollflusses abgedeckt werden. Da die Detailschicht in einer herkömmlichen Programmiersprache implementiert wird, ist es nicht ohne weiteres möglich, Rückschlüsse auf deren Laufzeit zu ziehen. Um diese Hürde zu überwinden, kann bei der Simulation die voraussichtliche reale Laufzeit dieser Programmteile manuell angegeben werden. Neben der Emulation kann das Laufzeitverhalten des Entwurfs auch analytisch bestimmt werden. Dadurch kann der denkbar schlechteste Fall (engl. worst case) gefunden werden.

Ein sehr mächtiges Tool das diese Eigenschaften unterstützt ist „Rational Rose Real Time“ das von IBM (früher: Rational) vertrieben wird.

Wie bereits erwähnt, findet beim Entwicklungsprozess mit ROOM der Softwareentwurf auf einem Arbeitsplatzrechner statt und nicht auf dem Zielsystem selbst. Um eine bestmögliche Portabilität vom Entwicklungsrechner auf das Zielsystem zu gewährleisten, wird sowohl bei der Simulation als auch auf dem Zielsystem die ROOM Virtual Machine verwendet. Sie besitzt eine zur Java Virtual Machine vergleichbare Funktionsweise, erlaubt aber zusätzlich an ihr vorbei den direkten Zugriff auf die Hardware.

5.5.2 Einführung

Aktoren

Aktoren sind vergleichbar mit Klassen in anderen objektorientierten Programmiersprachen. Sie stellen eine vollständige Programmeinheit dar, d. h. sie können unabhängig von anderen Aktoren ausgeführt und getestet werden. Im Programm werden sie nebenläufig ausgeführt; jeder Aktor bildet dabei eine eigene Task. Aus diesem Grunde wird beim verwendeten Betriebssystem die Multitasking-Fähigkeit vorausgesetzt. Auf die innere Struktur kann von außen nicht direkt zugegriffen werden, sondern lediglich über ihre Schnittstellen. Dadurch wird eine Datenkapselung realisiert. Aktoren können hierarchisch dekomponiert werden. Sie bestehen dann aus mehreren Subaktoren (Selic, 1994).

Protokolle, Ports, Nachrichten

Da Aktoren nicht direkt miteinander kommunizieren können, werden hierfür spezielle Schnittstellen, sogenannte *Ports*, benötigt. Über Ports werden bidirektionale Nachrichten ausgetauscht, die aus einem Signal sowie optionalen Parametern bestehen. Über einen Port dürfen nur Nachrichten bestimmter Typen gesendet bzw.

empfangen werden. Diese Nachrichtenregeln werden in *Protokollen* festgelegt. Ein Protokoll besteht wiederum aus einer Menge von Signalen, deren Parametern und der jeweiligen Kommunikationsrichtung (ein- oder ausgehend). Um zwei Ports zu verbinden, muss ein Port konjugiert werden, d. h. die Nachrichtenrichtung wird invertiert. Ein Protokoll muss dann nur einem der beiden Ports zugewiesen werden, da dem anderen Port automatisch das konjugierte Protokoll zugewiesen wird. Ist ein Aktor in Subaktoren dekomponiert und soll ein Port einer der Subaktoren nach außen weitergegeben werden, so können normale Ports nicht benutzt werden, da der empfangende Port die Nachricht nicht nach außen weiterleiten würde. Eine Lösung bieten hier die sogenannten „Relay Ports“, die eine Nachricht empfangen und weiterleiten (Selic, 1994).

ROOM Charts

In ROOM wird das Verhalten eines Programms durch endliche Automaten beschrieben. Die graphische Darstellung dieser Automaten bezeichnet man als ROOM Chart. Die Zustandswechsel werden durch sogenannte Trigger ausgelöst. Ein Trigger besteht aus einem Signal und einem Port. Kommt nun an diesem Port das angegebene Signal an wenn der Trigger aktiv ist, findet ein Zustandsübergang statt. Bei jedem Zustandsübergang kann zusätzlich eine Aktion angegeben werden, die beim Zustandsübergang ausgeführt wird. Bevor ROOM ein neues Signal verarbeitet, führt es vorher die Aktion vollständig aus. Da es bei den parallel laufenden Aktoren vorkommen kann, dass zwei Signale zum fast selben Zeitpunkt kommen hat jeder Port eine Warteschlange, welche die Signale solange puffert, bis sie abgearbeitet werden können. Das bei Echtzeitsystemen oft Signale vorkommen, die sofort ausführt werden müssen (z. B. der Notstopp einer Maschine), gibt es die Möglichkeit, den Signalen eine statische Priorität zuzuordnen. Dadurch können Signale mit hoher Priorität solche mit einer niedrigeren in der Warteschlange überholen.

Wie bei Aktoren ist es auch bei Zuständen möglich, sie hierarchisch zu dekomponieren. Befindet man sich aber gerade bei der Abarbeitung eines Subzustands und ein Trigger aus der nächst höheren Kompositionsebene wird ausgelöst, so wird die Abarbeitung des Subzustands abgebrochen, und der Zustandsübergang der höheren Ebene wird ausgeführt. Dies geschieht aber nur dann, wenn der gleiche Trigger nicht auch bei beim Subzustand einen Zustandsübergang auslöst. Mit bedingten Übergängen ist es möglich, den Zielzustand eines Zustandsübergangs von zusätzlichen Bedingungen abhängig zu machen (Selic, 1994).

Vererbung

Um eine bessere Wiederverwendbarkeit der Konstrukte zu ermöglichen, unterstützt ROOM die Vererbung für die Sprachelemente Aktoren, ROOM Charts sowie Protokolle. Allerdings ist nur die Einfachvererbung möglich; ansonsten können alle Attribute vererbt, verändert oder gelöscht werden.

Dynamischer Entwurf

Die bisher vorgestellten Elemente erlauben nur den statischen Entwurf. In ROOM ist es aber möglich Aktoren erst zur Laufzeit zu erzeugen. Dazu dienen optionale Instanzen als Platzhalter für dynamische Aktoren, die statisch festlegen, welche Aktorklasse verwendet werden soll. Durch eine Kombination aus austauschbaren und optionalen Instanzen wird eine Schnittstelle für die zur Laufzeit erzeugten Objekte geschaffen. Dies ist möglich da ROOM auch „Dispatching“ unterstützt.

Schichten

Um ROOM für sehr umfangreiche Entwürfe übersichtlich zu halten, kann man den Entwurf in verschiedene Schichten unterteilen. Die Kommunikation zwischen zwei Schichten wird mit speziellen Aktoren realisiert. Die sendende Schicht benutzt einen sogenannten Service Provision Point (SPP) und die empfangende einen Service Access Point (SAP). Die Verbindung beider Schichten erfolgt dann implizit. Auf diese Weise kann jeder Aktor den Service Point seiner Schicht wie einen ganz normalen Port nutzen (Selic, 1994).

5.5.3 Echtzeitfähigkeit

Wie sein Name bereits verrät, wurde ROOM für den Entwurf von Echtzeitsystemen entwickelt. Inwieweit ROOM tatsächlich den Anforderungen zur Modellierung von Echtzeitsystemen gerecht wird, soll im Folgenden kurz diskutiert werden:

Rechtzeitigkeit

Rechtzeitigkeit: Bei Echtzeitsystemen spielt die Rechtzeitigkeit eine besonders große Rolle. Jedoch bietet ROOM keine Möglichkeit Laufzeiten und Reaktionszeiten zu modellieren. Allerdings ist es möglich, die Worst Case Execution Time zu berechnen, um den für ein Zielsystem spezifischen Nachweis der Rechtzeitigkeit zu führen.

Zuverlässigkeit

Zuverlässigkeit: Wie bereits erwähnt, wird mit Hilfe der automatischen Codegenerierung in ROOM ein Architekturverfall vermieden, die Wiederverwendung von Aktoren und Schnittstellen ermöglicht und damit letztendlich auch die Wartbarkeit der Software erhöht. Dies trägt erheblich zur Zuverlässigkeit des entwickelten Systems bei. Weiterhin wirken sich umfangreiche Möglichkeiten für

Test und Analyse in den ROOM Werkzeuge positiv auf die Zuverlässigkeit aus.

Verteiltheit

Verteiltheit: Durch Protokolle und Verarbeitung von Signalen durch Automaten lassen sich verteilte Systeme in ROOM gut darstellen.

Nebenläufigkeit

Nebenläufigkeit: Aktoren laufen in ROOM nebenläufig ab, Nebenläufigkeit wird demnach unterstützt.

Komponierbarkeit

Komplexität: Aktoren und ROOM Charts können hierarchisch dekomponiert werden. Dadurch kann ein komplexer Softwareentwurf in mehrere kleinere zerlegt werden, die dann einzeln im Detail entworfen werden.

Dynamik

Dynamik: In ROOM können Aktoren dynamisch zur Laufzeit erstellt werden. Durch optionale Instanzen wird zeigen an wo zur Laufzeit Aktoren entstehen können. Dadurch kann die Dynamik bereits im Entwurf dargestellt werden.

Insgesamt kann festgehalten werden, dass die Sprache ROOM zur Entwicklung von Echtzeitsystemen nur bedingt geeignet ist, da sie es nicht erlaubt, Echtzeitanforderungen in Form der Rechtzeitigkeit (siehe Kapitel 3) zu modellieren. Die verfügbaren Tools schließen aber diese Lücke größtenteils, da mit ihrer Hilfe Echtzeitanforderungen teils simulativ, teils analytisch überprüft werden können.

5.6 Hardware/Software-Codesign

Die meisten modernen eingebetteten Systeme bestehen aus kooperierenden Hardware- *und* Softwareteilen. Dabei wird die Gesamtaufgabe der Systementwicklung in Teilaufgaben zerlegt, die dann entweder von Hardware- oder Softwarekomponenten bearbeitet werden. So lassen sich Teilaufgaben an die harte Echtzeitanforderungen geknüpft werden, oft nur mit Hilfe von (teuerer) Spezialhardware realisieren. Andererseits sollen Entwurfskosten bzw. -zeit sowie die Systemkomplexität eingebetteter Systeme so gering wie möglich gehalten werden.

Time-to-Market

Vor dem Hintergrund stetig schrumpfender Produktlebenszyklen eingebetteter Systeme ist dies dringend erforderlich. So gibt es beispielsweise in der Telekommunikationsindustrie Produkte, die einen Lebenszyklus von weniger als zwei Jahren besitzen. Das zwingt die Industrie, die Produkte so schnell und kostengünstig wie möglich auf den Markt zu bringen. Dadurch steigen die Time-to-Market-Anforderungen, die zu bewältigen sind, erheblich an.

Der Entwurfsprozess für solche gemischte Hardware/Software-Systeme stellt uns vor neue Herausforderungen, da zum einen die Komplexität der zu entwerfenden Systeme sehr hoch ist und zum anderen die zyklische Abhängigkeit von Hardware- und Softwarekomponenten aufgelöst werden muss. Der Einsatz von rechnergestützten Entwurfswerkzeugen ist hier notwendig.

Cosimulation

Stand der Technik ist hier lediglich die sogenannte *Cosimulation.* Bei ihr wird für eine vorgegebene Hardware die „passende" Software entwickelt und anschließend mit Ansätzen des Debugging sowie der gemischten Simulation von Hard- und Software überprüft, ob die Entwicklungsziele weitestgehend eingehalten werden konnten. Bei dieser Vorgehensweise werden Hard- bzw. Software getrennt spezifiziert. Schon zu Beginn des Entwicklungsprozesses ist es erforderlich zu entscheiden, welche Teile des Produkts in Hardware und welche in Software realisiert werden sollen. Treten zu einem späteren Zeitpunkt Unstimmigkeiten auf, können diese meist nur durch umfangreiche Änderungen in vielen Entwicklungsschritten behoben werden. Wünschenswert ist hier ein Entwicklungsmodell, welches die dedizierte und schrittweise Verbesserung von Entwicklungsschritten zulässt, sowie eine integrierte Entwicklung des gesamten eingebetteten Systems, die alle Anforderungen (hinsichtlich Funktionalität, Zeitbedingungen, Sicherheit etc.) erfüllt.

HW/SW-Codesign

Den vorgenannten Herausforderungen kann mittels Hardware/Software-Codesign entgegen getreten werden. Unter *Hardware/Software Codesign* (kurz: HW/SW-Codesign) versteht man den gemeinsamen Entwurf von Hardware- und Softwarekomponenten eines Systems.

Bei diesem Ansatz wird erst zu einem späten Zeitpunkt in der Entwicklungsphase entschieden, welche Teilbereiche eines Produktes in Hardware und welche in Software realisiert werden. Beide Teilbereiche werden bis zu diesem Zeitpunkt gemeinsam und gleichzeitig entwickelt. Ziel dieser späten Entscheidung ist die Optimierung einer Nutzen-Kosten-Zielfunktion, in welcher Anforderungen wie harte Echtzeitbedingungen genauso ihren Niederschlag finden können wie Entwicklungskosten.

Die Entwicklung solcher Systeme wirft jedoch eine Reihe neuartiger Herausforderungen des Entwurfs auf, insbesondere

Herausforderungen

- die Frage der Auswahl von Hardware- und Softwarekomponenten,
- die Partitionierung einer Spezifikation in Hard- und Software, d. h. die Abbildung von Teilen der Spezifikation auf Hardware bzw. auf Software,
- die automatische Synthese von Schnittstellen- und Kommunikationsstrukturen und
- die Verifikation und Cosimulation.

Allgemein lässt sich der Prozess des HW/SW-Codesign technologieunabhängig in folgende 3 Schritte unterteilen:

Schritte des HW/SW-Codesign

- **Modellierung:** Konzeption und kontinuierliche, möglichst schrittweise Verfeinerung der Spezifikation bis zum integrierten HW/SW-Modell
- **Validierung:** Überprüfung auf Übereinstimmung zwischen Systemanforderungen und -verhalten.
- **Implementierung:** Physikalische Realisierung der Hardware und Codeerzeugung der Software.

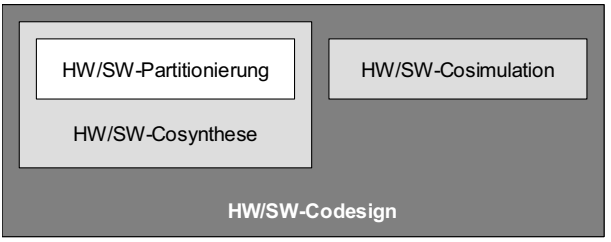


Abb. 5.5: Einordnung HW/SW-Codesign

Abbildung 5.5 zeigt eine Einordnung der Fachbegriffe. Das Kriterium bei der *Hardware/Software-Partitionierung* ist das folgende: Standardhardware auf Basis programmierbarer Mikrocontroller ist im Vergleich zu Spezialhardware billiger, rechnet aber weniger schnell. Aufgrund der immer höheren Anforderungen an die Verarbeitungsgeschwindigkeit und –vielfältigkeit haben daher die Entwickler eingebetteter Systeme immer häufiger die Entscheidung zwischen einer sehr komplexen Spezialhardware oder Standardhardware mit großem Softwareanteil

zu treffen. In diesem Zusammenhang spricht man auch vom *Hardware/Software-Tradeoff*.

Ausgewählte Ansätze des HW/SW-Codesign

Zwar hat sich bis heute noch keine Standardlösung für das HW/SW-Codesign durchgesetzt, wohl gibt es aber zahlreiche, größtenteils sehr fundierte Ansätze hierfür, von denen wir im Folgenden exemplarisch einige wenige aufgreifen werden:

- **COSYMA** Cosynthesis for Embedded Architectures: Technische Universität Braunschweig, siehe (Ernst et al., 1993).
- **VULCAN:** Stanford University, siehe (Gupta, DeMicheli, 1993).
- **SpecSyn:** University of California, Irvine, siehe (Gajski et al., 1998).
- **POLIS:** University of California, Berkeley, siehe (Balarin et al., 1997).

Partitionierung

Bei diesen Ansätzen des HW/SW-Codesigns kommen Verfahren aus dem Compilerbau gleichermaßen zum Einsatz wie solche aus der Hardwaresynthese. Von zentralem Interesse in aktuellen Forschungsarbeiten sind hier immer wieder verschiedene Aspekte *der Partitionierung.* Alternative Zielfunktionen der Partitionierung können prinzipiell die folgenden sein:

- *Verbesserung der Gesamtperformanz* einer Applikation oder einer Gruppe von Applikationen. Die wesentliche Aktion der Partitionierung besteht hierbei in einem sogenannten „Profiling“ mit typischen Datensätzen zur Abschätzung des Systemverhaltens.
- *Einhaltung harter Echtzeitanforderungen* für dedizierte Operationen. Eine weitverbreitete Strategie ist hierbei die Abbildung der gesamten Operation in Hardware, während der „Rest“ möglichst in Software bleibt, um die Systemkosten möglichst gering zu halten.
- *Einhaltung dedizierter Implementierungsgrößen*, da auch physikalische Größen der Hardware, auf die abgebildet wird, Rahmenbedingungen darstellen können. In diesem Fall wird diese Abbildung so gestaltet werden, dass jede Funktionalität in einem physikalischen Baustein integriert wird.

Schritte der Partitionierung

Die Partitionierung wird dabei in der Regel in Form der nachstehenden drei Schritte durchgeführt:

- **Allokation:** Aus gegebener HW-Bibliothek (z. B. Standardprozessoren, digitale Signalprozessoren, Mikrocontroller, Applikationsspezifische Integrierte Schaltungen, Speicher, Busse) benötigte Komponenten auswählen bzw. Synthese der Komponenten
- **Partitionierung** der funktionalen Objekte im engeren Sinne (nur Aufteilung)
- Zuordnung (**Mapping**, **Technologieauswahl**) der partitionierten Funktionalität zu HW-Komponenten

Alternativen der Partitionierung

Für die Partitionierung (im weiteren Sinne) gibt es die folgenden beiden grundsätzlichen Alternativen:

- **Strukturelle Partitionierung** (System wird erst implementiert, dann partitioniert)
- **Funktionale Partitionierung** (System wird erst partitioniert, dann implementiert)

Aufgrund ihrer algorithmischen Komplexität kann die Partitionierung meist nicht vollständig Computer-gestützt erfolgen, sondern erfordert vielmehr zusätzlich die Kreativität des Entwicklers.

Zwei aktuelle Ansätze zur funktionalen Partitionierung sind COSYMA und VULCAN (siehe auch oben). Wie Tabelle 5.1 zeigt, unterscheiden sie sich teilweise erheblich.

Umfänge des HW/SW-Codesign

Ein zukünftiger, idealtypischer Lösungsansatz für HW/SW-Codesign müsste eine Entwicklungsumgebung mit -verfahren nebst -methodik für folgende Umfänge vorsehen:

- Systemmodellierung funktionaler und nicht-funktionaler Aspekte
- Möglichkeit der Technologievorgabe
- Automatische sowie teilautomatische HW/SW-Partitionierung
- Hierarchische Analyse
- Schrittweise Verfeinerung
- Automatische HW/SW-Cosynthese und Codegenerierung

Tab. 5.1: COSYMA und VULCAN im Vergleich

	COSYMA	VULCAN
Sprache	C^x: C erweitert um Prozesskonzept und Interprozesskommunikation	HardwareC: C erweitert um Prozesskonzept und Interprozesskommunikation
Echtzeitanforderungen	Spezifikation von min. bzw. max. Laufzeiten zwischen Labels im Programm	Spezifikation von min. bzw. max Laufzeiten sowie Datenraten
Strategie	SW-orientiert: Soviel Funktionalität wie möglich (aus Kostengründen) in SW implementieren, aufwändige Programmblöcke (z. B. Schleifen) auf HW verlagern, anwendungsspezifische HW nur dort, wo andernfalls harte Echtzeitanforderungen verletzt würden.	HW-orientiert: Soviel Funktionalität wie möglich (aus Zeitgründen) in HW implementieren, SW nur dort, wo harte Echtzeitanforderungen nicht verletzt werden.
Datenaustausch	über gemeinsamen Speicher	mittels Sende- und Empfangoperationen
Zielarchitektur	1 Standardprozessor, 1 Coprozessor (ASIC), Kopplung über 1 gemeinsamen Speicher	1 Standardprozessor, mehrere ASICs, Kopplung über 1 globalen Bus
Abstraktionsebene	Programmblöcke	Programmblöcke und Operationen
Optimierungsverfahren	Simulated Annealing	Greedy Algorithmus

Eine umfangreiche kommentierte Bibliographie zu HW/SW-Codesign ist in (Buchenrieder, 1995) zu finden. Ferner beleuchten (De Micheli, 1996), (Kumar, 1995) und (Rosenblit, 1995) das Thema aus unterschiedlichen Blickwinkeln. Darüber hinaus sei dem interessierten Leser für aktuelle Detailinformationen die Lektüre der Tagungsbände (engl. proceedings) der internationalen Workshops bzw. Symposen „on Hardware/Software Co-Design“ (kurz: CODES) zu raten die bis 2002 zehnmal stattgefunden haben und seit 2003 in Kombination mit dem Thema der Systemsynthese als „International Conference on Hardware/Software Codesign and System Synthesis“ (www.codesign-symposium.org) weitergeführt werden.

5.7 Die MARMOT-Methode

Komponenten

Der am Fraunhofer-Institut für Experimentelles Software Engineering (IESE) entwickelte MARMOT (Method for Component-Based Real-Time Object-Oriented Development and Testing) Ansatz verfolgt die Idee, Systeme als Kombination verschiedener schon existierender Bauteile bzw. Komponenten zu betrachten (vgl. www.marmot-project.de). Diese Bauteile haben sich bereits in anderen Applikationen bewährt und müssen evtl. nur noch an das neue System angepasst werden. Solche Komponenten müssen nicht zwingend selbst entwickelt worden sein und können somit auch von Drittanbietern stammen.

Völlig problemlos gestaltet sich ein derartiges Zusammenbauen natürlich nicht. Objektorientierte Sprachen sind compilergebunden. Somit sind Objekte zur Laufzeit in größere Systeme eingebunden und stehen nicht zur Wiederverwendung zur Verfügung. Ein weiteres Problem ergibt sich oft aus der Vernachlässigung nichtfunktionaler Eigenschaften (Speicherplatzbedarf, Sicherheit, Performanz, Zeitverhalten). Effektive und systematische Wiederverwendung hängt jedoch von der Komponentenqualität *und* ihren nichtfunktionalen Eigenschaften ab.

MARMOT unterstützt diesen Entwicklungsansatz unter Beachtung der genannten Herausforderungen. MARMOT basiert auf der KobrA-Methode (siehe unten), erweitert diese und unterstützt gezielt die Entwicklung eingebetteter, sicherheitskritischer Echtzeit-Systeme. Solche Systeme, die aus Hard- und Software Bausteinen bestehen, werden mittels MARMOT durchgehend und systematisch beschrieben. Dabei gibt es keine Trennung der Hard- und Softwarekomponenten. Beide werden einheitlich betrachtet und beschrieben. Je nach Komponententyp (Hard- oder Software) werden bestimmte Konzepte und Dokumente zur Verfügung gestellt.

Wichtig bei der Entwicklung neuer Applikationen sind die einzelnen Komponenten. Durch deren einheitliche Entwicklungsschritte ergibt sich eine Unabhängigkeit von der Größe und Komplexität des Systems. Durch verschiedene Sichtweisen auf die einzelnen Komponenten („was“ versus „wie“) wird schon während der Entwicklung auf deren funktionale bzw. nichtfunktionale Anforderungen eingegangen. Die bereits eben erwähnten Probleme werden somit explizit berücksichtigt. Jede Komponente wird als eigenes System betrachtet, jedes System wiederum als Komponente. Komponenten werden durch ihre Spezifikation und ihre Realisierung beschrieben. Die Spezifikation beschreibt dabei die

Funktion einer Komponente, die Realisierung hingegen deren Umsetzung.

In beiden sind UML-Modelle sowie Bau- und Schaltpläne enthalten. Dies verdeutlicht nochmals den Gedanken der verzahnten Entwicklung von Hard- und Software. In einem rekursiven Ansatz werden die einzelnen Komponenten soweit verfeinert bzw. spezifiziert, bis eine bereits existierende Lösung gefunden wird. Diese kann dann als Element eingegliedert werden. Gibt es eine solche Lösung noch nicht, wird die Komponente entwickelt und kann für weitere Projekte verwendet werden. MARMOT Projekte sind hierarchisch aufgebaut, d. h. sie werden als Summe einzelner Komponenten betrachtet. Komponenten ordnen sich in das Gesamtsystem ein, können aber völlig isoliert betrachtet und entwickelt werden.

Eine derartige Entwicklungsstrategie bringt eine hohe Änderbarkeit mit sich. Somit kann schnell auf geänderte Verhältnisse oder neue Anforderungen reagiert werden. Dies gilt natürlich sowohl für die Software als auch für die Hardware. Durch das Prinzip der Wiederverwendung und der auf Sichten basierten Komponentenentwicklung wird der Qualitätssicherung schon früh im Projekt Rechnung getragen. Für das fertige Produkt und daraus hervorgehenden Weiterentwicklungen und Varianten ergibt sich eine deutliche Qualitätssteigerung. Das Baukastenprinzip mit der Entwicklung und Verwendung eigener Elemente oder kommerzieller Produkte Dritter führt zu geringen Entwicklungskosten und steigert den Grad der Wiederverwendung. Die kombinierte Entwicklung von Hard- und Software verkürzen die Zeit bis zur Marktreife des Produktes.

KobrA Methode

Die *KobrA*-Methode (Komponentenbasierte Anwendungsentwicklung) entstand im Rahmen des *KobrA* – Projekts, das vom Bundesministerium für Bildung und Forschung ins Leben gerufen wurde. Beteiligt waren die Softlab GmbH aus München als Projektleiter, die Psipenta GmbH aus Berlin und die GMD FIRST aus Berlin. Die „Komponentenbasierte Anwendungsentwicklung“ basiert auf dem *Product Line Software Engineering*, besser bekannt als PULSE, des Fraunhofer Institute *for Experimental Software Engineering* (IESE).

Prozessteile

Die Entwicklung der Systemfamilie basiert auf drei Prozessteilen:

Kontext-realisierung

(1) Innerhalb der Kontextrealisierung wird ein Unternehmensmodell mit Konzept- und Prozessdiagrammen zur Beschreibung der Geschäftsprozesse, ein Strukturmodell zur Beschreibung von

strukturellen Interaktionen mittels Klassendiagramm, ein Aktivitätsmodell bestehend aus Use Cases (Anwendungsfällen) und Aktivitätsdiagrammen, ein Interaktionsmodell bestehend aus Sequenz- und Kollaborationsdiagrammen sowie ein Entscheidungsmodell erstellt.

Application Engineering

(2) Nach der Kontextrealisierung findet das Framework Engineering für die Systemfamilie oder das Application Engineering für ein Familienmitglied statt. Das Framework enthält die Referenzarchitektur der Systemfamilie, die das Ableiten einer Applikation aus den gewünschten Komponenten erlaubt. Auch das Framework selbst stellt eine Komponente dar, wodurch KobrA als rekursiver Prozess bezeichnet wird, d. h. alles wird als Komponente behandelt und kann wiederum aus Komponenten zusammengesetzt sein.

Implementierung

(3) Im dritten Teil der KobrA-Methode findet die Implementierung der Komponenten statt. Zum einen können bereits existierende Komponenten genutzt oder, falls existierende Komponente den gegebenen Anforderungen nicht entsprechen, müssen entsprechende Komponenten neu implementiert werden.

Trennung von Produkten und Prozessen

Das zentrale Konzept von *KobrA* ist die Trennung von Produkten und Prozessen. Die Produkte stellen dabei die Ergebnisse der Entwicklung dar und werden unabhängig von den Prozessen und zeitlich davor definiert. Alle Produkte zielen auf die Beschreibung von individuellen Komponenten ab, in denen die Variabilität der Familie enthalten ist.

UML Use Cases

Die Phase des *Requirements Engineering* in KobrA basiert auf textuellen Beschreibungen und Use Case Diagrammen der UML, die Variationen über Stereotypen modellieren. Die Variationen der einzelnen Komponenten werden in der Tabelle vermerkt, indem für jede Variante der Familie die benötigten Merkmale eingetragen werden, was in KobrA als *Product Map* bezeichnet wird. Für die spätere Ableitung von Applikationen wird in KobrA noch ein *Decision Model* erarbeitet, in dem alle Variationspunkte enthalten sind, wobei hier eine Ähnlichkeit zu den Merkmalmodellen von Kang besteht, ohne dass die Merkmalmodelle von KobrA genutzt werden. Vor der Ableitung einer Variante muss für jeden Variationspunkt entschieden werden, ob er im System vorhanden ist. Darüber hinaus sind auch Parameter als Variationspunkte möglich, wobei diese für eine konkrete Applikation mit einem Wert belegt werden müssen. Im *Decision Model* sind eine Reihe von Fragen enthalten, die der Kunde zur Ableitung einer Applikation beantworten muss. Stellt der Kunde eine Anforderung, die nicht als Komponente in der Systemfamilie enthalten ist, muss diesem

Wunsch durch Integration einer neuen Komponente in die Familie nachgekommen werden. Dabei werden wenn möglich Ähnlichkeiten zu bereits existierenden Komponenten der Familie ausgenutzt. Die Interaktion der einzelnen erfragten Variabilitäten des *Decision Models*, als *Effect* bezeichnet, sind ebenfalls ein textueller Form im Modell hinterlegt und müssen vom Entwickler aufgelöst werden.

Zusammenfassung KobrA

KobrA stellt eine umfassende Methode zur Entwicklung von Systemfamilien dar. Das von KobrA genutzte *Decision Model* zur Erfassung und Verarbeitung von Variabilität in der *Requirements Engineering* Phase stellt eine gute Basis für die Ableitung eines Familienmitglieds dar, kann jedoch mit der Übersichtlichkeit und Verständlichkeit der Merkmalmodelle nicht konkurrieren. Das *Decision Model* kann nicht automatisiert verarbeitet werden, insbesondere müssen die enthaltenen Auswirkungen, sog. *Effects,* manuell durch den Entwickler überprüft werden.

5.8 Hybride Systeme und hybride Automaten

Einordnung

Hybride Systeme gehören zur Klasse der eingebetteten Echtzeitsysteme. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Systeme werden dann als *hybrid* bezeichnet, wenn deren Eingabewerte nicht nur in digitaler Form vorliegen, sondern auch in analoger Form. Die analogen Werte bestehen dabei nicht aus einer Menge von definierten Werten, sondern sie verändern sich – wie in der Natur üblich – stufenlos. Beispiele für analoge Werte sind Temperatur, Neigungswinkel usw. Hybride Systeme kommen dann zum Einsatz, wenn mit der Umwelt interagiert wird. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Viele eingebettete Systeme sind hybride Systeme, da sie neben kontinuierlich arbeitenden Komponenten (Schnittstellen zur Umwelt) auch welche, die auf diskreter Basis (Steuerungen) funktionieren, beinhalten.

5.8.1 Einleitung

Beispiele

Beispiele für hybride Systeme sind Systeme wie das Antiblockiersystem (ABS), bei dem abhängig vom Rollen oder Gleiten des Reifens unterschiedlich von der Software eingegriffen

werden muss. Darüber hinaus spielen hybride Systeme angesichts der Flexibilität, die Software-basierte Steuerungen bieten und angesichts der sinkenden Hardwarepreise eine immer wichtigere Rolle in Forschung und Industrie. Einsatzgebiete für hybride Systeme sind z. B. Automatisierungstechnik, Fahrassistenzsysteme im Automobilbereich, Flugleitsysteme und Robotik. Einige Arbeiten in diesem Gebiet beschäftigen sich etwa mit Greifprozessen von Roboterhänden oder fußballspielenden Robotern.

Funktionen eines eingebetteten Systems wie etwa das Ein- und Ausschalten von Motoren oder Öffnen und Schließen von Ventilen zählen zu den ereignisdiskreten dynamischen Systemen. Bei der Untersuchung ereignisdiskreter dynamischer Systeme zeigte sich, dass bei ihnen ganz andere Probleme als bei kontinuierlichen Systemen auftreten. Die Ursache hierfür liegt in der Begrenztheit der Werte, die diskrete Signale und Systeme annehmen können.

In technischen Systemen interagieren die ereignisdiskreten Teile mit kontinuierlichen dynamischen Prozessen, indem sie z. B. Stelleingriffe ausführen oder kontinuierliche Prozesse starten oder beenden. Die durch dieses Zusammenspiel entstehenden Systeme bezeichnet man als hybride dynamische Systeme.

Signalfluss in Systemen

Signalfluss in Systemen:
Im System können die Informationen (Ein-/Ausgangsignale) auf drei verschiedene Arten fließen (siehe auch Abbildung 5.5):

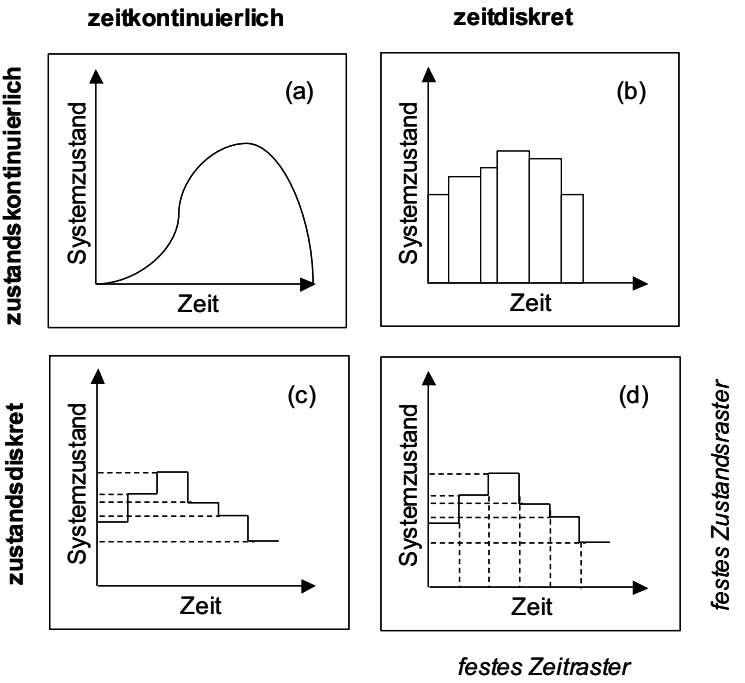
- **Kontinuierlich:** Informationen fließen in einem ununterbrochenen Strom (analog). Die meisten Prozesse aus der Natur, Physik und Technik sind kontinuierlicher Natur. Sie können durch Differentialgleichungen beschrieben werden.
- **Diskret:** Informationen fließen in (un-)regelmäßigen Abständen(digital). Die Elektronik und damit auch Computer arbeiten mit diskreten Informationen.
- **Hybrid:** Kombination aus kontinuierlichen und diskreten Informationsflüssen.

Die ersten beiden Arten werden von Wissenschaft und Industrie in weiten Teilen gut beherrscht. Die dritte Variante, die hybriden Informationsflüsse, stellt derzeit noch große Herausforderungen an die Informatik.

Kontinuierliche Signale können durch Abtastung in diskrete (quasi kontinuierliche) und diskrete Signale durch Interpolation in kontinuierliche umgewandelt werden. Neben diesen quasi kontinuierlichen, zeitdiskreten Signalen spielen ereignisdiskrete,

asynchron auftretende binäre Signale eine große Rolle, so beispielsweise in Steuerungen.

Abb. 5.5: Klassifikation Signalfluss



a) Kontinuierliche Systeme
b) Systeme diskreter Ereignisse
c) Diskrete Systeme
d) Ganzzahlige zeitdiskrete Systeme

Hybride Systeme haben häufig Regelungsaufgaben zu erfüllen. Standardmethoden der Regelungstechnik beschäftigen sich mit linearen Systemen. Lineare Systeme reagieren auf eine Linearkombination der Eingaben mit einer Linearkombination der Ausgaben. Hybride Systeme sind meist unstetig (Stauner, 2001). Es bedarf daher angepasster Ansätze für deren Spezifikation.

5.8.2 Spezifikation hybrider Systeme

Graphische Formalismen

Im Folgenden wollen wir, ohne Anspruch auf Vollständigkeit, einige Formalismen zur graphischen Spezifikation von hybriden Systemen kurz angeben.

Hybride Petri-Netze

Hybride Petri-Netze (David, 1997), (DiFebbraro, 2001):
In hybriden Petri-Netzen sind Token nicht mehr unteilbar. Sie können kontinuierlich fließen. Die Fließgeschwindigkeit der Token wird an speziellen „analogen" Knoten textuell durch Differentialgleichungen angegeben. Eine umfangreiche Online-Bibliographie zu hybriden Petri-Netzen kann unter bode.diee.unica.it/~hpn/ nachgeschlagen werden.

Hybride Statecharts

Hybride Statecharts (Kesten, Pnueli, 1992):
Sie stellen eine Erweiterung von Statecharts dar. Die Modellierung von kontinuierlicher Zeit geschieht über die Definition von Zeitintervallen für Transitionen. Jeder Transition ist ein Zeitintervall zugewiesen, welches obere und untere Grenzen für diese Transition festlegt. Zustände können mit Differentialgleichungen versehen werden, die kontinuierliche Änderungen beschreiben, solange das System in diesem Zustand ist. Der Vorteil dieses Ansatzes liegt in der Möglichkeit der hierarchischen Strukturierung.

HyCharts

HyCharts (Grosu, Stauner, 2002):
Sie sind eine Erweiterung des Modells der hybriden Statecharts. In HyCharts ist es möglich, auch die hierarchische Zerlegung eines Systems in einzelnen Teilen darzustellen.

Hybride Automaten

Hybride Automaten (Henzinger, 1996):
Sie sind eine Erweiterung von endlichen Automaten. Anders als diese enthalten sie sowohl diskrete als auch kontinuierliche Anteile. Der Zustand eines hybriden Automaten setzt sich aus „Ort" (engl. location) und „Variablenbelegung" (engl. valuation) zusammen. Solange der Automat in einem Ort ist, verändern sich die Variablen anhand vorgegebener Differentialgleichungen (DGL). Die Modellierung kontinuierlicher Änderungen geschieht in hybriden Automaten durch DGL in den Zuständen des Automaten. Diskrete Zustandsänderungen werden mit Hilfe von Transition vollzogen. Bei Benutzung einer Transition sind Zuweisungen an Variablen möglich.

Um dem Leser einen Eindruck von der Komplexität und den Herausforderungen der Modellierung hybrider Systeme zu verschaffen, wollen wir im Folgenden hybride Automaten ansatzweise analysieren.

Definition (hybrider Automat)

Definition (hybrider Automat) nach (Henzinger, 1996):
Ein hybrider Automat A ist definiert durch ein 10-Tupel (X, V, inv, init, flow, E, upd, jump, L, sync) =: A wobei:

- **X:** Endliche, geordnete Menge x_1, x_2, ..., x_n von Variablen aus den reellen Zahlen
- **V:** Endliche Menge von Orten (engl. locations). Bei hybriden Automaten sind Zustand und Ort nicht gleichbedeutend, wie das bei endlichen Automaten der Fall ist. Hier ist der Zustand durch das Tupel (v,s), bestehend aus einem Ort v und einer Belegung s charakterisiert.
- **inv:** Abbildung, die jedem Ort ein Prädikat als Invariante zuweist. In einem Ort v sind nur Belegungen s aus inv(v) erlaubt.
- **init:** Abbildung, die jedem Ort ein Prädikat als Anfangsbedingung (engl. initial condition) zuweist
- **flow:** Abbildung, die jedem Ort ein Prädikat (engl. flow condition) über $X \cup X' := \{x_1, x_2, ..., x_n, x'_1, x'_2, ..., x'_n\}$ zuweist. Sie beschreibt die Veränderung einer Variablen während sich der hybride Automat in einem Ort befindet. Hier sind nur DGL ersten Grades erlaubt. Damit wird das modellierbare Systemverhalten auf kontinuierliche Veränderungen beschränkt.
- **E:** Menge von Transitionen (engl. edges) zwischen den Orten. Die Übergänge werden jeweils durch einen Anfangs- und End-Ort dargestellt. Es sind auch mehrere Übergänge zwischen zwei Orten erlaubt.
- **upd:** Abbildung, die jedem Übergang eine Menge derjenigen Variablen zuweist, denen neue Werte zugewiesen werden (engl. update set). Bei einem Übergang können sich nur die Variablen verändern, die in der Update-Menge sind.
- **jump:** Abbildung, die jeder Transition ein Prädikat zuweist, welches erfüllt sein muss, damit die Transition gefeuert wird (engl. jump condition)
- **L:** Menge der Labels für die Transitionen. Sie stellt die endliche Menge von Synchronisationsbezeichnungen dar. Bei parallelen Automaten werden Übergänge mit gleicher Bezeichnung, d. h. mit gleichem Label gleichzeitig ausgeführt.

- **sync:** Abbildung, die jedem Übergang ein Label zuordnet. Jedem Übergang wird damit eine Synchronisationsbezeichnung zugeordnet.

Beispiel (Hybrider Automat) nach (Henzinger, 1996):
Nachstehende Abbildung 5.6 zeigt einen hybriden Automaten. Hierbei handelt es sich um die Steuerung eines Termostates zur Temperaturregelung eines Raumes. Bei einer angenommenen Anfangstemperatur von 20 Grad (Celsius) ist die Heizung zunächst aus. Im Zustand „Aus" fällt die Raumtemperatur aufgrund der gegebenen Umwelteinflüsse kontinuierlich linear um 0,1 Grad pro Zeiteinheit. Fällt sie unter 19 Grad, so findet ein diskreter Zustandsübergang von „Aus" nach „An" statt, d. h. die Steuerung des Termostates schaltet die Heizung ein. Da auch bei eingeschaltetem Zustand der Heizung noch die selben Umweltbedingungen gelten wie im ausgeschalteten, kühlt die Umwelt den Raum immer noch um 0,1 Grad pro Zeiteinheit. Diesem Kühlungseffekt wirkt nun aber eine „Heizleistung" von 5 Grad entgegen. Der vorgegebene hybride Automat in Abbildung 5.6 besitzt keine Labelmenge „L" und damit folglich auch keine Synchronisationsabbildung „sync"; diese wären nur bei Synchronisation mit anderen, parallel gekoppelten, hybriden Automaten erforderlich.

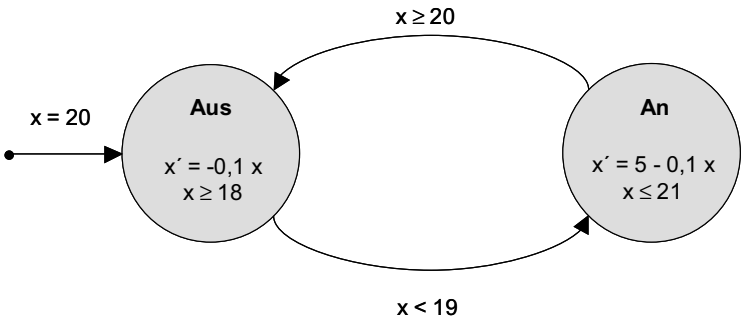


Abb. 5.6: Hybrider Automat (Termostat),vgl. (Henzinger, 1996)

Formale Verifikation, Model Checking

Die *formale Verifikation* von mit Hilfe hybrider Automaten spezifizierter hybrider Systeme ist durch Einsatz von Model Checking prinzipiell möglich. Hier kann beispielsweise das Softwarewerkzeug „Hytech" (Henzinger, 1996) Anwendung finden. Es erlaubt den Nachweis von sicherheitsrelevanten Eigenschaften, die insbesondere dadurch gekennzeichnet sind, dass sie nicht oder nur bei bestimmten Bedingungen eintreten sollen. Beispiel: „Alle Ampeln einer Kreuzung grün".

Entscheidbarkeit der Erreichbarkeit

Die Möglichkeit der formalen Verifikation von Sicherheitseigenschaften ist dabei jedoch stark eingeschränkt, da bereits die *Erreichbarkeit* von Systemzuständen nur für bestimmte Klassen hybrider Automaten entscheidbar ist. So ist zwar beispielsweise das Erreichbarkeitsproblem für die Klasse der sogenannten „Zeit-Automaten“ sowie für die Klasse der „einfachen Multirate-Zeit-Automaten“ entscheidbar, im Allgemeinen nicht aber für die Klassen der „linearen hybriden Automaten“ oder der „einfachen hybriden Automaten“. In keinem Fall entscheidbar ist das Erreichbarkeitsproblem ferner für „2-rate Zeit-Automaten“ sowie „einfache Integrator-Automaten“. Die vorgenannten Klassen hybrider Automaten unterscheiden sich im Wesentlichen durch die Form der Prädikate in „inv“, „init“, „flow“ und „jump“. Eine exakte Definition ist (Henzinger, 1996) zu entnehmen.

Die Verifikation eingebetteter Systeme ist wegen des erforderlichen Aufwands für die formale Modellierung zum einen, vor allem aber durch den exponentiell wachsenden kombinatorischen Aufwand schwierig. Für mittelgroße Systeme ist dies jedoch unter Einsatz von Model Checking bereits gut durchführbar. Die Verifikation hybrider Systeme kann ein noch viel komplexeres Problem darstellen. Es können Grenzzyklen, Gleitbewegungen, evtl. sogar chaotische Dynamik auftreten. Zurzeit erscheint der Ansatz der kompositionalen Analyse am erfolgversprechendsten; dabei werden gewisse Eigenschaften auf Teilsystemebene formal nachgewiesen und daraus das Verhalten des gesamten Systems gefolgert.

Vorteile

Hybride Automaten besitzen folgende *Vorteile:*

- Diskrete und kontinuierliche Anteile des Systems sind leicht modellierbar.
- Einfache Systeme sind einfach zu verstehen.
- Die formale Verifikation ist teilweise automatisierbar.

Nachteile

Sie weisen jedoch andererseits auch nachstehende *Nachteile* auf:

- Keine Hierarchie; komplexe Systeme werden ggf. unübersichtlich.
- Verifikationsalgorithmen sind sehr rechen- und speicherintensiv.
- Die Terminierung der Verifikationsalgorithmen ist bei vielen Klassen von hybriden Automaten nicht garantiert.
- Die Verifikation einzelner Module ist nicht möglich; nur das Gesamtsystem kann verifiziert werden.

5.9 Zusammenfassung

Die Entwicklung der Software eingebetteter Systeme ist eine komplexe Aufgabe, die derzeit noch nicht in allen Teilen zufriedenstellend beherrscht wird. Um die korrekte Funktionsweise solcher Systeme und kurze Entwicklungszeiten zu gewährleisten, ist es erforderlich, den Systementwickler in allen Phasen der Entwicklung möglichst gut methodisch und technisch durch ein passendes Werkzeug zu unterstützen.

Der Mensch kann aufgrund seiner visuellen Prägung Spezifikationen in Form von Diagrammen tendenziell besser verstehen und umsetzen als dies bei Beschreibungen in Textform der Fall ist. Aus diesem Grunde haben sich mehr und mehr graphische Beschreibungstechniken zum Entwurf von Softwaresystemen durchgesetzt. Einige hiervon (Statecharts, UML, ROOM usw.) haben wir in diesem Kapitel kennen gelernt.

Die getrennte Entwicklung von Hardware und Software führte in der Vergangenheit dazu, dass schon bei der Spezifikation und dem Entwurf berücksichtigt werden musste, welche Teile des Entwurfs letztendlich in Hardware und welche in Software realisiert werden sollen. Diese frühe Entscheidung war oft nur sub-optimal. Wir haben an dieser Stelle den Ansatz des HW/SW-Codesigns diskutiert, der hier Abhilfe schafft.

Hybride Systeme gehören zur Klasse der Echtzeitsysteme. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Ziel des vorangegangen Abschnitts war es, dem Leser einen Einblick in das Feld der hybriden Systemem zu geben und einige wichtige Ansätze kurz vorzustellen.

6 Softwarequalität eingebetteter Systeme

Der Qualität von eingebetteter Software eine besondere Bedeutung zuzumessen, ist eine lohnenswerte Investition. Dies belegen zahlreiche Beispiele, bei denen fehlerhafte Software zu Schädigungen von Maschinen oder gar Menschen geführt hat. Einige prominente Fälle, wo Softwarefehler zu massiven Konsequenzen geführt haben, schildern wir daher in diesem Kapitel, bevor wir dann zentrale Begriffe der Softwarequalität diskutieren. Um die Softwarequalität überprüfen zu können, sind spezielle Prüftechniken erforderlich. Wir geben zunächst eine Übersicht über derzeit mögliche Prüftechniken und schildern dann ausgewählte Vertreter etwas näher.

Kapitelübersicht

Die Qualitätssicherung eingebetteter Systeme, gerade auch in der Automobiltechnik oder Avionik ist eine schwierige aber gleichwohl dringend erforderliche Entwicklungsaufgabe. Nach diesem Kapitel sollte der Leser die beiden Begriffe Zuverlässigkeit und Sicherheit sowie die damit in Bezug stehenden Begriffe verstanden haben und Verfahren zu ihrer Herstellung kennen.

6.1 Motivation

Beispiel Softwarefehler Airbus A-320

Ein „eindrucksvolles“ Beispiel für einen Softwarefehler ist die Bruchlandung eines *Airbus A-320* auf dem Warschauer Flughafen am 14. September 1993: Ein Lufthansa-Airbus fängt bei der Landung in Warschau Feuer. Bei dem Unfall sterben zwei Menschen, 54 werden verletzt. Ursache war eine Fehlkonstruktion des Sensors zur Erkennung der Bodenberührung: Im „Flight Mode“ ließ sich die zum Bremsen notwendige Schubumkehr nicht einschalten. Hier handelte es sich um keinen Pilotenfehler, sondern

um falsche Entwurfsentscheidungen der Konstrukteure und Software-Ingenieure.

Beispiel SW-Fehler Sojus-Kapsel

Ein anderes Beispiel, wo ein Softwarefehler beinahe zu einer Katastrophe geführt hatte, war die Landung einer in einer *„Sojus"-Kapsel* zurückgekehrten ISS-Mannschaft im Jahre 2003. Nach russischen Angaben führte ein Softwarefehler zu einem absturzartigen Wiedereintritt, bei dem ein vom Computer falsch berechneter Eintrittswinkel zu einer übermäßigen Hitzeentwicklung geführt hat.

Beispiel SW-Fehler Ariane 5

Einer der wohl bekanntesten Repräsentanten für mangelnde Softwarequalität ist der Absturz bzw. die ferngelenkte Zerstörung der europäischen Trägerrakete *Ariane 5* auf ihrem Jungfernflug am 4. Juni 1996 in Kourou. Die Rakete war mit vier Satelliten bestückt. Etwa 37 Sekunden nach dem Start erreichte die Ariane 5 eine Horizontalgeschwindigkeit von mehr als 32.768 internen Einheiten. Die in der Programmiersprache Ada realisierte Konvertierung dieses Wertes in eine vorzeichenbehaftete Integer-Variable führte daher zu einem Überlauf, der nicht abgefangen wurde, obwohl die Hardware redundant ausgelegt war. Da es sich hier jedoch um einen reinen Softwarefehler handelte, blieb diese Redundanz wirkungslos. Folglich wurden Diagnosedaten zum Hauptrechner geschickt, die dieser als Flugbahndaten interpretierte, so dass dieser unsinnige Steuerbefehle generierte. Die Rakete drohte daraufhin zu bersten und musste ferngelenkt gesprengt werden, um größeren Schaden aufgrund der noch niedrigen Höhe zu vermeiden. Interessanterweise war die Software von der Ariane 4 wiederverwendet worden und hatte dort auch problemlos funktioniert. Der Gesamtschaden belief sich auf 500 Millionen US-Dollar.

6.2 Begriffe

SW-Qualität

Der Begriff der Softwarequalität ist nach der Norm ISO/IEC 9126 wie folgt definiert:

Definition (Softwarequalität) nach ISO/IEC 9126:
Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Der gesamte Standard ISO/IEC 9126 umfasst vier Teile, nämlich 9126-1 bis 9126-4.

Die eigentliche, konkrete Beurteilung der Qualität geschieht mit Hilfe von *Qualitätsmerkmalen.* Sie stellen Eigenschaften einer Funktionseinheit dar, anhand derer ihre Qualität beschrieben und beurteilt werden, die jedoch keine Aussage über den Grad der Ausprägung enthalten. Ein Qualitätsmerkmal kann ggf. hierarchisch in mehrere Teilmerkmale verfeinert werden (Liggesmeyer, 2002).

SW-Qualitätsmerkmale

Die ISO/IEC 9126 unterscheidet beispielsweise zwischen den folgenden *Softwarequalitätsmerkmalen:*

- Funktionalität,
- Zuverlässigkeit,
- Benutzbarkeit,
- Effizienz,
- Änderbarkeit und
- Übertragbarkeit.

Zwischen Qualitätsmerkmalen können Wechselwirkungen und Abhängigkeiten bestehen. Die Forderung nach einer insgesamt in jeder Hinsicht bestmöglichen Qualität ist daher unsinnig (Liggesmeyer, 2002).

Failure, Fault, Error

In der deutschen Sprache spricht man gerne etwas ungenau von einem „Softwarefehler". Richtig ist es aber vielmehr, zwischen den Begriffen „Fehlverhalten", „Fehler" und „Irrtum" zu unterscheiden (siehe DIN 66271). Auch im Englischen existieren hier drei unterschiedliche Ausdrücke:

- **Failure:** Es handelt sich um ein Fehlverhalten eines Programms, das während seiner Ausführung tatsächlich auftritt *(Fehlverhalten, Fehlerwirkung, äußerer Fehler).*
- **Fault:** Es handelt sich um eine fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann *(Fehler, Fehlerzustand, innerer Fehler).*
- **Error:** Es handelt sich um eine fehlerhafte Aktion, die zu einer fehlerhaften Programmstelle führt *(Irrtum, Fehlhandlung).*

Qualitätssicherung

Auf Basis dieser Unterscheidung können wir folgende Feststellungen ableiten: Fehler (errors) bei der Programmentwicklung können zu Fehlern (fault)s in einem Programm führen, die ihrerseits Fehler (failure) bei der Programmausführung bewirken können. Die *konstruktive Qualitätssicherung* reduziert menschliche Fehler (errors). Die *analytische Qualitätssicherung* entdeckt Programm-

fehler (faults). *Testen* löst Laufzeitfehler aus (failures) und führt zur Entdeckung von Programmfehlern (faults).

Korrektheit

Neben dem Fehlerbegriff und den in der ISO/IEC 9126 aufgelisteten Softwarequalitätsmerkmalen steht die Definition des Qualitätsmerkmals der *Korrektheit* besonders im Vordergrund.

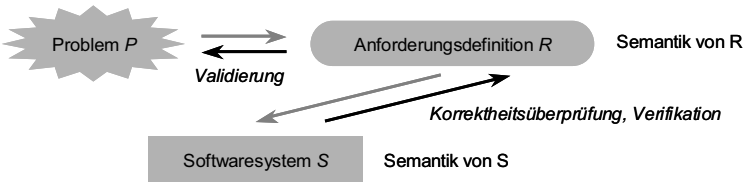
Bei (Endres, 1977) wird die Korrektheit erstmals als Synonym für Fehlerfreiheit als die Übereinstimmung zwischen dem beobachteten und dem gewünschten Verhalten definiert: „... ein Programm ist korrekt, wenn es frei ist von logischen Fehlern, d. h. wenn seine Implementierung übereinstimmt mit einer mehr oder weniger expliziten Spezifikation dessen, was das Programm tun soll."

Die IEEE definiert (vgl. ANSI 72983) Korrektheit als den Grad der Konsistenz zwischen Spezifikation und Programm bzw. als Grad der Erfüllung der Benutzererwartung durch ein Programm (Liggesmeyer, 2002).

Verifikation, Validierung

Die eben genannten Definitionen sind eher praxisorientiert als für eine optimale formale und damit „scharfe" Definition des Begriffs geeignet. Beispielsweise wird der Grad der Erfüllung der Erwartungen eines möglichen Benutzers in erheblicher Weise von der Auswahl der Befragten beeinflusst. Darüber hinaus können bereits die Programmspezifikation und die tatsächliche Benutzererwartung differieren. Aus diesem Grund unterscheidet man beim Nachweis, ob ein Programm mit seiner Spezifikation bzw. den Benutzeranforderungen übereinstimmt zwischen den beiden Begriffe der Verifikation bzw. der Validierung. Die *Verifikation* überprüft, ob das Programm mit seiner (formalen) Spezifikation übereinstimmt, die *Validierung* (oder Validation) stellt dagegen fest, ob die Spezifikation und die tatsächlichen Benutzeranforderungen identisch sind. Abbildung 6.1 gibt einen Überblick.

Abb. 6.1: Einordnung der Begriffe Korrektheit, Verifikation, Valididerung



In der Informatik hat sich darum der folgende Korrektheitsbegriff etabliert (Liggesmeyer, 2002):

- Korrektheit besitzt keinen graduellen Charakter, d. h. eine Software ist entweder korrekt oder nicht korrekt.

- Eine fehlerfreie Software ist korrekt.
- Eine Software ist korrekt, wenn sie konsistent zu ihrer Spezifikation ist.
- Existiert zu einer Software keine Spezifikation, so ist keine Überprüfung der Korrektheit möglich.

Fehlerfreie Softwaresysteme gibt es derzeit nicht und wird es mit hoher Wahrscheinlichkeit auch in mittelfristiger Zukunft nicht geben, sobald die Software einen gewissen Komplexitätsgrad überschreitet. Ein Fehlerzustand liegt oftmals darin begründet, dass sowohl während der Softwareentwicklung als auch bei der Überprüfung auf Fehler an bestimmte Ausnahmesituationen nicht gedacht wurde und dann auch nicht überprüft wurden (Spillner und Linz, 2003).

Prüftechniken

Im Folgenden wollen wir nun einige Techniken kennen lernen, welche in der Lage sind, größtenteils durch maschinelle Unterstützung, eine Überprüfung eines Programms hinsichtlich seiner Korrektheit durchzuführen. Diese Techniken werden im Allgemeinen als *Prüftechniken* bezeichnet. Bevor wir einige wenige, aber besonders gebräuchlich herausgreifen werden, wollen wir zunächst ein (mögliches und anerkanntes) Klassifikationsschema für Software-Prüftechniken angeben, das von (Liggesmeyer, 2002) vorgeschlagen wurde (vgl. Abbildung 6.2).

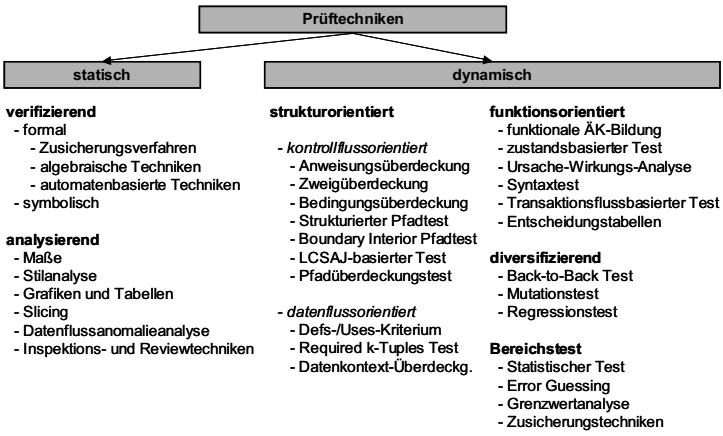


Abb. 6.2: Klassifikation von Software-Prüftechniken nach (Liggesmeyer, 2002)

Dieses Klassifikationsschema unterteilt in zwei Klassen, die statischen und die dynamischen Techniken. Sie unterscheiden sich dadurch, ob das zu überprüfende Programm zur Ausführung

kommen muss (dynamisch) oder nicht (statisch). Die meisten der dynamischen Prüftechniken besitzen eine hohe Praxisrelevanz. Besonders wichtig sind hierbei die funktionsorientierten sowie einige der kontrollflussorientierten Techniken. (Liggesmeyer, 2002).

6.3 Zuverlässigkeit eingebetteter Systeme

Neben der Echtzeitfähigkeit gilt die *Verlässlichkeit (bzw. Zuverlässigkeit oder Fehlertoleranz)* als die wichtigste nicht-funktionale Eigenschaft eingebetteter Systeme. Der Mensch versucht tendenziell, Sicherheit und Zuverlässigkeit durch Automatisierung zu erhöhen, obgleich die Statistik hier ein anderes Bild zeichnet (Schürmann, 2001): Etwa viermal so viele Unfälle passieren, wenn der Mensch *nicht* mehr in die Steuerung eingreifen kann.

Beispiel Flugzeuge

Unternehmen der Flugzeugindustrie (EADS, Honeywell, Boing) und US-amerikanische Behörden (NASA, DoD) planen derzeit die Entwicklung eines Notsystems, das verhindern soll, dass Verkehrsflugzeuge absichtlich oder unabsichtlich an Hindernissen zerschellen: Der Autopilot würde anstelle des Piloten in Gefahrensituation die Steuerung vollständig übernehmen, ohne dass dieser eingreifen könnte (Spiegel, 2003). Der Einsatz ist beispielsweise im Airbus A380 geplant, welcher das bis dato größte Verkehrsflugzeug werden soll. Ein derart autarkes System, das in Notsituationen zuverlässig das Steuer von Passagierjets übernimmt und sie aus dem Gefahrenbereich steuert, gilt als sehr erfolgsversprechend, denn auch ohne Bedrohung durch Terrorismus liegen nach wie vor den meisten Abstürzen Navigationsfehlern zugrunde.

Obwohl sich die Testergebnisse erster Entwicklungen als positiv erwiesen haben, gibt es sowohl unter Fluggästen, Flugsicherheitsbehörden als auch Piloten eine weit verbreitete Skepsis, die Kontrolle über das Flugzeug in Krisensituationen vollständig einem Computersystem (einem eingebetteten System) zu überlassen. Sie fordern die Möglichkeit, dass Crewmitglieder das System ausschalten können, womit es freilich zur Vermeidung von Terrorakten unverwertbar wäre.

Auch wenn ca. 80% aller Flugzeugkatastrophen durch menschliches Versagen verursacht werden (Schürmann, 2001), gibt es keine Statistik, die besagt, wie oft der Pilot ein Versagen der Technik korrigiert bzw. beherrscht.

Technisch gesehen wäre die Einführung dieses Sicherheitssystems ein eher marginaler technologischer Entwicklungsschritt. Bereits heute ist der Einsatz von Autopiloten bei Start, Navigation und Landung unter minimaler menschlicher Beteiligung Routine. Die neue Technologie würde dies weiterführen.

Mensch-Maschine-Kooperation

Besser wäre möglicherweise der Einsatz eines *Mensch-Maschine-Teams,* bei dem jeder das macht, was er besser kann: Eingebettete Computersysteme regeln die technische Umgebung in Standardsituationen und der menschliche Benutzer ist für die Behandlung von Ausnahmen und unvorhersehbaren Situationen zuständig. Natürlich begehen Menschen ständig Fehler, machen aber keine Aufgabe zweimal identisch. Eine Maschine dagegen begeht zwar seltener Fehler, ist aber im Fehlerfall nicht in der Lage, zu korrigieren. Menschliche Fehler können durch Plausibilitätsprüfungen erkannt werden, Maschinenfehler dagegen durch den sprichwörtlichen "gesunden Menschenverstand". Durch den Einsatz jeder Sicherheitsmaßnahme kann sich neues Gefahrenpotential ergeben, so beispielsweise bei einem Bahnübergang: Zwar erhöht der Einsatz von Schranken am Bahnübergang die Sicherheit, aber ohne Schranken besteht nicht die Gefahr, dass ein Fahrzeug zwischen den Schranken liegen bleibt. Darüber hinaus mag man bei beschrankten Bahnübergängen eher als bei unbeschrankten dazu neigen, diesen unachtsam zu überqueren, weil man sich auf das System verlässt. Wir wollen im Folgenden den Terminus der Zuverlässigkeit und verwandte Begriffe analysieren.

Definition (Zuverlässigkeit)

Definition (Zuverlässigkeit):
Zuverlässigkeit (engl. reliability) ist die Wahrscheinlichkeit, dass ein System seine definierte Funktion innerhalb eines vorgegebenen Zeitraums und unter den erwarteten Arbeitsbedingungen voll erfüllt, das heißt intakt ist und es zu keinem Systemausfall kommt.

Schließlich ist neben der Zuverlässigkeit auch die *Verfügbarkeit* ein wichtiges Gütemaß. Vom Begriff der Zuverlässigkeit lässt sich die Verfügbarkeit wie folgt abgrenzen:

Definition (Verfügbarkeit)

Definition (Verfügbarkeit):
Die Verfügbarkeit (engl. availability) eines Systems ist der Zeitraum gemessen am Anteil der Gesamtbetriebszeit des Systems, in dem es für den beabsichtigten Zweck eingesetzt werden kann.

Definition (Systemausfall)

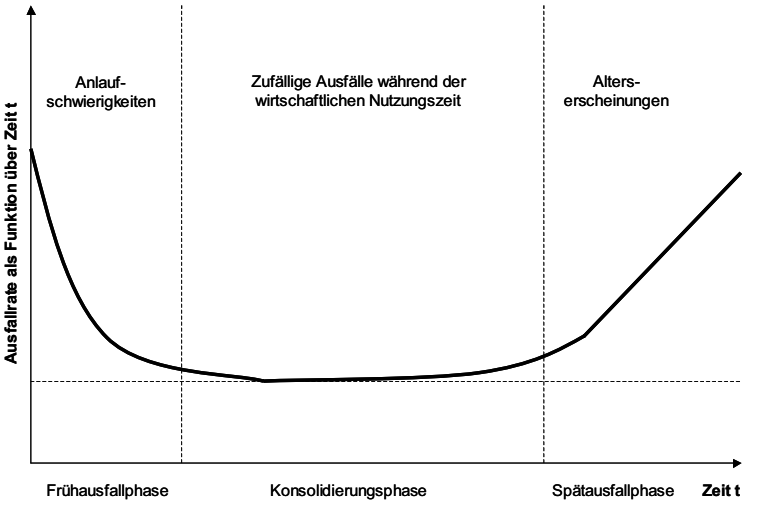
Definition (Systemausfall):
Ein Systemausfall (kurz: Ausfall, engl. failure) liegt vor, wenn ein System seine geforderte Funktion nicht mehr erfüllt.

Mit Ausfall ist hier zur Vereinfachung im Folgenden stets der Totalausfall des Systems gemeint. Wir betrachten keine Zwischenstufen der Funktionsfähigkeit (sogenannte *Änderungsausfälle*). Ein Beispiel hierfür wäre bei einem Fernseher der Tonausfall obwohl die Bildübertragung weiterhin einwandfrei funktioniert (Fränzle, 2002).

Ausfallrate, Badewannenkurve

Betrachtet man die *Ausfallsrate* (= Maß für die Anzahl von Ausfällen pro Zeiteinheit) einer elektronischen Komponente entlang ihrer zeitlichen Nutzung, so ergibt sich die sogenannte *„Badewannenkurve“,* deren Verlauf an den Querschnitt einer Badewanne erinnert; vgl. Abbildung 6.3. Ihr Name rührt daher, dass die Ausfallwahrscheinlichkeit einer Komponente am Anfang ihrer Nutzung sehr hoch ist, dann stark abfällt, um schließlich gegen Ende der Nutzungsdauer wieder stark anzusteigen.

Abb. 6.3: Badewannenkurve



Definition (Risiko)

Definition (Risiko):
Ein Risiko ist das Produkt der zu erwartenden Eintrittshäufigkeit (Wahrscheinlichkeit) eines zum Schaden führenden Ereignisses und des bei Eintritt des Ereignisses zu erwartenden Schadensausmaßes.

Definition (Grenzrisiko):
Mit Grenzrisiko bezeichnen wir das größte noch vertretbare Risiko.

Definition (Grenzrisiko)

Das Grenzrisiko ist auf rein quantitativer Basis ein schwer erfassbarer Wert. Die Anforderungsdefinition eines sicherheitskritischen eingebetteten Systems enthält im Regelfall neben Sicherheitsanforderungen sowie funktionalen und nicht-funktionalen Anforderungen auch Anforderungen bezüglich der Systemzuverlässigkeit.

So schreibt beispielsweise die oberste Luftfahrtbehörde der Vereinigten Staaten, die FAA (Federal Aviation Administration) für die Zertifizierung eines neuen Flugzeugtyps vor, dass die Wahrscheinlichkeit für eine sogenannte „catastrophic failure condition" kleiner als 10^{-9} pro Flugstunde ist, d. h. in einer Milliarde Stunden darf höchstens ein solcher Ausfall bezogen auf sämtliche Flugzeuge dieses Typs auftreten (Fränzle, 2002).

Analyse, Synthese

Derartige Zuverlässigkeitsanforderungen bzw. Ausfallraten können nicht empirisch durch Testen nachgewiesen werden. Folglich benötigt man Entwurfs- und Analysemethoden, die sicherstellen, dass ein eingebettetes System die Anforderungen an seine Zuverlässigkeit erfüllt. Genauer gesagt braucht es zum einen Methoden, um die Zuverlässigkeit konstruktiv in das System mit einzubauen *(Synthese),* sowie zum anderen Analyse-Methoden, welche die Zuverlässigkeit eines Systems aus der Zuverlässigkeit seiner einzelnen Komponenten schließen können *(Analyse).* Die Anforderungen an die Zuverlässigkeit, d. h. die zu erzielende Ausfallrate, veranlasst den Designer, bestimmte konstruktive Maßnahmen zu ergreifen und eine Systemarchitektur zu wählen, die diese Anforderungen gewährleistet. Während des weiteren Entwurfs und der Verfeinerung des Systems werden Analyseverfahren eingesetzt, die dann bestätigen, dass der vorgeschlagene Systementwurf den Zuverlässigkeitsanforderungen tatsächlich genügt.

Reliability Engineering

Beide Techniken sind Teil des sogenannten *Reliability Engineering,* das darauf abzielt, die Zuverlässigkeit eines Systems zu erhöhen, die Wahrscheinlichkeit eines Ausfalls zu minimieren und damit insgesamt die Sicherheit des Systems zu verbessern.

6.3.1 Konstruktive Maßnahmen

Fehlertoleranz, Redundanz

Die Zuverlässigkeit eines Systems wird durch Fehler (engl. faults) eingeschränkt. Es stellt sich nun die Frage, welche Maßnahmen man konstruktiv ergreifen kann, um mit diesen tatsächlich auftretenden Fehlern fertig zu werden. Dies geschieht durch die fehlertolerante Auslegung des Systems. *Fehlertoleranz* basiert immer auf einer Form von *Redundanz,* d. h. man macht das System komplexer als es bei Abwesenheit von Fehlern erforderlich wäre.

Unterschiedliche Formen von Redundanz sind die Verwendung zusätzlicher Hardware oder Software mit dem Ziel, Fehler zu erkennen oder zu tolerieren oder der Gebrauch zusätzlicher Information (Beispiele: Paritätsbits, Prüfsummen, fehlererkennende und/oder -korrigierende Codes).

Früher basierten die meisten Ansätze zur Fehlertoleranz auf dem Einsatz redundanter Hardware. Heutzutage werden dagegen oftmals vielfältige Mischformen eingesetzt, um optimale Fehlertoleranz zu erreichen (Fränzle, 2002).

6.3.1.1 Einsatz redundanter Hardware

Ansätze zur Fehlertoleranz

Fehlertoleranz durch redundante Hardware kann durch folgende unterschiedliche Ansätze erreicht werden:

- **Statische Redundanz**, die auf „voting“ (abstimmen) basiert: Hier werden identische Hardwarekomponenten parallel geschaltet und deren Berechnungsergebnisse von einer Voting-Komponente verglichen und anschließend ein Mehrheitsentscheid durchgeführt. Das Voting kann in einer oder mehreren Stufen hintereinander durchgeführt worden.
- **Dynamische Redundanz**, die auf Fehlererkennung und anschließender Rekonfiguration des Systems basiert. Wird ein Fehler erkannt, so schaltet das System auf eine Reservekomponente, beispielsweise den Standby Datenbankserver.
- Schließlich gibt es noch eine **hybride Methode**, welche die beiden vorgenannten Verfahren kombiniert. Dies funktioniert im Wesentlichen wie folgt: Mehrere identische und aktive Systemkomponenten sind über einen Umschalter mit einem Voter verbunden. Nur wenn eine Komponente ausfällt, maskiert der Voter diesen Fehler mithilfe eines Mehrheitsentscheids. Die feh-

lerhafte Komponente wird erkannt und durch Umschalten auf eine der Reservekomponenten ersetzt.

6.3.1.2 Einsatz redundanter Software

Ariane 5

Anders als bei der Hardware-Redundanz ist der mehrmalige Einsatz identischer (baugleicher) Software nicht zielführend. Ein Negativ-Beispiel hierfür hat der Jungfernflug der europäischen Trägerrakete Ariane 5 am 4. Juni 1996 geliefert, dessen Ursachen an dieser Stelle weiter analysiert werden sollen. Wie bereits zu Beginn dieses Kapitels erläutert, musste 37 Sekunden nach ihrem Start die Rakete ferngelenkt zerstört werden, weil sie die vorgeschriebene Flugbahn verlassen hatte und zudem damit zu rechnen war, dass sie in bewohntes Gebiet stürzen könnte. Die Ursache für die Fehlberechnung des Kurses lag daran, dass Software, die vom Vorgängermodell Ariane 4 ohne weitere Überprüfung übernommen wurde, einen Überlauf produzierte – kurioserweise ausgerechnet in einem Softwareteil, der für die Steuerung der Ariane 5 gar nicht nötig gewesen wäre. Obwohl diese Software doppelt vorhanden war, wurde natürlich zweimal der selbe Überlauf produziert.

Das einfache Replizieren von Software (wie bei Hardware) bringt also keinerlei Vorteile. Redundante Software muss demnach mehrfach parallel entwickelt werden. Da dies zu sehr hohen Kosten führt, ist ihr Einsatz hoch sicherheitskritischen Systemen vorbehalten. Folgende Ansätze können unterschieden werden:

Ansätze zur Redundanz

- **Statische Redundanz** (N-Versions Programming): Mehrere Entwicklerteams erstellen verschiedene Implementierungen eines Programms, die auf einem bzw. mehreren Mikroprozessoren nebenläufig (Zeitscheibenverfahren bzw. echte Parallelität) ablaufen. Es folgt ein Vergleich der Ergebnisse sowie ein Mehrheitsentscheid durch einen Voter. Aufgrund der hohen Implementierungskosten sowie ggf. der Verlangsamung der Ausführungszeit ist dieser Ansatz nur bei hochkritischen Systemen, wie z. B. dem Primary Flight Control System des Airbus A330/340 geeignet (Fränzle, 2002).
- **Dynamische Redundanz** (Recovery Blocks): Hier wird eine permanente Fehlererkennung verwendet, um die Funktionsfähigkeit einer Softwarekomponente während des Betriebs zu überprüfen. Wird ein Fehler erkannt, wird auf die Reservekomponente (alternative zweite Implementierung) umgeschaltet.

6.3.2 Analytische Verfahren

Bisher haben wir diskutiert, wie man durch Ausnutzung von Redundanz zuverlässige Gesamtsysteme aus möglicherweise weniger zuverlässigen Systemkomponenten *konstruieren* kann. Im Folgenden wollen wir kurz darauf eingehen, wie man die Zuverlässigkeit von Systemen *analysieren* kann. Unser Ziel wird dabei sein, die Zuverlässigkeit des Systems aus der Zuverlässigkeit seiner Komponenten zu folgern. Dies setzt natürlich voraus, dass wir die Zuverlässigkeit dieser Komponenten kennen.

Zuverlässigkeit

Wie wir bereits gesehen haben, kann die Zuverlässigkeit einer Komponente als Wahrscheinlichkeitswert zwischen 0 und 1 quantifiziert werden. Diese variiert aber in der Regel über den Verlauf der Zeit, wie wir bereits bei der „Badewannenkurve“ im Zusammenhang mit der Ausfallrate gesehen haben.

Die Badewannenkurve wird zur Beschreibung von Früh-, Zufalls- und Verschleißausfällen verwendet. Um den Funktionsverlauf möglichst ideal an die praktischen Ausfälle anpassen zu können, wird eine 3-parametrige Weibull-Verteilung verwendet. Der dritte Parameter, der zur Weibull-Verteilung eingeführt wird, ist ein Lageparameter, mit dem es möglich wird, die Gesamtverteilung aus einzelnen verschobenen Verteilungen zusammenzusetzen.

Kompositionale Zuverlässigkeit

Um unser Ziel verfolgen zu können, also die Zuverlässigkeit eines Gesamtsystems aus der Zuverlässigkeit seiner Komponenten zu berechnen, nehmen wir im Folgenden an, die Zuverlässigkeitswerte für alle Komponenten seien uns bekannt. Wir gehen bei der Berechnung der Gesamtzuverlässigkeit schrittweise vor, indem wir *induktiv* aus einfachen Systemen, angefangen bei Einzelkomponenten, komplexere bauen.

Dieses Zusammenbauen von Komponenten ist vergleichsweise einfach, da es nur die parallele sowie die serielle Kopplung von Komponenten gibt (hierbei bezeichnet $R_i(t)$ mit $0<R_i(t)<1$ die Zuverlässigkeit (engl. reliability) der i-ten Komponente zum Zeitpunkt t und R(t) mit $0<R(t)<1$ die Zuverlässigkeit des Gesamtsystems zum Zeitpunkt t):

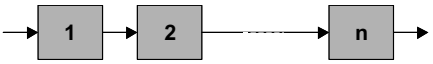
- **Serielle Kopplung** (vgl. Abbildung 6.4(a)): Das Gesamtsystem ist nur dann intakt, wenn *alle* in Serie geschalteten Komponenten intakt sind. Der Ausfall bereits einer beliebigen einzelnen Komponente bewirkt den Ausfall des Gesamtsystems. Die Zuverlässigkeit des Gesamtssystems errechnet sich als Produkt der Zuverlässigkeiten aller Systemkomponenten:

$$R(t) = R_1(t) * ... * R_n(t)$$

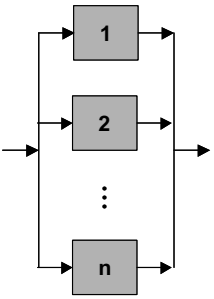
- **Parallele Kopplung** (vgl. Abbildung 6.4(b)): Das Gesamtsystem ist intakt, wenn mindestens eine ihrer parallel gekoppelten Komponenten intakt ist. Die Zuverlässigkeit des Gesamtsystems ist hier etwas komplizierter zu bestimmen. Sie ergibt sich als Produkt der Zuverlässigkeiten aller Systemkomponenten:

$$R(t) = 1-[(1-R_1(t)) * ...* (1-R_n(t))]$$

Das Gesamtsystem ist also nur dann defekt, wenn alle Komponenten defekt sind; die Wahrscheinlichkeit für einen Defekt ist eins minus den Wert für die Zuverlässigkeit.



(a) Serielle Kopplung



(b) Parallele Kopplung

Abb. 6.4: (a) serielle und (b) parallele Kopplung von Komponenten

Bei den Berechnungen haben wir dabei die *Annahme* getroffen, alle in Serie bzw. parallel geschalteten Komponenten seinen unabhängig. Dies erlaubt uns die Betrachtung ihrer Zuverlässigkeiten als *stochastisch unabhängige* Wahrscheinlichkeiten als Grundlage obiger Multiplikationen.

Die Berechnung der *Zuverlässigkeit eines komplexeren Systems* kann immer auf eine Kombination von serieller und paralleler Kopplung zurückgeführt werden. Dazu werden induktiv parallele und serielle Teilsysteme zu jeweils einem „neuen“ Block zusammengefasst.

6.3.3 Stochastische Abhängigkeit

Die vorgenannten Zuverlässigkeitsmodelle gehen davon aus, dass die Ausfallereignisse der einzelnen Systemkomponenten sowohl stochastisch unabhängig als auch unabhängig von der Situation sind bzw. sich zumindest hinreichend genau durch ein solches Modell abstrahieren lassen. Dies ist in der Praxis allerdings häufig nicht der Fall. Zum Beispiel ist die Ausfallwahrscheinlichkeit eines Cold Standby Datenbankservers (ein Ersatzsystem für den eigentlichen Datenbankserver) in der Standby-Phase extrem gering, im Aktivierungsfall, welcher ja gerade bei Ausfall des Primärsystems eintritt, nimmt sie aber signifikant zu.

Markov-Ketten

Um derartige Systeme adäquat modellieren zu können, benötigt man zustandsbasierte Modelle, die eine situationsabhängige Variation der Ereigniswahrscheinlichkeiten darstellen können. Eines der wohl bekanntesten derartigen stochastischen Modelle ist das der *Markov-Kette* bzw. des diskreten *Markov-Prozesses* (Krengel, 1991).

6.3.4 Gefahrenanalyse

Die Zuverlässigkeit eines Gesamtsystems ergibt sich nicht immer unmittelbar aus der seiner Komponentenstruktur. Für die Analyse der Zuverlässigkeit werden deshalb in der Regel auch systematische Suchverfahren angewandt, die den Zusammenhang zwischen Komponentenfehlern und Fehlfunktionen des Gesamtsystems aufzudecken vermögen. Diese *analytischen Suchverfahren* werden im Allgemeinen unter dem Terminus *Gefahrenanalyse* zusammengefasst.

Definition (Gefahr)

Definition (Gefahr):
Als Gefahr (engl. hazard) bezeichnet man dabei eine Sachlage oder Situation bzw. einen Systemzustand, in der bzw. in dem eine Schädigung der Umgebung (Umwelt, Mensch, Maschine) möglich ist.

Eine Gefahrensituation ist also eine Situation, in der das Risiko größer als das Grenzrisiko ist. Gefahren lassen sich im Prinzip auf die ihnen ursächlich zu Grunde liegenden Fehler (engl. faults), die ihrer Art nach zufällig, beispielsweise die Alterung einer

Komponente, oder systematisch, also in das System von vorne herein hineinkonstruiert, sein können, zurückverfolgen.

Definition (Unfall)

Definition (Unfall):
Tritt eine Schädigung dann tatsächlich ein, so bezeichnet man dieses Ereignis als Unfall (engl. accident).

Rückwärts-, Vorwärtsanalyse

Systematisierte Suchverfahren zur Gefahrenanalyse könnten prinzipiell an einer beliebigen Stelle im Ursache-Wirkungsgeflecht, das Fehler mit Gefahren verbindet, ansetzen. In der Praxis haben sich allerdings solche Verfahren durchgesetzt (Fränzle, 2002), die an einem der beiden Endpunkte ansetzen, die also entweder von den möglichen Gefahren ausgehend rückwärts *(Rückwärtsanalyse)* nach der zur jeweiligen Gefahr führenden Fehlerkombinationen suchen oder von möglichen Fehlern oder Fehlerkombinationen ausgehend vorwärts *(Vorwärtsanalyse)* die möglichen Gefahren bestimmen.

Ereignisbaum-analyse

Ein Beispiel der Vorwärtsanalyse ist die *Ereignisbaumanalyse,* ein Beispiel für die Rückwärtsanalyse (deduktiver Ansatz) die Fehlerbaumanalyse (engl. Fault Tree Analysis, kurz: FTA). Sie liefert eine graphische Übersicht möglicher Ursachen und auslösender Ereignisse für einen bestimmten Fehler im System. Die Systemmodellierung geschieht durch den Fehlerbaum, dessen Knoten Symbole für Ereignisse und deren logische Verknüpfungen sind.

FMEA

Ein weiterer, induktiver und systematischer Ansatz ist die *Failure Mode and Effect Analysis (FMEA)* zur Aufdeckung einzelner Fehler auf Komponentenebene und zur Ausarbeitung von Gegenmaßnahmen. Für jede Systemkomponente werden hierbei folgende Fragestellungen untersucht:

- Welche Fehler(-ursachen) können auftreten?
- Welche Folgen haben diese Fehler?
- Wie können diese Fehler vermieden bzw. das Risiko minimiert werden?

Ziele der FMEA

Die Fehlerliste führt dann zu einer Systemüberarbeitung, die wiederum eine neue Analyse notwendig macht; insgesamt ergibt sich ein iterativer Analyseprozess. Die FMEA hat folgende Ziele:

- Kein Fehler darf einen negativen Einfluss (auf redundante Systemteile) haben.
- Kein Fehler darf die Abschaltung der Stromversorgung eines defekten Systemteils verhindern.

- Kein Fehler darf in kritischen Echtzeitfunktionen auftreten.

Ausblick (Biologie):
Beim Bau fehlertoleranter eingebetteter Systeme kann man sicherlich von der Natur, beispielsweise dem menschlichen Körper einiges lernen. Man denke beispielsweise an das Herz, bei dem bei Ausfall des Sinusknotens (70 bis 80 Hz) der Atrioventrikularknoten (40 bis 60 Hz) dessen Funktion mit verminderter Leistungsfähigkeit übernehmen kann. Die Komponente Herz ist weiterhin verfügbar.

Ein anderes Beispiel aus der Biologie ist die DNA-Reparatur mit Fehlertoleranz: Marburger Max-Planck-Forscher haben entdeckt, wie Zellen nicht nur die Effizienz, sondern auch die Genauigkeit von DNA-Reparaturen steuern können (Stelter und Ulrich, 2003). Für die Verdopplung des menschlichen Erbgutes sind spezielle Enzyme verantwortlich, die sogenannten DNA-Polymerasen. Ihre Kopiergenauigkeit trägt entscheidend zur exakten Weitergabe der genetischen Information einer Zelle bei. Beschädigungen der DNA blockieren allerdings diese Enzyme und würden die Zellteilung verhindern, wenn die Zelle nicht über eine Reihe anderer DNA-Polymerasen verfügen würde, die auf die Überwindung derartiger Blockaden spezialisiert sind. Diese gewissermaßen „Notfall-Spezialisten“ sind jedoch wegen ihrer Toleranz gegenüber beschädigter DNA weniger genau und können dadurch unerwünschte, im schlimmsten Falle krebsauslösende Mutationen verursachen. Jüngst wurde ein Signalweg entdeckt, der die Aktivität der Notfall-Polymerasen reguliert und so die Genauigkeit der Erbgutverdopplung mitbestimmt. Die Marburger Max-Planck-Wissenschaftler hoffen so, mit ihren Forschungsergebnissen einen Weg gefunden zu haben, der in Zellen die Entstehung unerwünschter Mutationen verhindert, ohne dass dadurch wichtige fehlerfreie Reparaturvorgänge beeinträchtigt werden. Das könnte ein wichtiger neuer Schritt zur Bekämpfung der Krebsentstehung sein.

6.4 Sicherheit eingebetteter Systeme

Wie ist nun der Zusammenhang zwischen *Zuverlässigkeit* (engl. reliability) und *Sicherheit* (engl. safety)? Ein Systemausfall ist oftmals die Ursache eines Unfalls und damit einer Unzuverlässigkeit. Der Begriff der Sicherheit ist jedoch kein Synonym für Zuverlässigkeit, sondern weiter gefasst als die Zuverlässigkeit. Die internationale Norm *ISO 8402* (ISO = International Standardization

Organization) definiert „safety“ wie folgt: „State in which the risk of harm (to persons) or damage is limited to an acceptable level“.

Verifikation, Korrektheit

Die Sicherheit eines Systems ist eine Eigenschaft, die eng daran gekoppelt ist, ob die Funktionsweise des implementierten Systems mit dessen *Spezifikation* tatsächlich übereinstimmt. Man spricht dann auch von der *Korrektheit* des Systems bezüglich einer Eigenschaft. Mathematische Verfahren und Techniken, die dies sicherstellen, fasst man unter dem Begriff der *Verifikation* (semiautomatisches Theorembeweisen, vollautomatisches Model Checking) zusammen.

Ein sicheres System ist nicht zwangsläufig auch ein zuverlässiges. Viele Unfälle treten auf, obwohl alle Systemkomponenten gemäß ihrer Spezifikation arbeiten. Umgekehrt muss der Ausfall einer Komponente nicht unbedingt die Sicherheit beeinträchtigen oder gar zu einem Unfall führen. Fällt beispielsweise die Zentralverriegelung eines Fahrzeugs aus, so ist dies im Normalbetrieb völlig unbedenklich. Anders ist dies bei einem Unfall, bei dem die Zentralverriegelung automatisch alle Türen entriegeln sollte, damit eine reibungslose Versorgung der Insassen möglich ist.

Als Folgerung kann man schließen, dass die Zuverlässigkeit in der Regel eine notwendige, jedoch keine hinreichende Bedingung für die Sicherheit ist. Eine mangelhafte Zuverlässigkeit ist nur für jene Gefahren und Unfälle verantwortlich, die durch Systemversagen verursacht werden, nicht aber für solche Gefahren, die von dem System bei korrekter Funktionsweise ausgehen.

Validierung

Von der Verifikation grenzt sich der Begriff der *Validierung* ab (siehe auch Abbildung 6.1). Letztere führt den Nachweis, ob das (spezifizierte) System mit den Anforderungen des Auftraggebers bzw. Benutzers einhergeht, wohingegen die Verifikation im Entwicklungsprozess später ansetzt und überprüft, ob das implementierte System mit der Systemspezifikation übereinstimmt und ob die Systemspezifikation bestimmte Eigenschaften erfüllt. Validierung ist also die Antwort auf die Frage, ob das *richtige System* (also das von Auftraggeber gewollte) gebaut wird. Dagegen gibt die Verifikation eine Antwort auf die Frage, ob das *System richtig* (also korrekt) konstruiert wird.

Vorsicht ist darüber hinaus insofern geboten, da der Begriff der Sicherheit, wie wir ihn hier verwenden mit Datenschutz, Kryptographie usw. nichts zu tun hat. Im Englischen spräche man in diesem Fall von „*security*“.

6.4.1 Testen

Für die Entwicklung von (eingebetteter) Software ist, wie wir noch in Kapitel 7 genauer sehen werden, eine systematische Vorgehensweise erforderlich. Dabei folgt man Methoden, die grundlegende Arbeitsschritte wie z. B. Analyse, Entwurf, Konstruktion, Testen und Abnahme beinhalten. Da Fehler in eingebetteten Systemen generell sehr kostspielig sind, wachsen auch die Anforderungen an die Softwarequalität. Um nun die Qualität der Software zu steigern und etwaige Folgekosten zu vermeiden, muss sie ausgiebig getestet werden. Da der Testaufwand im Gegensatz zur Codierung meistens höher liegt und somit die Entwicklungskosten nach oben treibt, sind Testverfahren, -sprachen, -methoden und -werkzeuge zu verwenden, die dem entgegenwirken können.

6.4.1.1 Überblick

Wie wir bereits festgestellt haben, löst das Testen von Software Laufzeitfehler (failures) aus und führt zur Entdeckung von Programmfehlern (faults). Dieser Prozess der Fehlerlokalisierung muss der Fehlerkorrektur stets vorausgehen. Bekannt ist zunächst lediglich die Fehlerwirkung (failure), nicht aber die genaue Stelle (fault) in der Software, die zu dem Fehler führt.

Debugging

Das Lokalisieren und Beheben von Fehlern wird oft auch als *Debugging* bezeichnet. Debugging und Testen werden – fälschlicherweise – oft gleichgesetzt; es handelt sich hier allerdings um zwei völlig unterschiedliche und getrennt zu betrachtende Aufgaben (Spillner und Linz, 2003). Während das Debugging das Ziel hat, Defekte bzw. Fehlerzustände zu beheben, ist es die Aufgabe des Testens, Fehlerwirkungen *gezielt und systematisch* aufzudecken.

Testen, Test, Testobjekt

Unter dem *Testen* von Software wird jede (im Allgemeinen stichprobenartige) Ausführung eines *Testobjekts*, also beispielsweise eines Programms oder dessen Komponenten verstanden, die seiner Überprüfung dient. Die Randbedingungen für die Ausführung des Tests müssen ebenfalls festgelegt sein. Ein Vergleich zwischen dem Sollverhalten (definiert durch die Spezifikation des Testobjekts) und dem Ist-Verhalten dient zur Bestimmung, ob das Testobjekt die (in seiner Spezifikation) geforderten Eigenschaften erfüllt. Oft wird der gesamte Prozess, ein Testobjekt auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen, als *Test* bezeichnet.

Zum Testprozess gehören neben der Aktivität des Ausführens des Testobjekts mit *Testdaten* auch die Planung, Durchführung und das Auswerten der Tests. Man spricht hier auch vom *Testmanagement.* Ein *Testlauf* umfasst die Ausführung eines oder mehrerer Testfälle. Zu einem *Testfall* gehören die festgelegten Randbedingungen; meist handelt es sich hier um die Voraussetzungen zur Ausführung, die Eingabewerte und die erwarteten Ausgaben bzw. das erwartete Verhalten des Testobjekts. Ein Testfall sollte so gewählt werden, dass er in der Lage ist, eine bisher nicht bekannte Fehlerwirkung mit relativ hoher Wahrscheinlichkeit aufzudecken. Meistens werden mehrere Testfälle zu sogenannten *Testszenarien* zusammengefasst. Hierbei kann das Ergebnis eines Testfalls als Ausgangssituation für den darauf folgenden Testfall verwendet werden. Alle Testfälle werden dann im Zusammenspiel sukzessive in einem Testlauf zur Ausführung gebracht.

Testdaten, Testlauf, Testfall, Testmanagement, Testszenarien

Der Prüfer hat alle Testfälle zu ermitteln und zu organisieren. Danach kann er daraus seine Daten generieren und die Sollergebnisse bestimmen. Ein extra für den Test erstellter Ablaufplan hilft bei der Durchführung, um evtl. ausgelassene oder mehrmals wiederholte Testfälle zu verhindern. Alle zum Programm gehörenden Komponenten wie z. B. die Dokumentationen, verschiedenste Konfigurationsversionen, Installationsroutinen usw. sollten mitgetestet werden. Die Ergebnisse sind begleitend in einer Testdokumentation festzuhalten, um sie am Ende des Tests auswerten zu können.

TTCN-3 (Testing and Test Control Notation 3) ist die einzige standardisierte Spezifikationssprache für Tests (Quelle: Presseinformation TestingTech, August 2003, vgl. www.testingtech.de). Sie ermöglicht die Spezifikation und Implementierung von Testfällen sowie deren Ausführungsreihenfolge. Ferner unterstützt es verteiltes und funktionales Testen. Typische Anwendungsgebiete von TTCN-3 sind z. B. der Modul Test, der Internet Protokoll Test, der API Test u.v.m. Mit der Variante *TimedTTCN-3* wurde eine flexible Erweiterung von TTCN-3 zum Testen von Realtime Anforderungen realisiert.

TTCN-3

6.4.1.2 Ausgewählte Testverfahren

Im Folgenden wollen wir exemplarisch einige bekannte Testverfahren erläutern:

Black-Box Test, Funktionstest

Funktionstest (Black-Box Test):
Mit dem Funktionstest sollen Umstände entdeckt werden, bei denen sich der Prüfgegenstand nicht gemäß den Anforderungen bzw. Spezifikationen verhält. Die Testfälle werden hierbei aus den Anforderungen bzw. Spezifikationen abgeleitet. Der Prüfgegenstand wird als schwarzer Kasten (engl. black box) angesehen, d. h. der Prüfer ist nicht an der internen Struktur und dem Verhalten des Prüfgegenstandes interessiert. Die folgenden Black-Box Testfallentwurfsmethoden lassen sich unterscheiden:

- **Äquivalenzklassenbildung:** Ziel ist es, durch die Bildung von Äquivalenzklassen eine hohe Fehlerentdeckungswahrscheinlichkeit mit einer minimalen Anzahl von Testfällen zu erreichen. Dabei werden die gesamten Eingabedaten eines Programms in eine endliche Anzahl von Äquivalenzklassen unterteilt, so dass man annehmen kann, dass mit jedem beliebigen Repräsentanten einer Klasse die gleichen Fehler wie mit jedem anderen Repräsentanten dieser Klasse gefunden werden.
- **Grenzwertanalyse:** Hierbei wird versucht, Testfälle zu definieren, mit denen Fehler im Zusammenhang mit der Behandlung der Grenzen von Wertebereichen aufgedeckt werden können. Die Aufgabe der Grenzwertanalyse besteht nun darin, die Grenzen von Wertebereichen bei der Definition von Testfällen zu berücksichtigen. Ausgangspunkt sind die mittels Äquivalenzklassenbildung ermittelten Äquivalenzklassen. Im Unterschied zur Äquivalenzklassenbildung wird kein beliebiger Repräsentant der Klasse als Testfall ausgewählt, sondern Repräsentanten an den Grenzen der Klassen. Die Grenzwertanalyse stellt somit eine Ergänzung des Testfallentwurfs gemäß Äquivalenzklassenbildung dar.
- **Intuitive Testfallermittlung:** Bei der intuitiven Testfallermittlung wird versucht, die systematisch ermittelten Testfälle qualitativ zu verbessern indem ergänzende Testfälle mit eingebracht werden. Diese zusätzlichen Testfälle werden vom Prüfer aufgrund seiner Erfahrung mit sogenannten Standardfehlern erstellt.
- **Funktionsüberdeckung:** Hierbei gilt es Testfälle zu erstellen, mit denen nachgewiesen werden kann, dass die jeweilige Funktion vorhanden ist und auch ausgeführt werden kann. Der Prüfgegenstand soll somit sein Normalverhalten und das Ausnahmeverhalten zeigen.

Vorsicht: Alle funktionsorientierten Tests sind Black-Box Tests, jedoch nicht alle Black-Box Tests sind funktionsorientiert (z. B. der Back-to-Back Test) (Liggesmeyer, 2002). In früherer Literatur werden beide Begriffe fälschlicherweise als synonym betrachtet.

White-Box Test, Strukturtest

Strukturtest (White-Box Test):
Bei diesem Testverfahren wird die interne Struktur des Prüfgegenstandes untersucht, um aufgrund der Programmlogik und unter Berücksichtigung der Spezifikationen ablauforientierte Testfälle zu bestimmen. Beim Erstellen der Testfälle wird der angesprochene Bereich des Prüfgegenstandes betrachtet. Betrachtungsgegenstand können beispielsweise Pfade, Anweisungen, Zweige und Bedingungen sein. Die folgenden White-Box Testfallentwurfsmethoden lassen sich unterscheiden:

- **Pfadüberdeckung:** Bei der Pfadüberdeckung gilt es Testfälle zu erstellen, die eine geforderte Mindestanzahl von Pfaden (Programmabläufen) im Prüfgegenstand zur Ausführung bringen. Die Ausführung aller Pfade ist meistens aus Komplexitätsgründen nicht möglich. So würde der vollständige Pfadüberdekkungstest einer Fallunterscheidung mit n Fällen sowie umgebender Schleife mit k Iterationen $(n+1)^k + ... + (n+1)^2 + (n+1)$ unterschiedliche Pfade untersuchen müssen. Dies würde für n=4 und k=20 sowie einem Zeitaufwand von 5 Minuten pro Testfall (Spezifikation, Ausführung, Auswertung) in etwa zu einem theoretischen Zeitaufwand von 1 Milliarde Jahren führen. Es handelt sich also hier um ein rein theoretisches Kriterium, das aber als Vergleichsmaßstab für andere Testverfahren dient. Allerdings kann selbst dieses komplexe Verfahren noch nicht alle Fehler (z. B. Berechnungsfehler) entdecken, da es sich immer noch um keinen erschöpfenden Test aller möglichen Eingabewerte handelt.
- **Anweisungsüberdeckung (C0-Test):** Es werden Testfälle erstellt, die eine geforderte Mindestanzahl von Anweisungen im Prüfgegenstand zur Ausführung bringen. Hier handelt es sich um ein Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen betrachtet werden, und so bleiben viele Fehler unentdeckt.
- **Zweigüberdeckung (C1-Test):** Erstellt werden Testfälle, die bei jeder Entscheidung bzw. Fallunterscheidung (if-then-else) mindestens einmal den Then-Zweig sowie den Else Zweig durchlaufen. Hier handelt es sich um ein *realistisches* Minimalkriterium. Der Zweigüberdeckungstest umfasst auch den An-

weisungsüberdeckungstest. Fehler bei Schleifen oder anderer Kombination von Zweigen bleiben allerdings unentdeckt.

- **Bedingungsüberdeckung:** Unter Berücksichtigung der Spezifikation werden Bedingungen identifiziert und entsprechende Testfälle definiert. Die Testfälle werden anhand von Pfadablaufanalysen ermittelt. Jede Teilbedingung einer Kontrollflussbedingung (z. B. von If- oder While-Anweisung) muss einmal den Wert „true“ und einmal den Wert „false“ annehmen. Es werden folgende Sonderfälle unterschieden: Die atomare Bedingungsüberdeckung, die Mehrfachbedingungsüberdeckung sowie die minimale Mehrfachbedingungsüberdeckung. Auch die Bedingungsüberdeckung ist eine Obermenge der Anweisungsüberdeckung.

Als besonders effektiv hat sich die Kombination aus Funktionstest und Strukturtest erwiesen, da sich die Stärken und Schwächen beider Verfahren ausgleichen. Da die Ermittlung von Strukturtestfällen in der Regel aufwändiger ist, wird angeraten, den Funktionstest als erstes durchzuführen.

Klassifikationsbaum

Klassifikationsbaum Methode:
Mit der Klassifikationsbaum (engl. classification tree) Methode steht eine systematische und leicht erlernbare graphische Testmethode zur Verfügung, die zu redundanzarmen und fehlersensitiven Testfällen führt. Die grundsätzliche Idee der Klassifikationsbaum Methode ist es, zuerst die Menge der möglichen Eingaben für das Testobjekt getrennt auf verschiedene Weisen, unter jeweils einem geeigneten Gesichtspunkt zu zerlegen, um dann durch Kombination dieser Zerlegungen zu Testfällen zu gelangen.

Target Test

Test von Echtzeitanforderungen (Target Test):
Für diesen Test sind eine Reihe von Anforderungen durch ein Testsystem zu erfüllen. Hierzu zählt die Berücksichtigung von Echtzeitbedingungen. Um diese zu überprüfen, leitet der Prüfer Testfälle zur Feststellung der kürzesten und längsten Ausführungszeit aus dem Programmcode ab.

6.4.2 Manuelle Prüftechniken

Inspektion, Review, Walkthrough

Das manuelle Prüfen von Programmcode (also des Quelltexts eines Programms) und dessen zugehöriger Dokumente wie etwa dem Pflichtenheft sind eine in der Praxis häufig verwendete Prüftechnik. Das manuelle Prüfen existiert in zahlreichen Ausprägungen (Liggesmeyer, 2002). Man unterscheidet in der Regel zwischen

- der (formalen) Inspektion,
- dem (konventionellen) Review und
- dem (strukturierten) Walkthrough.

Die hier genannten manuellen Prüftechniken erfordern eine Prüfung des Prüfobjekts im Rahmen einer Gruppensitzung. Sie unterscheiden sich durch die Formalität des Vorgehens im Rahmen dieser Sitzung(en) und deren Aufwand. Formale Inspektionen sind ein besonders effektives, aber im Allgemeinen auch sehr zeitaufwändiges Mittel zur Lokalisation von Fehlern. Ein Walkthrough wird oft als weniger sorgfältiges Review betrachtet. Reviews in Sitzungstechnik nehmen eine mittlere Stellung ein. Sie verlangen einen geringeren Ressourceneinsatz (vor allem Arbeitszeit) als formale Inspektionstechniken.

Ein großer Vorteil manueller Prüftechniken ist, dass sie in der Lage sind, semantische Aspekte des Prüfobjekts zu beachten. Durch den Einsatz von Experten in den Sitzungen können inhaltliche Aspekte des Prüfobjekts bewertet werden und die Qualität zahlreicher unterschiedlicher Softwarequalitätsmerkmale beurteilt werden, wie etwa die Verständlichkeit, Änderbarkeit, Aussagekraft von Variablen und Kommentaren usw. Der Nachteil der manuellen Prüftechniken ist bereits durch ihre Namensgebung ausgedrückt: Derartige Tätigkeiten können nicht durch CASE-Tools (CASE = Computer Aided Software Engineering), also Softwarewerkzeuge, automatisiert werden.

Empirische Studien haben gezeigt (Möller, 1996), dass Softwarefehler, die in den frühen Phasen der Softwareentwicklung, also z. B. in der Analyse- oder Entwurfsphase, entstehen, vergleichsweise hohe Korrekturkosten verursachen. Der Aufwand für die Fehlerkorrektur wird umso höher, je größer die Distanz zwischen der Fehlerentstehung und der Fehlerentdeckung ist (Boehm 1982). Kostet das Beheben eines Fehlers während der Konzeptentwicklung der Software noch 1 Euro, so sind es in der Entwurfsphase schon 3 Euro, in der Phase der Implementierung

10 Euro und nach Abschluss aller Tests 50 Euro. Wird der Fehler schließlich erst während des Betriebs der Software gefunden, so liegt der Kostenfaktor im Vergleich zur Konzeptphase bei 150:1.

Somit erscheint es sinnvoll, gerade die Dokumente der frühen Entwicklungsphasen einer Prüfung zu unterziehen, wobei syntaktische Prüfungen durch CASE-Tools durchgeführt werden können, semantische Prüfungen jedoch eine manuelle Bewertung durch Menschen erfordern (Liggesmeyer, 2002). Hier nehmen die genannten manuellen Prüftechniken eine entscheidende Rolle ein.

6.4.3 Formale Verifikation

Es gibt mehrere Möglichkeiten wie etwa Simulation, Testen oder formale Verifikation, um die Korrektheit eines eingebetteten, reaktiven Softwareentwurfs zu überprüfen. Simulation und Testen sind die derzeit in der Praxis am weitesten verbreitete Ansätze. Die Simulation kann verwendet werden, um Spezifikationen auszuführen, das System zu analysieren und ggf. sogar dessen Verhalten zu visualisieren. Um jedoch einen möglichst hohen Anteil der Softwarefehler, beispielsweise 95 Prozent, mit Hilfe von Simulation oder Testen zu entdecken, sind sehr umfangreiche Simulations- bzw. Testverfahren erforderlich. Eine hundertprozentige Korrektheit der Spezifikation mit ihrer Hilfe nachzuweisen, ist allerdings nicht möglich. Die vollumfängliche Übereinstimmung des Softwareverhaltens mit der Anforderungsspezifikation kann nur durch formale Verifikation sichergestellt werden.

Verifikation

Verifikation im engeren Sinne bedeutet den formalen Nachweis, dass ein gegebenes (Software- oder Hardware-) System die ihm zugedachten Eigenschaften erfüllt, dass es also seine Spezifikation erfüllt. Neben eher traditionellen, auf der Zusicherung von Vor- und Nachbedingungen für Programmabschnitte basierten Methoden wie beispielsweise dem Hoare-Kalkül von C.A.R. „Tony“ Hoare wird heute vor allem das sogenannte Model Checking verwendet.

Model Checking

Model Checking wird sowohl im Bereich der Hardwareverifikation wie auch zunehmend im Bereich der Softwareverifikation verwendet. Es ist ebenfalls Logik-basiert und eignet sich besonders zur vollautomatischen Verifikation von Eigenschaften (z. B. Sicherheitseigenschaften) nebenläufiger, reaktiver Systeme. Bei der im Bereich des Model Checkings verwendeten Logik handelt es sich um Varianten der normalerweise im Rahmen eines selbst

naturwissenschaftlichen Studiums in der Regel eher selten gelehrten temporalen bzw. modalen Logik. Um Logik-basierte Verifikationssysteme verstehen und benutzen zu können, ist eine gewisse Vertrautheit mit den zu Grunde liegenden formalen Logiken und Kalkülen erforderlich, die wir im Rahmen dieses Buches jedoch nicht schaffen können. Dennoch soll im Folgenden eine kurze, pragmatische und dennoch präzise Einführung gegeben werden.

Eine sehr gute Definition von Model Checking findet sich in (Clarke und Schlinglo, 2001): „Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems". Es handelt sich um ein Entscheidungsverfahren, um zu prüfen, ob eine gegebene Kripke-Struktur ein Modell für eine angegebene temporallogische Formel ist. Für diese Prüfung ist in der Regel eine erschöpfende Durchsuchung *aller* Systemzustände erforderlich. Damit liegt die größte Herausforderung des Model Checkings auf der Hand, nämlich die automatische Verifikation einer in vielen Fällen „explodierenden" Anzahl von Zuständen. Symbolisches Model Checking sowie kompositionale Ansätze schaffen hier bis zu einem gewissen Grad Abhilfe.

Temporale Logik

Haben wir mit Hilfe einer bestimmten formalen Spezifikationstechnik, wie etwa Statecharts oder Esterel, das Verhalten eines reaktiven Systems definiert, möchten wir in der Regel einen Nachweis vitaler, also sicherheitskritischer Systemeigenschaften erbringen. Um solche Eigenschaften formal mittels Model Checking nachweisen zu können, müssen sie zunächst unter Zuhilfenahme einer geeigneten formalen Sprache spezifiziert werden. Klassische Logik ist ungeeignet, um die Dynamik veränderlicher (ggf. nicht-deterministischer) Systeme zu beschreiben.

Safety, Liveness

Hier müssen Aussagen wie etwa (A1) „irgendwann (in der Zukunft) wird an der Kreuzung die Ampel 1 grün und die Ampel 2 rot" oder (A2) „Ampel 1 und Ampel 2 sind zu keinem Zeitpunkt beide grün" getroffen werden können. Grundsätzlich wird bei temporallogischen Aussagen zwischen sogenannten *Safety-* sowie *Liveness-Eigenschaften* unterschieden. Eine Safety-Eigenschaft (engl. safety property) dient dazu, zu jedem Zeitpunkt und für jede (ggf. nicht-deterministische) Verhaltensalternative die Absenz von *unerwünschten* Zuständen zu vermeiden, vgl. Aussage (A2). Im Englischen wird dies oft prägnant durch „nothing bad will happen" zusammengefasst.

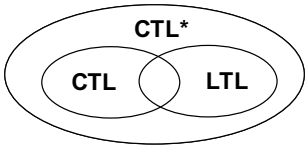
Eine Liveness-Eigenschaft dagegen stellt sicher, dass irgendwann in einem (ggf. nicht-deterministischen) Ausführungspfad des Systems ein *gewünschter* Zustand eintritt, siehe (A1). Dies fasst man im Englischen oft kurz mit „eventually, something good will happen" zusammen.

Temporale Logiken stellen hierfür adäquate Formalismen zu solchen Spezifikation dar. Eine temporale Logik stellt unterschiedliche Operatoren zur Verfügung, um Zeit auszudrücken. Es gibt unterschiedliche Arten der temporalen Logik, die sich sowohl in Semantik als auch in Ausdrucksmächtigkeit unterschieden; (Emerson, 1990) bietet diesbezüglich einen guten Überblick.

*CTL, LTL, CTL**

Eine weiteres Unterscheidungsmerkmal der temporalen Logiken ist die Modellierung der Zeit: Ein diskretes Zeitmodell ist ebenso möglich wie notwendig (für diskrete Systeme) wie ein kontinuierliches (für hybride Systeme). Die am weitesten verbreiteten diskreten temporalen Logiken sind die sogenannte verzweigende temporale Logik (engl. branching time logic) CTL, siehe (Clarke u. Emerson, 1981), die lineare temporale Logik (engl. linear time logic) LTL (Pnueli, 1977) sowie ihre gemeinsame Obermenge CTL* (Emerson u. Halpen, 1986); vgl. Abbildung 6.5. Diese Logiken besitzen eine unterschiedliche Ausdrucksmächtigkeit.

*Abb. 6.5: Klassifikation CTL, LTL, CTL**



Modell

Um mit Hilfe von Model Checking nachweisen zu können, dass eine Systemeigenschaft E von einer Spezifikation S erfüllt wird, benötigen wir ein *Modell* M(S) von S. Das Modell M(S) wird in der Regel durch eine bestimmte Art von endlichen Zustandsübergangssystemen definiert, nämlich durch sogenannte *Kripke-Strukturen* (Emerson, 1990). Sie sind wie folgt definiert:

Definition (Kripke-Struktur)

Definition (Kripke-Struktur):
Eine Kripke-Struktur ist ein Viertupel (S_0, S, R, L), wobei

- S eine endliche Menge von Zuständen (engl. states) darstellt, in der jeder Zustand eine Belegung innerhalb des zu modellierenden Systems repräsentiert;
- S_0 die Menge der Anfangszustände repräsentiert;
- $R \subseteq S \times S$ die Zustandsübergangsrelation darstellt und
- $L: S \rightarrow \wp(A)$ die Markierungsfunktion (von engl. labeling) repräsentiert, wobei A die Menge der Atome des Modells ist.

Model Checking bedeutet dann das Erbringen eines formalen Beweises, dass E in M(S) gilt, oder kurz $M(S) \models E$. Ein besonderer Vorteil des Model Checkings liegt darin, diesen Beweis vollautomatisch unter Zuhilfenahme sogenannter BDDs (Binary Decision Diagrams), also binärer Entscheidungsdiagramme (Bryant, 1986), durchführen zu können. In diesem Falle spricht man vom *symbolischen Model Checking* (McMillan, 1993).

Ein Model Checking-Werkzeug kennt somit zwei mögliche Antworten auf die Frage „$M(S) \models E$“: Ja, die Eigenschaft E gilt in M(S) oder nein, sie gilt nicht. Im letzteren Fall wird zusätzlich ein passendes Gegenbeispiel angegeben.

Theorem-beweisen

Da Model Checking nur für die Prüfung von Prüfobjekten mit einem endlichen Zustandsraum verwendet werden kann, hat sich in letzter Zeit im Bereich der formalen Verifikation sicherheitskritischer Systeme darüber hinaus auch das *interaktive Theorembeweisen* durchgesetzt. Hierbei werden die Verifikations-Aufgaben in einer möglichst ausdrucksstarken Logik formuliert (z. B. in Higher Order Logic, kurz: HOL). Ein Experte führt dann einen mathematischen Beweis der Korrektheit unter Zuhilfenahme eines interaktiven Beweissystems (z. B. „Isabelle“, siehe im Internet isabelle.in.tum.de) durch, das die vorgegebenen Schritte des Experten auf Korrektheit überprüft. „Leichte“ Teilbeweise werden dabei mit Hilfe vorgegebener Heuristiken (sogenannten Taktiken) zu lösen versucht.

6.5 Zusammenfassung

Eingebettete Systeme dringen in alle Lebensbereiche ein und übernehmen dort für den Menschen teilweise oder vollständig auch sicherheitskritische Aufgaben. Da eingebettete Systeme meist komplexe Steuerungsaufgaben in einer technischen Umgebung übernehmen, deren Defekt oder inkorrekte Funktionsweise zu ernsthaften Auswirkungen auf menschliches Leben haben kann, ist die Sicherung einer hohen Qualität dieser Systeme Pflicht.

Eine mangelhafte Softwarequalität solcher Systeme kann in einigen Anwendungsbereichen signifikante Gefährdungen hervorrufen. Beispiele hierfür wurden diskutiert. Die wesentlichen Begriffe der Softwarequalität haben wir definiert. Hierbei haben wir vor allem die beiden Begriffe der Zuverlässigkeit sowie der Sicherheit diskutiert und festgestellt, dass sich beide nicht ausschließen, aber auch nicht synonym betrachtet werden können. Obwohl die

Sicherheit hier der umfassendere Begriff ist, gibt es zuverlässige Systeme, die nicht sicher sind und umgekehrt.

Um eine möglichst hohe Softwarequalität garantieren zu können sind verschiedenste sowohl konstruktive als auch analytische Verfahren bekannt, wobei wir hier im Wesentlichen drei unterschiedliche Ansätze aus dem Bereich der letztgenannten skizziert haben.

ISEB, ASQF

Das lebenslange Lernen ist aber insbesondere im Bereich der Informationstechnologie unverzichtbar. Um im Weiterbildungsbereich für dieses Thema für bessere Vergleichbarkeit der Kursanbieter untereinander und für einen anerkannten Nachweis zu sorgen, ist in England das Information Systems Examinations Board (ISEB, vgl. www.iseb.org.uk) ins Leben gerufen worden, das unter anderem auch im Bereich des Softwaretestens tätig ist. Die Aktivitäten der ISEB wurden auch von anderen Ländern aufgegriffen und vergleichbare Initiationen gestartet. In Deutschland zeichnet vor allem die ASQF (vgl. www.asqf.de) und die Fachgruppe TAV (Test, Analyse und Verifikation von Software) der Gesellschaft für Informatik e. V. (kurz: GI) für dieses Thema verantwortlich (Spillner und Linz, 2003).

7 Vorgehensmodelle und Standards der Entwicklung

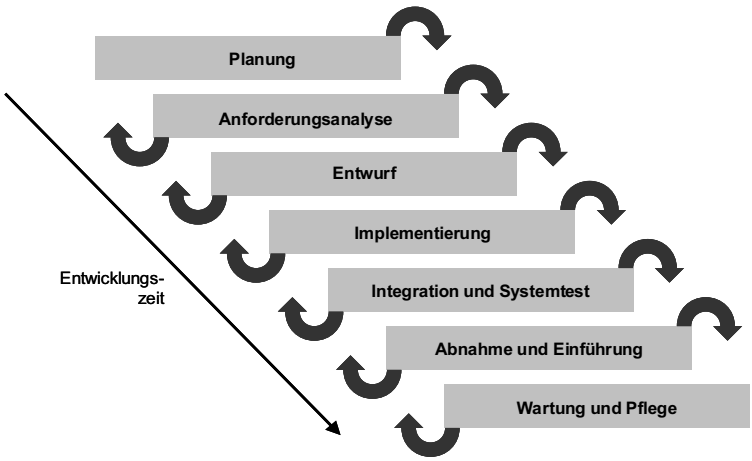
Um den Prozess der Softwareentwicklung strukturiert und steuerbar durchführen zu können, werden spezielle Vorgehensmodelle eingesetzt. Es existieren mehrere ganz unterschiedliche Vorgehensmodelle. Ausgewählte Beispiele hierfür sind das Wasserfall-Modell, das allgemeine V-Modell (beide von Boehm mit entwickelt), das spezielle V-Modell, entwickelt als Vorgehensmodell des Bundes und der Länder, sowie das V-Modell XT als jüngster Vertreter. Bei dem zur UML (Unified Modeling Language) gehörigen Rational Unified Process (kurz: RUP) und Extreme Programming (XP) handelt es sich um aktuelle Ansätze zur Entwicklung von objektorientierten Softwaresystemen. Alle Modelle legen eine Systematik zur geordneten Projektabwicklung fest und definieren damit eine für alle beteiligten Personen gemeinsame und verbindliche Sicht der auszuführenden Tätigkeiten sowie deren zeitliche Abfolge im Projekt (Spillner und Linz, 2003). Die meisten unterteilen die Entwicklungsdauer in verschiedene Phasen, die jeweils mit einem bestimmten Ergebnis in Form eines oder mehrerer Dokumente abzuschließen sind. Der Abschluss einer Phase, meist auch als Meilenstein bezeichnet, ist dann erreicht, wenn alle geforderten Dokumente fertig gestellt wurden und alle erforderlichen Qualitätskriterien erfüllt sind. Manche Vorgehensmodelle schreiben auch die in den jeweiligen Phasen einzusetzenden Methoden und Verfahren vor oder machen zumindest Vorschläge hierfür.

7.1 Das Wasserfall-Modell

Das *Wasserfall-Modell* war das erste grundlegende dieser Vorgehensmodelle. Es ist „bestechend einfach“ (Spillner und Linz,

2003) und aufgrund seines hohen Bekanntheitsgrades weit verbreitet (siehe Abbildung 3.1). Eine Entwicklungsphase kann erst dann begonnen werden, wenn die direkt vorhergehende abgeschlossen wurde. Rückkopplungsschleifen, die bei einer ggf. erforderlichen Überarbeitung entstehen können, gibt es nur zwischen direkt aufeinanderfolgenden Phasen. Ein signifikanter Nachteil des Wasserfall-Modells ist, dass das Testen als einmalige Aktion am Ende des Entwicklungsprojekts vor der Inbetriebnahme betrachtet wird. Dieser Nachteil wirkt sich bei sicherheitskritischen eingebetteten Systemen besonders nachhaltig aus.

Abb. 7.1: Das Wasserfall-Modell (vgl. Balzert, 1998)



7.2 Das V-Modell

Allgemeines V-Modell

Beim allgemeinen *V-Modell von Boehm* handelt es sich um eine Erweiterung des Wasserfall-Modells, bei dem die konstruktiven Aktivitäten von den prüfenden getrennt werden. Die Namensgebung erfolgte aufgrund des sich durch diese Trennung ergebende Form des Buchstaben V, wobei die konstruktiven Aktivitäten bis zur Implementierung auf dem linken, absteigenden Ast des „V"s zu finden sind und die Testaktivitäten auf dem rechten, aufsteigenden Ast (vgl. Abbildung 7.2).

Das *spezielle V-Modell* des Bundes und der Länder ist neben dem militärischen Bereich auch für den Bereich der Bundesverwaltung verbindlich (seit etwa 1992). Es ist konsistent zur Norm DIN EN

ISO 9001 (Liggesmeyer, 2002). Beim V-Modell bilden organisatorische Maßnahmen den Schwerpunkt. Dabei werden folgende Entwicklungsebenen behandelt:

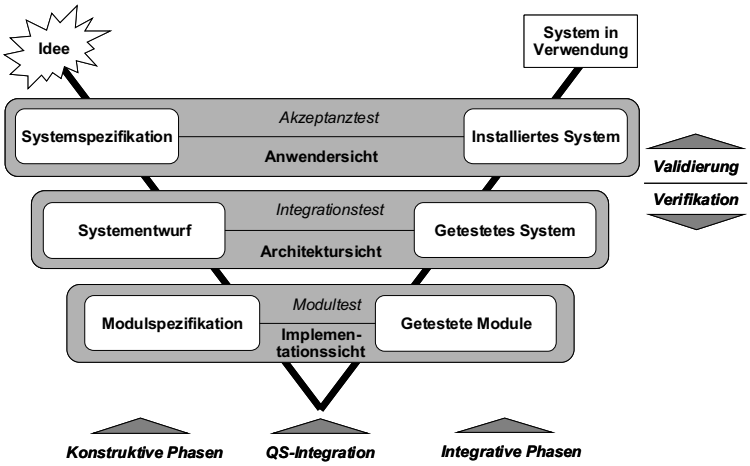


Abb. 7.2: Das V-Modell (stark abstrahiert)

Spezielles V-Modell

- **Vorgehensweise** („*was* ist zu tun?“): Diese Ebene legt neben einem allgemeinen Regelungsteil fest, welche Tätigkeiten im Verlauf der Systementwicklung in den einzelnen Submodellen (SE, QS, KM und PM) durchzuführen sind, welche Ergebnisse dabei zu produzieren sind und welche Inhalte diese Ergebnisse haben müssen (spezifiziert im sogenannten Allgemeiner Umdruck (AU) 250). Im AU 250 wird darüber hinaus eine Einführung in die im V-Modell verwendete Terminologie gegeben.
- **Methoden** („*wie* ist etwas zu tun?“): Auf dieser Ebene erfolgt eine Zuordnung von Methoden zu den auf der ersten Ebene festgelegten Tätigkeiten nebst einer Beschreibung (dies wird im AU 251 geregelt).
- **Werkzeuganforderungen** („*womit* ist etwas zu tun?“): Hier ist festgehalten, welche Eigenschaften eine Softwareentwicklungsumgebung erfüllen muss, um als einsatztauglich für das V-Modell eingestuft werden zu können (siehe AU 252).

Submodelle

Das V-Modell besteht aus den vier Submodellen SE (System Erstellung, ein System kann sowohl aus Hardware- als auch aus Software bestehen), QS (Qualitätssicherung), PM (Projektmanagement) sowie KM (Konfigurationsmanagement). Auf allen Ebenen

werden die Regelungen nach den Tätigkeitsbereichen, den sogenannten Submodellen gegliedert:

- **SE** erstellt das System bzw. die Software.
- **QS** gibt Qualitätsanforderungen, Prüffälle und -kriterien vor und untersucht die Produkte und die Einhaltung der Standards.
- **PM** plant, kontrolliert und informiert die Submodelle SE, QS und KM.
- **KM** verwaltet die erzeugten Produkte.

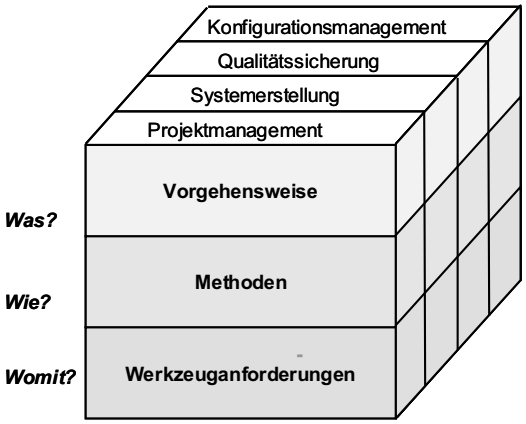


Abb. 7.3: Die Dimensionen bzw. Ebenen des V-Modells

Dieses Modell definiert die Aktivitäten, Ergebnisse (Produkte) und Zustände der Produkte sehr genau. Dabei werden nicht nur der Entwicklungsprozess, sondern auch die Prozesse der Wartung und Pflege beschrieben. Da das spezielle V-Modell ein großes Gewicht auf Maßnahmen der Qualitätssicherung legt, hat es über den öffentlichen Bereich hinaus auch eine breite Verwendung im Bereich der eingebetteten Systeme, insbesondere in der Automobilindustrie, gefunden. Auf unterschiedlichen Abstraktionsebenen werden die logischen Abhängigkeiten zwischen den Aktivitäten, Produkten und Produktzuständen detailliert beschrieben. Das V-Modell eignet sich daher insbesondere dann, wenn verschiedene dieser Abstraktionsebenen von unterschiedlichen Personengruppen, beispielsweise in einem Lieferantenverhältnis entwickelt werden. Auch aus diesem Grund wird es gerne beim Automobilbau zur Entwicklung eingebetteter Systeme verwendet. Weitere Informationen zum speziellen V-Modell findet der Leser im Internet unter www.v-modell.iabg.de sowie unter www.ansstand.de.

Alle dieser Vorgehensmodelle enthalten Ansätze zum Testen, einer Prüftechnik zur analytischen Qualitätssicherung. Allerdings unterscheiden sich die Modelle hinsichtlich der Bedeutung und des Umfangs des Testens teilweise sehr deutlich. Es liegt daher auf der Hand, warum wir uns im vorangegangen Kapitel dem Thema der Softwarequalität eingebetteter Systeme widmeten.

7.3 Das V-Modell XT

Historie

Am 4. November 2004 wurde das V-Modell XT (kurz für eXtreme Tailoring) dem interministeriellen Koordinierungsausschuss (IMKA) vorgestellt. Gleichzeitig ist das KBSt-Portal im Internet für das Bundesministerium des Inneren eröffnet worden (siehe www.kbst.bund.de). Dort wird die Version 0.9 des V-Modell XT der Öffentlichkeit präsentiert. Die endgültige Version 1.0 wird am 4. Februar 2005 veröffentlicht.

Das V-Modell XT ist eine eingetragene Marke der Bundesrepublik Deutschland und ist für die Bereiche Verteidigung, Behörden und Firmen vorgesehen. Seit November 2002 haben sich unter der Leitung des Vereins ANSSTAND e.V. (Anwender des Software-Entwicklungsstandards der öffentlichen Verwaltung) die Überlegungen zur Fortschreibung des V-Modells-97 konkretisiert. Es wurde ein Konsortium unter der Leitung der Technischen Universität München gebildet. In dem Konsortium sind die Firmen EADS, Siemens München und IABG (siehe www.vmodell.iabg.de) vertreten. Die Fortschreibung soll in 3 Phasen erfolgen (Quelle: www.kbst.bund.de):

- In der **Phase 1** sollen die Anforderungen an das neue V-Modell 200X definiert und ein Strategiepapier zur Umsetzung erstellt werden.
- In **Phase 2** sollen die Anforderungen aus Phase 1 umgesetzt werden.
- In **Phase 3** soll das V-Modell 200X eingeführt und verbreitet werden.

Im Juni 2003 wurde die Phase 1 zur Fortschreibung des V-Modell abgeschlossen und abgenommen. Die Phase 2 wurde im Zeitraum Juni 2003 bis August 2004 abgeschlossen. Das neue V-Modell hat einen Namenswechsel durchlaufen. Der Arbeitsbegriff V-Modell 200x wurde durch V-Modell XT (für eXtreme Tailoring) ersetzt und

der Name durch die Bundesrepublik Deutschland geschützt. Das V-Modell ist im Verlauf von sechs Releases zu einem beträchtlichen Umfang gewachsen und steht nun in PDF und HTML zur Verfügung (siehe www.kbst.bund.de). Die Phase 3 hat ab September 2004 begonnen und wird bis ins erste Quartal 2006 andauern. Wesentliche Ziele der Phase sind (Quelle: www.kbst.bund.de):

- Etablierung eines Änderungsprozesses
- Durchführung letzter Änderungen am Modell
- Englischübersetzung
- Erstellung eines Internet-Portals, einer Lerntour und von Schulungsunterlagen
- Überprüfung der Anwendbarkeit durch Pilotprojekte

7.3.1 Grundlagen

Im Rahmen der Weiterentwicklung wurde das V-Modell inhaltlich erweitert. Ferner wurden die Qualitätseigenschaften des V-Modells verbessert, insbesondere hinsichtlich der projekt- und organisationsspezifischen Anpassbarkeit, der Anwendbarkeit im Projekt, der Skalierbarkeit auf unterschiedliche Projektgrößen sowie der Änder- und Erweiterbarkeit des V-Modells selbst. Um dies zu erreichen wurde die Struktur des V-Modells komplett überarbeitet, und das ehemals monolithische V-Modell wurde in einzelne Bausteine aufgeteilt. Vordefinierte Ablaufrahmen beschreiben, welche dieser Bausteine in einer konkreten Projektkonstellation zum Einsatz kommen, und in welcher Reihenfolge die benötigten Produkte und Zwischenergebnisse zu erarbeiten sind (Quelle: www.kbst.bund.de).

Eigenschaften

Ein wesentliches Prinzip des V-Modells ist seine Ziel- und Ergebnis-orientierte Vorgehensweise. Diese Grundphilosophie ist an vielen Stellen im V-Modell sichtbar:

- Produkte stehen im Mittelpunkt des V-Modells. Sie sind die zentralen Projektergebnisse.
- Projektdurchführungsstrategien und Entscheidungspunkte geben die Reihenfolge der Produktfertigstellung und somit die grundlegende Struktur des Projektverlaufs vor.

- Die detaillierte Projektplanung und Steuerung wird auf der Basis der Bearbeitung und Fertigstellung von Produkten durchgeführt.
- Für jedes Produkt ist eindeutig eine Rolle verantwortlich bzw. in einem konkreten Projekt eine dieser Rolle zugeordnete Person oder Organisationseinheit.
- Die Produktqualität ist durch definierte Anforderungen an das Produkt und explizite Beschreibungen der Abhängigkeiten zu anderen Produkten überprüfbar.

Die im V-Modell definierten Produkte sind somit die zentralen Zwischen- und Endergebnisse des Projektes. Ausgehend von den Projektzielen werden diese Ergebnisse bei der Projektkonzeption und Planung definiert und im Zuge einer professionellen Vorgehensweise während des Projektverlaufs bearbeitet und fertig gestellt. Die Ziel- und Ergebnisorientierung des V-Modells vermeidet unnötige, nicht an Ergebnissen ausgerichtete Tätigkeiten. Aktivitäten und Teilaktivitäten, die keinen Beitrag zur Ergebniserstellung liefern, werden im V-Modell nicht beschrieben. Diese Fokussierung des V-Modells stellt eine wesentliche Grundvoraussetzung für eine effiziente Projektabwicklung dar (Quelle: www.kbst.bund.de).

7.3.2 Anwendung des V-Modell XT

Ablaufschema

Ein Projekt läuft bei Verwendung des V-Modell XT nach dem folgenden Schema ab (Quelle: www.kbst.bund.de):

- **Projektvorschlag:** Zuerst wird ein Projektvorschlag erstellt, auf dessen Grundlage sich der Auftraggeber für oder gegen eine Ausschreibung des Projektes entschließen kann. Dabei kann man sich an die V-Modell-Referenz Produkte halten, in der Vorschläge für Inhalt und Gliederung eines Projektvorschlags beschrieben sind.
- **Projektfortschrittsentscheidung:** Nun wird der Projektvorschlag ausführlich besprochen, nötige Änderungen werden vorgenommen und somit das Projekt verbindlich festgelegt. Ebenfalls werden Terminziele, Qualitätsziele, Ressourcenplanung und Budget festgelegt.

- **Projektdefinition:** Bei der Projektdefinition werden die Rahmenbedingungen verfeinert und erweitert. Dabei wird unter Verwendung des Tailorings eine projektspezifische Anpassung des V-Modells an das Projekt durchgeführt und im Projekthandbuch dokumentiert. D. h. der Projektleiter wählt aus mehreren zur Verfügung stehenden Projektmerkmalen jeweils die für das Projekt passenden Werte aus. Eine eingehende Beschreibung des Tailoring-Mechanismus des V-Modells XT findet sich auf den Seiten des BMI (www.kbst.bund.de). Das Tailoring kann mit Hilfe eines XT-Editors oder von Hand durchgeführt werden.

  Ergebnis des Tailorings:

  - Charakterisierung des Projekts → Vorgehenstypen
  - Vorgehensbausteine
  - Projektdurchführungsstrategie

  Als Ergebnis dieses Abschnitts sollen folgende Dokumente entstehen:

  - Projekthandbuch: Im Projekthandbuch werden die Anpassungen und Ausgestaltung des Projekts für Management und Entwicklung festgelegt.
  - Projektplan
  - QS-Handbuch
- **Anforderungen festlegen**
- **Projekt ausschreiben**
- **Projekt beauftragen**
- **Abnahme**
- **Änderungsplan festlegen:** Falls Änderungen nötig sind, ist wieder ab Punkt „Anforderungen festlegen“ fortzufahren.
- **Projektende**

7.3.3 Zielsetzung und Aufbau des V-Modell XT

Unter Berücksichtigung des gesamten Systemlebenszyklus ist das V-Modell als Leitfaden zum Planen und Durchführen von Entwicklungsprojekten zu sehen. Durch die standardisierten, methodischen Vorgaben des V-Modells werden komplexere und umfangreichere Projekte besser plan- und nachvollziehbarer und

erzielen so zuverlässigere Ergebnisse von höherer Qualität. Das neue V-Modell XT regelt die Verantwortlichkeit zwischen Auftraggeber und Auftragnehmer *wer wann was* zu tun hat und bildet somit die Grundlage für die vertragliche Zusammenarbeit. Kleine und mittelständische Unternehmen profitieren ebenso vom neuen V-Modell XT. Sie haben damit die Möglichkeit auf standardisierte und erprobte Vorgaben für Entwicklungs- und Managementprozesse zurückzugreifen, um hochwertige und zuverlässige Entwicklungsergebnisse mit überschaubarem Aufwand zu erzielen (Quelle: www.kbst.bund.de).

7.3.3.1 V-Modell XT als Weiterentwicklung des V-Modells 97

Das V-Modell XT ist die Weiterentwicklung des im Jahre 1997 fertiggestellten V-Modell 97. Das V-Modell 97 spiegelt im Jahre 2004 nicht mehr den aktuellen Stand der Informationstechnologie wider. Neuere Ansätze wie die komponentenbasierte Entwicklung oder der Test-First-Ansatz werden nur bedingt berücksichtigt. Das hat zur Folge, dass das V-Modell 97 nicht mehr in wünschenswertem Maße genutzt wird. Die Erfahrungen, die mit dem V-Modell 97 gesammelt und die Verbesserungsvorschläge die gemacht wurden, werden im V-Modell XT umgesetzt. Das IT-AmtBw A5 und das BMI-KBSt haben die Entwicklungsstandards für IT-Systeme des Bundes auf Basis des V-Modells 97 weiterentwickelt. Folgende Anforderungen wurden umgesetzt (Quelle: www.kbst.bund.de):

Umgesetzte Anforderungen

- Verbesserung folgender Qualitätseigenschaften:
  - Möglichkeit zur Anpassung an verschiedene Projekte und Organisationen
  - Anwendbarkeit im Projekt
  - Skalierbarkeit auf unterschiedliche Projektgrößen
  - Änder- und Erweiterbarkeit des V-Modells selbst
- Berücksichtigung neuer Standards und Anpassung an neue Vorschriften
- Erweiterung des Anwendungsbereiches auf die Betrachtung des gesamten Systemlebenszyklus bereits während der Entwicklung
- Einführung eines organisationsspezifischen Verbesserungsprozesses für Vorgehensmodelle

7.3.3.2 Zielsetzung des V-Modells XT

Ziele

Folgende Ziele werden mit der V-Modell konformen Durchführung eines Projektes verfolgt (Quelle: www.kbst.bund.de):

Minimierung der Projektrisiken:
Durch eine standardisierte Vorgehensweise verbessert man die Projekttransparenz, und alle Ergebnisse und Rollen werden definiert. Planungsabweichungen können dadurch frühzeitig erkannt und das Projektrisiko damit minimiert werden.

Verbesserung der Gewährleistung der Qualität:
Durch das standardisierte Vorgehensmodell wird sichergestellt, dass die zu liefernden Ergebnisse vollständig und von gewünschter Qualität sind. Durch die im Modell definierten Zwischenergebnisse können Ergebnisse frühzeitig überprüft werden. Durch die Vereinheitlichung der Produktinhalte sind die Ergebnisse besser lesbar, verständlicher und leichter zu prüfen.

Eindämmung der Gesamtkosten über den gesamten Projekt- und Systemlebenszyklus:
Durch die Anwendung des standardisierten Vorgehensmodells ist eine transparente Kalkulation, Abschätzung und Steuerung in jeder Phase des V-Modells möglich. Erzeugte Ergebnisse sind einheitlich und leicht nachvollziehbar, dadurch werden Abhängigkeiten zwischen Auftraggeber und Auftragnehmer verringert.

Verbesserung der Kommunikation zwischen den Beteiligten:
Eine standardisierte und einheitliche Beschreibung stellt die Basis für alle Projektbeteiligten dar und reduziert die Reibungsverluste zwischen diesen.

7.3.3.3 Grenzen des V-Modells XT

Grenzen

Folgende Gesichtspunkte werden im V-Modell XT nicht (Quelle: www.kbst.bund.de) berücksichtigt:

- Die Vergabe von Dienstleistungen betrachtet nur die Vergabe von kompletten Gewerken.
- Keine Unterstützung für die Entscheidungsfindung, ob der Auftraggeber des Gesamtsystems entwickeln oder ein Standardprodukt einkaufen soll (engl. make or buy).

- Keine Betrachtung der Vergabe von Unteraufträgen ohne Ausschreibungsverfahren.
- Keine Unterscheidung zwischen Auftraggeber und Auftragnehmer bei Einführung und Pflege eines organisationsspezifischen Vorgehensmodells.
- Die Organisation und Durchführung von Betrieb, Instandhaltung, Instandsetzung und Aussonderung des Systems wird nicht abgedeckt. Die Planung und Konzeption dieser Aufgaben ist dagegen geregelt.

7.3.4 V-Modell XT Produktvorlagen

Das V-Modell XT bietet für die Projekttypen

- Einführung und Pflege eines organisationsspezifischen Vorgehensmodells,
- Systementwicklungsprojekt eines Auftraggebers und
- Systementwicklungsprojekt eines Auftragnehmers

jeweils Vorlagen aller zu erstellender Produkte an. Bei den dafür angebotenen Dokumenten handelt es sich um den vollständigen Umfang der Produktvorlagen für den jeweiligen Projekttyp. Für die Generierung projektspezifischer Produktvorlagen existiert das Tool V-Modell XT Projektassistent.

Jede der drei Projekttypen wird durch eine Reihe von Produktvorlagen spezifiziert. Diese Produktvorlagen geben einen groben Rahmen vor, der zu komplettieren ist. Ein solcher Rahmen besteht im Wesentlichen aus einem Kopf und einer Inhaltsangabe. Die Inhaltsangabe gibt einen Leitfaden vor, um die Produktvorlage genau niederzuschreiben (Quelle: www.kbst.bund.de).

7.3.5 V-Modell XT Werkzeuge

Im Lieferumfang des V-Modells sind zwei Softwarewerkzeuge: Der V-Modell XT Projektassistent und der V-Modell XT Editor.

7.3.5.1 Der V-Modell XT Projektassistent

Der V-Modell XT Projektassistent ist die Referenzimplementierung für den im V-Modell XT spezifizierten Tailoring-Mechanismus. Mit seiner Hilfe können für ein konkretes Projekt die Vorgaben des Standards an die Projektsituation angepasst werden. Der Projektassistent ermöglicht die Auswahl von Projektmerkmalen, die konsistente Kombination von Vorgehensbausteinen und die Festlegung von Projektdurchführungsstrategien. Das auf diese Weise justierte V-Modell XT kann anschließend in Form von HTML-Seiten exportiert werden (Quelle: www.kbst.bund.de).

7.3.5.2 V-Modell XT Editors

4Ever

Das V-Modell XT bietet unterschiedliche Möglichkeiten zur Erweiterung und Anpassung des vorgegebenen Rahmenwerks an branchen- oder unternehmensspezifische Bedürfnisse. So können beispielsweise neue Vorgehensbausteine hinzugefügt oder die Struktur der Projektvorlagen ergänzt werden. Hierfür bietet sich die Verwendung des V-Modell XT Editors an, mit dessen Hilfe das V-Modell XT auch entwickelt wurde. Es handelt sich hierbei um einen formularbasierten, strukturierten Editor, der als Plugin der Entwicklungsplattform Eclipse realisiert wurde. Der Editor sowie das zugrunde liegende Framework 4Ever der 4Soft GmbH wurden im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Projekts NOW als Open-Source Software verfügbar gemacht und wird seither kontinuierlich weiterentwickelt (Quelle: www.kbst.bund.de). Der grundlegende Baustein des V-Modell XT Editors ist das Framework „4Ever“. Es wurde im Auftrag des Bundesministerium des Inneren, Abteilung KBSt von der Technischen Universität München in Zusammenarbeit mit der 4Soft GmbH entwickelt. Das Framework legt seinen Schwerpunkt auf eine Team-basierte, verteilte und kontrollierte Erschaffung von großen, strukturierten Dokumenten.

7.4 Die ROPES-Methode

ROPES (Rapid Object-Oriented Process for Embedded Systems) ist eine objektorientierte Methode zur Softwareentwicklung. Auf Basis von Use-Cases (nach Priorität, Risiko und Allgemeinheit sortiert)

wird ein iterativer „Spiral Lifecycle“ durchlaufen (Douglas, 1999a). Das Ergebnis eines solchen Durchlaufs ist ein ausführbarer Prototyp. Daher ist die automatische Codegenerierung ein zentraler Bestandteil dieses Prozesses. Die Philosophie von ROPES lautet „think horizontally, do vertically“ (z. B. horizontal = Hardware-Schichten eines Systems, vertikal = verschiedene Schritte bzw. Zyklen).

Wie beim Rational Unified Process sind die Kernelemente bei ROPES Phasen, Aktivitäten und Artefakte. Die vier Hauptphasen von ROPES lauten: Analyse, Design, Übersetzung und Test. Ergebnisse dieser Phasen sind dann folglich das Analyse-Objektmodell, das Entwurfs-Objektmodell, Anwendungskomponenten sowie die getestete Anwendung. Eine Phase besteht aus mehreren Aktivitäten. Artefakte sind Ergebnisse von Aktivitäten die Einfluss auf eine Phase haben. Bei diesem Prozess nimmt neben der automatischen Codegenerierung auch das Hardware/Software-Codesign und Testen eine wichtige Rolle ein (siehe auch www.embedded.com).

7.5 Der OSEK-Standard

Historie

Das OSEK-Konsortium entstand 1993 als ein Zusammenschluss von BMW, Bosch, Daimler-Chrysler, Opel, Siemens und VW sowie des Instituts für industrielle Informationstechnologie der Universität Karlsruhe (TH). Parallel dazu entstand in Frankreich die Arbeitsgruppe VDX (Vehicle Distributed Executive), deren Mitglieder Renault und Peugeot waren. Im Jahr 1994 schlossen sich die beiden Arbeitsgruppen zusammen und bildeten das OSEK/VDX Konsortium (www.osek-vdx.org).

Ziel

Als Ziel hat sich dieses Konsortium gesetzt, eine einheitliche und offene Architektur für

- die Kommunikation,
- das Netzwerk-Management und
- das Betriebssystem

in automobilen Systemen zu spezifizieren. Hintergrund dieser Aktivitäten ist natürlich, die Entwicklungszeiten und -kosten für Steuergeräte zu minimieren und deren Portierbarkeit zu erhöhen. OSEK selbst ist also kein Betriebssystem, sondern lediglich ein Standard, der die Schnittstellen und das Verhalten eines OSEK-

kompatiblen Betriebssystems vorgibt. Die OSEK-Spezifikation beschreibt ein statisches Echtzeitbetriebssystem, dessen Objekte (wie beispielsweise Tasks, Ereignisse, Nachrichten oder Ressourcen) zur Compile-Zeit angelegt werden. Diese Betriebssystem-Objekte werden in der speziellen Beschreibungssprache OIL (OSEK Implementation Language) definiert. Zu einem Objekt gehören bestimmte OIL-Attribute wie etwa die Priorität einer Task. Zusätzlich zu den Standard-Attributen können OSEK-Hersteller weitere Attribute definieren, die z.B. ihre OSEK-Implementierung auf bestimmte Microcontroller anpassen.

Als nützlicher Nebeneffekt steigt auch die Qualität der Steuergerätesoftware sowie deren Wartbarkeit (Quelle: OSEK/VDX Steering Committee: OSEK/VDX Operating System Specification 2.2.1, OSEK/VDX Steering Committee, 2004).

Steuergremium

Aufgabe des Steuergremiums, dem immer noch (Stand: November 2004) die vorgenannten Mitglieder angehören, ist es, das Gesamtprojekt OSEK/VDX zu steuern und zu koordinieren. Darüber hinaus gibt es ein offenes technisches Komitee, dem neben den Mitgliedern aus dem Steuergremium zurzeit über 60 Mitglieder aus den verschiedensten Bereichen der Automobilindustrie angehören. Neben der Ausarbeitung der Spezifikationen und deren Publikation wurde in letzter Zeit die Standardisierung durch ISO (International Organization for Standardization) vorangetrieben, die nun in der ISO 17356 ihren Niederschlag gefunden hat.

Bei OSEK-konformen Betriebssystemen handelt es sich um statisch konfigurierte Betriebssysteme. Dies bedeutet, dass alle Objekte des Betriebssystems zur Compile- bzw. Link-Zeit angelegt werden und sich nicht während der Laufzeit ändern lassen. Ein dynamisches Generieren von z. B. Tasks ist dadurch nicht möglich. Vorteil dieses Konzepts ist vor allem die erhöhte Betriebssicherheit. Probleme wie die aus Microsoft Windows bekannte Meldung „für diese Operation steht nicht genügend Arbeitsspeicher zur Verfügung“ können dadurch nicht auftreten. Ein weiterer Vorteil ist der geringere Verwaltungsaufwand des Kernels. Dadurch kann während der Laufzeit kein zusätzlicher Speicher angefordert werden. Alle Puffer müssen auf ihren Maximalbedarf ausgelegt werden, da OSEK hierzu keine Dienste anbietet (Lemieux, 2001).

7.6 AUTOSAR

Historie

AUTOSAR ist ein Akronym für „Automotive Open System Architecture“. In diesem Konsortium haben sich Automobilhersteller und Zulieferfirmen zusammengeschlossen. Zu den Kernpartnern gehören momentan (Stand: November 2004) die Firmen BMW Group, Bosch, Continental, DaimlerChrysler, Ford, Peugeot Citroen, Siemens VDO Automotive, Toyota sowie die Volkswagen AG. Dieser Kreis erweitert sich noch um viele weitere Teilnehmer in verschiedenen Funktionsgruppen (Neugebauer, 2004). Nähere Informationen zum aktuellen Mitgliederkreis sind auf der offiziellen Webseite des Konsortiums abrufbar, siehe www.autosar.org.

Ziel

Ziel dieser Partnerschaft ist die Definition eines offenen Standards für die Architektur von Elektronikkomponenten im Automobilbereich. Grundlage dafür ist die in diesem Bereich immer komplexer werdende Elektronik. Es wird dabei die Normierung der grundlegenden Systemfunktionen, Basisfunktionalitäten und Funktionsschnittstellen abgedeckt. Die AUTOSAR-Plattform soll dabei für die Bereiche Karosserieelektronik, Motor und Kraftübertragung, Sicherheit, Multimedia- und Telematiksysteme sowie das Mensch-Maschine-Interface als Grundlage dienen. Die in diesen Bereichen oft erheblichen Entwicklungskosten und Risiken sollen durch die Standardisierung minimiert werden (Quelle: www.autosar.org).

Bei der Entwicklung von Software für eingebettete Systeme ist durch den unterschiedlichen Aufbau der Mikrocontrollersysteme bei einem Systemwechsel immer eine Neuentwicklung der Software nötig. Es können zwar Codestücke ohne direkten Hardwarezugriff weiter verwendet werden, alle anderen Teile der Software müssen jedoch an den neuen Mikrocontroller angepasst werden. Um diesen Neuentwicklungsaufwand zu minimieren, wird an der Schnittstelle zwischen Hardware und Software eine sogenannte Basissoftware eingesetzt. Diese setzt sich aus einem Pool von Softwarekomponenten zusammen, welche eine Hardwareabstraktion bzw. Abstraktion von externen Komponenten des verwendeten Mikrocontrollersystems darstellen.

Durch die Basissoftware wird es den Anwendungsentwicklern ermöglicht, die Applikationssoftware weitestgehend unabhängig vom eingesetzten Mikrocontroller zu entwickeln. Diese Applikationen können daher erheblich einfacher für andere Mikrocontroller eingesetzt und wieder verwendet werden. Umgesetzt wird dieses Baukastenprinzip durch eine einheitlichen

API (Anwendungsprogrammierschnittstelle), die zwischen den Modulen der Basissoftware und der Applikation eingesetzt wird. Diese Unabhängigkeit vom Controller bringt jedoch auch den Nachteil mit sich, dass viele hardwarenahe Spezialfunktionen der Mikrocontroller nicht (mehr) genutzt werden können.

Die grundlegende Architektur der Software, wie sie in einem typischen Steuergerät mit Mikrocontroller eingesetzt wird, zeigt die Abbildung 7.4.

Jedes Mitglied des Konsortiums nimmt eine bestimmte Rolle mit entsprechenden Rechten und Pflichten ein. Abbildung 7.5 gibt einen Überblick über die dreigliedrige Struktur der Initiative (siehe AUTOSAR Light Version V1.5, AUTOSAR GbR, 2004).

Abb. 7.4: Grundlegende Architektur Steuergeräte-software

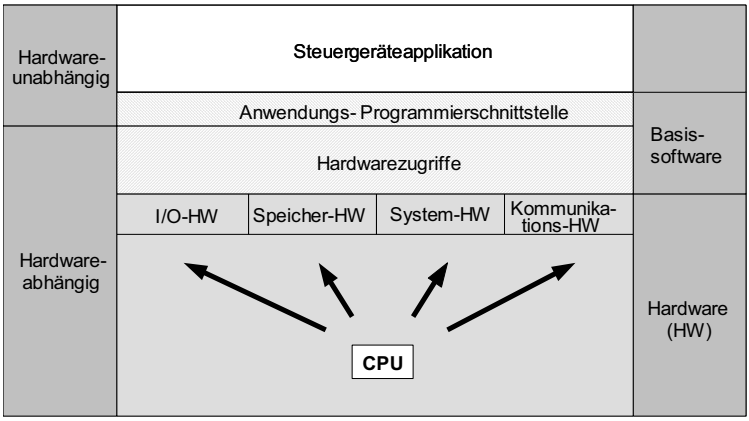
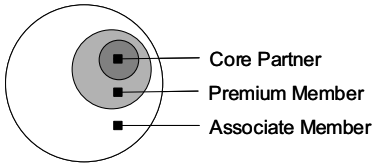


Abb. 7.5: Organisatorische Struktur der AUTOSAR-Initiative



Core Partner

Die „Core Partner" des Konsortiums leisten technische Beiträge und überwachen bzw. organisieren die Zusammenarbeit unter den Partnern in Form von Arbeitsgruppen. Sie tragen die Entscheidungsbefugnis über die zu entwickelnden Funktionsumfänge, der Zertifizierung der Ergebnisse sowie über die Veröffentlichung außerhalb der AUTOSAR-Partner.

Premium Member

Die „Premium Member" beteiligen sich an der Entwicklung innerhalb von Arbeitsgruppen, bzw. fungieren als Leitungsorgan dieser. Sie haben dabei Zugriff auf die aktuellen Entwicklungen und

Zwischenergebnisse der AUTOSAR-Technologien, auch wenn diese noch nicht endgültig spezifiziert sind. Um die Ziele innerhalb der Arbeitsgruppe erfüllen zu können, müssen diese Partner mit dem für diese Gruppe entsprechenden Know-how ausgestattet sein. Darüber hinaus sind diese Mitglieder bei Beteiligungsabstimmungen stimmberechtigt.

Associate Member

Alle „Associate Member“ haben Zugriff auf die erarbeiteten AUTOSAR-Standards, jedoch erst nach deren endgültiger Spezifikation. Sie erhalten die Informationen jedoch vor der allgemeinen Veröffentlichung. Über ein Stimmrecht verfügen diese Mitglieder nicht.

Darüber hinaus gibt es noch unterstützende Rollen, die „Development Members“ und „Attendees“. Bei den „Development Members“ handelt es ich um Kooperationspartner, die zur Aufgabenlösung innerhalb einer Arbeitsgruppe herangezogen werden. Die „Attendees“ sind Firmen-unabhängige Teilnehmer aus Forschung und Entwicklung, welche zum gegenseitigen Erfahrungsaustausch beitragen.

Die komplette Standardisierung soll bis August 2006 abgeschlossen und der Standard für jedermann veröffentlicht werden. Es sind dabei jedoch nur die Definitionen des Standards an sich frei. Sämtliche Softwareimplementierungen des Standards, die im Rahmen der Entwicklung von den Konsortiumsmitgliedern erarbeitet wurden, bleiben weiterhin deren Eigentum und werden nicht ohne Lizenzgebühren erhältlich sein.

HIS

Älter als AUTOSAR ist die Herstellerinitiative Software (HIS), in der sich die Fahrzeughersteller Audi, BMW, DaimlerChrysler, Porsche und Volkswagen zusammengeschlossen haben. Kernziel der fünf HIS-Arbeitskreise (Prozess-Assessment, Software-Test, Standard-Software, Flashbare Software sowie Simulation und Tools) ist die Vereinheitlichung von Anforderungen an Komponenten und Prozesse (Chodura, 2004). Aktuell werden die Inhalte und Zielsetzungen von HIS überprüft, um ggf. Doppelarbeiten mit AUTOSAR zu vermeiden. Für aktuelle Inhalte sei auf die Seiten von www.automotive-his.de verwiesen.

7.7 Zusammenfassung

Um den Prozess der Softwareentwicklung strukturiert und steuerbar durchführen zu können, werden sogenannte Vorgehensmodelle eingesetzt. Es existieren mehrere ganz unterschiedliche Vorgehens-

modelle, wie beispielsweise das Wasserfallmodell, das V-Modell, der Rational Unified Process sowie der Ansatz des Extreme Programming. Das V-Modell ist besonders als Vorgehensmodell für die Entwicklung eingebetteter Systeme geeignet, da es ein hohes Maß an Qualitätssicherung vorschreibt, so z. B. durch die obligatorische formale Vorgehensweise, die Erstellung zahlreicher Zwischenergebnisse (Dokumente usw.), durch Testaktivitäten auf verschiedenen Entwicklungsstufen sowie durch ein eigenes Submodul zur Qualitätssicherung.

Gerade in der automobilen Softwareentwicklung spielen neben rigiden Vorgehensmodellen auch Standards wie OSEK, AUTOSAR oder HIS (Hersteller-Initiative-Software) eine entscheidende Rolle bei der Erstellung wiederverwendbarer, hochqualitativer Steuergerätesoftware.

8 Schlussbemerkungen

Ob Mobiltelefon, PDA, Waschmaschine oder Kraftfahrzeug – immer mehr entscheidet ein immer größer werdender Anteil an Software über den Erfolg eines Produkts. Jedes Gerät soll mit jedem kommunizieren können, möglichst klein und dennoch kostengünstig sein. Intelligente Kleidung, Fahrerassistenzprogramme im Auto und Drive-by-wire sind nur einige der inzwischen schon realisierten Beispiele. Sie alle teilen die Eigenschaft, von einem sogenannten eingebetteten System gesteuert zu werden.

Unter einem solchen System versteht man ein in ein umgebendes technisches System eingebettetes und mit diesem in Wechselwirkung stehendes Computersystem. Eingebettete Systeme finden sich in fast allen modernen technischen Systemen. Beispiele sind Bremsassistenten, Fahrdynamiksteuerungen (intelligente Tempomaten, engl. Adaptive Cruise Control) und Fahrzeugumfeldschutzsysteme in Kraftfahrzeugen ebenso wie Verkehrsleitsysteme, medizinische Geräte, Telekommunikationsgeräte oder Produktionsteuerungsanlagen. Auch in der Robotik, Automatisierungstechnik, Luft- und Raumfahrttechnik und in zahlreichen Konsumgütern leisten sie gute Dienste. Sie übernehmen dort komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben und verleihen dem Gerät damit wettbewerbsentscheidende Zusatzeigenschaften.

Dies hat Embedded Systems zu einer der schnellstwachsensten Branchen auf dem Gebiet der Informatik gemacht. Gerade vor dem Hintergrund aktueller Trends der Informationstechnologie (IT) wie Outsourcing bzw. Offshoring kann die Softwareentwicklung eingebetteter System eine konstruktive Gegenmaßnahme sein, um interessante und gut bezahlte IT-Arbeitsplätze auch zukünftig im Lande halten zu können.

Hierfür ist allerdings bei vielen Verantwortlichen ein Umdenken erforderlich. Bei dem Entwurf eingebetteter Systeme ist neben der Hardwareentwurfstätigkeit mindestens gleichermaßen die Softwareentwicklung wichtig. Diese beinhaltet – teilweise spezifische – Analysemethoden, Beschreibungsverfahren, Programmiertechniken,

Vorgehensmodelle, Qualitätssicherungsmechanismen und Betriebssystemkonzepte. Hinzu kommt eine übergeordnete Sicht, die das gekoppelte Verhalten der Hardware- und Softwarekomponenten betrifft, kurz das Hardware/Software-Codesign.

In Abhängigkeit von der Anwendungsdomäne und ihrem spezifischen Kostendruck variiert der physikalische Aufbau eingebetteter Systeme stark. Er reicht von Ein-Chip-Computern mit Stückkosten von wenigen Eurocent für den Einsatz in preiswerten Konsumgütern über Steuergeräte in Flugzeugen und Automobilen bis zu modular aufgebauten, schrankgroßen Rechnern in der Automatisierungstechnik. Trotz dieser Unterschiede weisen alle eingebetteten Systeme einen gleichartigen logischen Aufbau auf, bei dem an die Stelle der Mensch-Maschine-Schnittstelle eine Schnittstelle (Sensorik, Aktuatorik) zu einem umgebenden technischen System tritt. Gerade in diesem Punkt unterscheiden sich eingebettete Systeme maßgeblich von anderen Informationssystemen. Bei letzteren werden Benutzereingaben in einem iterativen Prozess gelesen, bearbeitet und schließlich in Ausgaben transformiert.

Die Beschreibung der Systeminteraktion mit einer technischen Umgebung stellt die Entwickler eingebetter Systeme jedoch vor Herausforderungen, die sich von denen bei der Entwicklung „herkömmlicher“ Informationssysteme maßgeblich unterscheiden. Der Mensch verfügt über wesentlich flexiblere Reaktionsmöglichkeiten als eine technische Umgebung. Im Unterschied zum Menschen muss diese u. a. automatisch und sinnvoll auf Fehlermeldungen reagieren können und ggf. sogar einen Systemabsturz erkennen können. Systemantworten müssen unter Umständen in einem fest definiertem Zeitfenster erfolgen (vgl. Airbag- oder ABS-Steuerungen). Geschieht die Systemreaktion inkorrekt oder vielleicht auch nur geringfügig zu spät, kann dies fatale Folgen für System und Umgebung induzieren. Ein irrtümlicherweise ausgelöster Airbag ist ebenso nutzlos und gefährlich wie ein zu spät aufgeblasener. Letztendlich kann ein inkorrekt arbeitendes eingebettetes System Gesundheit und Leben seiner Benutzer schädigen.

In Kombination mit der Tatsache, dass einmal in den Massenmarkt ausgelieferte eingebettete Systeme nur schwer bzw. verbunden mit einem hohen Kostenaufwand korrigierbar sind, führt dies zu besonderen funktionalen und nicht-funktionalen Qualitätsanforderungen an die Hard- und besonders Softwareentwicklung eingebetteter Systeme. Allerdings haben, wie die alljährliche ADAC-Pannenstatistik zeigt, Rückrufaktionen hiesiger Premiummarken in den vergangenen Jahren deutlich zugenommen.

Die meisten eingebetteten Systeme sind reaktive Systeme, die beständig mit ihrer technischen Umgebung reagieren. Ihre gewissenhafte, formale und methodische Behandlung ist von größter praktischer Bedeutung und Kern regen wissenschaftlichen Interesses. Für einen systematischen Entwicklungsprozess werden formale und dennoch intuitive Beschreibungstechniken im Softwareentwurf (Statecharts, Esterel usw.) ebenso benötigt wie Standard-Prozessmodelle (z. B. das V-Modell XT) und Referenzarchitekturen (vgl. OSEK, AUTOSAR, HIS). Oftmals sind hier sogar Modelle nötig, die über die klassischen Modelle für Hardware oder Software Systeme hinausgehen (wie etwa Hardware/Software-Codesign oder hybride Systeme).

Ziel dieses Buches war es, einen kompakten und dennoch ausreichend tiefgehenden Überblick über ausgewählte aktuelle Spitzentechnolgien der Softwareentwicklung eingebetter Systeme zu geben, die zwar Stand der Wissenschaft sein mögen, aber – obwohl weitgehend fundiert und werkzeugunterstützt – noch nicht Stand der Technik sind. Damit soll es Studenten und Praktikern gleichermaßen als Leitfaden und Ideenmotor dienen.

Weiterer Forschungsbedarf besteht insbesondere bei der Entwicklung mathematischer Systemmodelle und formaler Semantiken für Beschreibungssprachen, der semantischen Integration unterschiedlicher Modellierungstechniken, methodischer Richtlinien, einer umfassenden Werkzeugunterstützung, insbesondere bei der Qualitätssicherung, sowie der möglichst automatischen Generierung von performanten, ablauffähigen Systemen.

Literaturverzeichnis

(Balarin et al., 1997) F. Balarin et al: Hardware-Software Co-Design of Embedded Systems – The POLIS Approach. Kluwer Academic Publishers, 1997.

(Balzert, 1998) Helmut Balzert: Lehrbuch der Software-Technik. Band 2, Spektrum Akademischer Verlag, 1998.

(Balzert, 2001) Heide Balzert: UML kompakt mit Checklisten. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001.

(Bender, 2003) Klaus Bender: Mikroelektronische Steuergeräte. ITM der Technischen Universität München, 2003.

(Bergerand, 1986) J-L. Bergerand: LUSTRE: un langage déclaratif pour le temps réel. Dissertation, Institut National Polytechnique de Grenoble, Grenoble, Frankreich, 1986.

(Berry, 1998) Gérard Berry: The Foundations of Esterel. In: G. Plotikin, C. Stirling und M. Tofte (Editoren): Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, 1998.

(Berthold, 1996) J. Berthold: Unix und Echtzeit? Sonderdruck aus Elektronik-Industrie, Ausgabe 6/1996, Seiten 106–110, Hüthig GmbH Heidelberg, 1996.

(Betzler, 2001) Klaus Betzler: Elektronische Messdatenverarbeitung. Universität Osnabrück, Juni 2001.

(Boehm, 1982) Berry Boehm: Software Engineering Economics. Prentice Hall, Seite 40, 1982.

(Broy et al., 1998) Manfred Broy, Michael von der Beeck, Ingolf Krüger: SOFTBED: Problemanalyse für ein Großverbundprojekt Systemtechnik Automobil – Software für eingebettete Systeme. Problemanalyse im Auftrag des BMBF, März 1998.

(Broy und Pree, 2003) Manfred Broy, Wolfgang Pree: Ein Wegweiser für Forschung und Lehre im Software Engineering eingebetteter Systeme. Informatik Spektrum, Springer-Verlag, Seiten 3–7, Februar 2003.

(Broy und Scholz, 1998) Manfred Broy, Peter Scholz: Anforderungs-spezifikation und Entwurf eingebetteter Softwaresysteme für Anwendungen im Kfz. In: Mobil mit Mikroelektronik und Mikrosystemtechnik, 1998.

(Boehm und Abts, 2000) Berry Boehm, C. Abts: Software Cost Estimation with Cocomo II. Prentice Hall International, 2000.

(Bryant, 1986) R.E. Bryant: Graph Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, Band 8, Seiten 677–691, 1986.

(Buchenrieder, 1995) Klaus Buchenrieder: Hardware/Software Codesign – An Annotated Bibliography. IT Press, 1995.

(Bundschuh u. Fabry, 2000) M. Bundschuh, A. Fabry: Aufwandsschätzung von IT-Projekten. MITP-Verlag, 2000.

(Butenhof, 1997) D. R. Butenhof: Programming with POSIX Threads. Addison-Wesley, 1997.

(Chodura, 2004) Hartmut Chodura, Peter-Michael Hofmann, Bernhard Kalusche, Jürgen Knoblach, Jochem Spohr und Thomas Weber: Standardisierung im Automotive-Umfeld. Elektronik Automotive, 4/2004.

(Clarke u. Emerson, 1981) E.M. Clarke und E.A. Emersion: Characterizing Properties of Parallel Programs as Fixpoints. Nummer 85 der Lecture Notes in Computer Science (LNCS), Seiten 169–181, Springer-Verlag, 1981.

(Clarke u. Schlinglo, 2001) Clarke, Schlinglo: Model Checking. In: Handbook of Automated Reasoning, Band II, Seiten 1637–1790, 2001.

(David, 1997) R. David: Modeling of Hybrid Systems Using Continuous and Hybrid Petri Nets, Proceedings of the Conference on Petri Nets and Performances Evaluation, Seiten 47–58, Saint Malo, Frankreich, Juni 1997.

(De Micheli, 1996) Giovanni De Micheli and Mariagiovanna Sami (Editoren): Hardware/Software Co-Design. NATO ASI, Series E: Applied Sciences, Band 310, Kluwer Academic Publishers, 1996.

(Devooght, 2003) D. Devooght: Betriebssysteme für mobile Systeme. Universität Koblenz-Landau, 2003.

(Di Febbraro, 2001) A. Di Febbraro, A. Giua, G. Menga: Special Issue on Hybrid Petri Nets. Discrete Event Dynamic Systems, Band 11, Nummer 1 und 2, Januar–April, 2001.

(Douglas, 1999) B. P. Douglas, G. Booch: Doing Hard Time: Developing Real-Time Systems with UML – Objects, Frameworks and Patterns. Addison-Wesley, 1999.

(Douglas, 1999a) Bruce Powel Douglas: ROPES: Rapid Object-oriented Process for Embedded Systems. I-Logix Inc., Israel, 1999.

(Dumke, 1992) Reiner Dumke: Softwareentwicklung nach Maß. Vieweg-Verlag, 1992.

(Dumke, 1995) Reiner Dumke: Modernes Software Engineering, Vieweg-Verlag, 1995.

(Emerson u. Halpen, 1986) E.A. Emerson und J.Y. Halpen: Sometimes and Not Never Revisited: On Branching versus Linear Time Temporal Logic. Journal of the ACM, Nummer 33(1), Seiten 151–178, 1986.

(Emerson, 1990) E.A. Emerson: Handbook of Theoretical Computer Science. Band B, Kapitel 16: Temporal and Modal Logic, Seiten 995–1072. The MIT Press, 1990.

(Endres, 1977) A. Endres: Analyse und Verifikation von Programmen. Oldenbourg-Verlag, 1977.

(Ernst et al., 1993) R. Ernst, J. Henkel, T. Benner: Hardware-Software Cosynthesis for Microcontrollers. IEEE Design&Test of Computers, Dezember 1993.(Flynn et al., 2001) Michael J. Flynn, S.F. Oberman: Advanced Computer Arithmetic Design. John Wiley & Sons, 2001.

(Fränzle, 2002) Martin Fränzle: Eingebettete Systeme I. Fachbereich Informatik der Carl von Ossietzky Universität Oldenburg, 2002.

(Gajski et al., 1998) D.D. Gajski, F. Fahid, S. Narayan, J. Gong: SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. IEEE Transactions on VLSI, März 1998.

(Grässle et al., 2003) Patrick Grässle, Henriette Baumann, Philippe Baumann: UML projektorientiert – Ausblick auf den neuen Standard 2.0. Galileo Press, 2003.

(Grosu, Stauner, 2002) Radu Grosu und Thomas Stauner: Modular and Visual Specification of Hybrid Systems: An Introduction to HyCharts. Formal Methods in System Design, 21(1): 5–38, 2002.

(Gunzert, 2003) Michael Gunzert: Komponentenbasierte Softwareentwicklung für sicherheitskritische eingebettete Systeme. Dissertation, Institut für Automatisierungs- und Softwaretechnik (IAS), Universität Stuttgart, Shaker Verlag, 2003.

(Gupta, DeMicheli, 1993) R.K. Gupta, G. De Micheli: Hardware-Software Cosynthesis for Digital Systems. IEEE Design&Test of Computers, September 1993(Halbwachs, 1991) Nicolas Halbwachs, P. Caspi, P. Raymond und D. Pilaud: The synchronous dataflow programming language Lustre**.** Proceedings of the IEEE, Band 79, Nummer 9, September 1991.

(Halbwachs, 1993) Nicolas Halbwachs: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, 1993.

(Harel und Pnueli, 1985) David Harel, Amir Pnueli: On the Development of Reactive Systems. In: Logics and Model of Concurrent Systems, Band 13 der NATO ASI Series F: Computer and System Sciences, Seiten 477–498, Springer-Verlag, 1985.

(Harel, und Politi 1998) David Harel, M. Politi: Modeling Reactive Systems with Statecharts. McGraw-Hill Inc., 1998.

(Harel, 1987) David Harel: Statecharts: A visual formalism for complex systems. Science of Computer Programming, Nr. 8, 1987.

(Henzinger, 1996) Thomas A. Henzinger: The Theory of Hybrid Automata. Proceedings of the 11th IEEE Symposium on Logic in Computer Science, 1996.

(Henzinger et al., 2001) Thomas A. Henzinger, Benjamin Horowitz, und Christoph M. Kirsch: Giotto: A time-triggered language for embedded programming. Proceedings of the First International Workshop on Embedded Software (EMSOFT), Seiten 166–184, Lecture Notes in Computer Science (LNCS) 2211, Springer-Verlag, 2001.

(Henzinger et al., 2003) Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido und Wolfgang Pree: From Control Models to Real-time Code using Giotto. IEEE Control Systems Magazine, Februar 2003.

(Huizing und Gerth, 1991) C. Huizing, R. Gerth: Semantics of Reactive Systems in Abstract Time. In: J.W. de Bakker, C. Huizing, W.P. de Roever und G. Rozenberg (Editoren): Real-Time – Theory in Practice, Band 600 der Lecture Notes in Computer Science (LNCS), Seiten 291–314, Niederlande, Juni 1991.

(Hürten, 1999) R. Hürten: Function Point Analysis. Expert-Verlag, 1999.

(Kelch, 2003) Rainer Kelch: Rechnergrundlagen – von der Binärlogik zum Schaltwerk. Fachbuchverlag Leipzig im Carl Hanser-Verlag, 2003.

(Kelch, 2003a) Rainer Kelch: Rechnergrundlagen – vom Rechenwerk zum Universalrechner. Fachbuchverlag Leipzig im Carl Hanser-Verlag, 2003.

(Kesten, Pnueli, 1992) Yonit Kesten, Amir Pnuelir: Timed and Hybrid Statecharts and Their Textual Representation. Lecture Notes In Computer Science (LNCS), Band 571, Seiten 591–620, Springer-Verlag, 1992.

(Klein et al., 1993) M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Harbour: A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, 1993.

(Knöll und Busse, 1991) H.-D. Knöll, J.Busse: Aufwandsschätzung von Software-Projekten in der Praxis. BI-Wissenschafts-Verlag, 1991.

(Kopetz, 1997) Hermann Kopetz: Real-Time Systems : Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 1997.

(Kopetz et al., 2002) Hermann Kopetz, Günther Bauer: The Time-Triggered Architecture. Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software, Oktober 2002.

(Krengel, 1991) Ulrich Krengel: Einführung in die Wahrscheinlichkeitstheorie und Statistik. 3. Auflage, Vieweg-Verlag, 1991

(Kumar, 1996) S. Kumar, J. Aylor, B.W. Johnson, Wm. A. Wulf: The Codesign of Embedded Systems. Kluwer Academic Publishers, 1996.

(Lemieux, 2001) Joseph Lemieux: Programming in the OSEK/VDX Environment. GMP Books, 2001.

(Liggesmeyer, 2002) Peter Liggesmeyer: Software-Qualität – Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, 2002.

(Lilge, Gralla, 1992) T. Lilge, Ch. Gralla: Drei Echtzeitbetriebssysteme für die digitale Regelung im Vergleich. Echtzeit '92 Kongressvorträge, Seiten 253–262, 1992.

(Manna et al. 1992) Z. Manna, A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems. Band 1, Springer-Verlag, 1992.

(Märtin, 2003) Christian Märtin: Einführung in die Rechnerarchitektur – Prozessoren und Systeme. Fachbuchverlag Leipzig im Carl Hanser-Verlag, 2003.

(McMillan, 1993) Kenneth L. McMillan: Symbolic Model Checking. Dissertation, Carnegie Mellon Universität, USA, 1993.

(Möller und Paulish, 1993) K.H. Möller, D.J. Paulish: Software-Metriken in der Praxis. Oldenbourg-Verlag, 1993.

(Möller, 1996) K.H. Möller: Ausgangsdaten für Qualitätsmetriken – Eine Fundgrube für Analysen. In: C. Ebert und R. Dumke (Editoren): Softwaremetriken in der Praxis. Seiten 105–116, Springer-Verlag, 1996.

(Neugebauer, 2004) Stefan Neugebauer: Entwicklung eines graphischen Mikrocontroller- und Basissoftwareemulators nach AUTOSAR-Spezifikation. Diplomarbeit, Fachbereich für Informatik, Fachhochschule Landshut, November 2004.

(Oestereich, 1997) Bernd Oestereich: Objektorientierte Softwareentwicklung mit UML. 3. Auflage, Oldenbourg-Verlag, 1997.

(Oestereich, 2004) Bernd Oestereich: Die UML 2.0 Kurzreferenz für die Praxis. 3. Auflage, Oldenbourg-Verlag, München, 2004.

(Peyton Jones, 1989) S. Peyton Jones: Parallel Implementations of Functional Pogramming Languages. The Computer Journal, 32(2): Seiten 175–186, 1989.

(Plauger, 1999) P.J. Plauger: Embedded C++. Seminar anlässlich der Embedded Systems Konferenz, Chicago, Illinois, USA, März 1999.

(Pree et al., 2003) Wolfgang Pree, Sebastian Fischmeister, Guido Menkhaus, Gerald Stieglbauer: Middleware für eingebettete Systeme. Seiten 11–13, NOEO-Wissenschaftsmagazin Salzburger Bildungs- und Forschungseinrichtungen, Ausgabe 3/2003, 2003.

(Rozenbeg, 1998) G. Rozenberg, F. Vaandrager (Editoren): Lectures on Embedded Systems. Band 1494 der Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1998.

(Rozenblit, 1995) Jerzy Rozenblit and Klaus Buchenrieder: Codesign: Computer Aided Software/Hardware Engineering. IEEE Press, 1995.

(Rosenstiel, 2003) Wolfgang Rosenstiel: Abschlussbericht DFG-Schwerpunkt-programm 1040: Entwurf und Entwurfsmethodik eingebetteter Systeme. Universität Tübingen, 1997–2003.

(Schiefer, 2004) Gunther Schiefer, Rebecca Bulander, Tamara Högler: Vergleich der Betriebssysteme mobiler Systeme. Institut AIFB, Universität Karlsruhe (TH), Untersuchung im Rahmen des Projektes MoMa/Mobiles Marketing zum Programm MobilMedia des Bundes-ministeriums für Wirtschaft und Arbeit (BMWA), 2004.

(Scholz, 1998) Peter Scholz: Design of Reactive Systems and their Distributed Implementation. Dissertation, Technischer Bericht TUM-I9821 der Fakultät für Informatik der Technischen Universität München, August 1998.

(Scholz, 2001) Peter Scholz: Incremental Design of Statechart Specifications. Science of Computer Programming 40 (2001), Seiten 119–145, Elsevier Science B.V., 2001.

(Schürmann, 2001) Bernd Schürmann: Eingebettete Systeme. AG Entwurfs-methodik eingebetteter Systeme der Universität Kaiserslautern, 2001.

(Selic, 1994) Bran Selic, Paul T. Ward, Garth Gullekson: Real Time Object Oriented Modeling. John Wiley & Sons, Februar 1994.

(Siemens, 1994) Siemens AG: Software – eine Schlüsseltechnologie mit großer Breitenwirkung. Dialog intern, Juli 1994, Siemens AG,

(Simon, 1999) D. Simon: An embedded software primer. Addison-Wesley, 1999.

(Spiegel, 2003) Spiegel Online: Flugsicherheit: Autopilot soll Terrorakte vereiteln. Spiegelnet AG, 2003.

(Spillner und Linz, 2003) Andreas Spillner, Tilo Linz: Basiswissen Softwaretest. dpunkt.verlag, Heidelberg, 2003.

(Stallings, 2000) William Stallings: Computer Organization and Architecture: Designing for Performance. 5. Ausgabe, Prentice-Hall, Upper Saddle River, New Jersey, 2000.

(Stankovic, 1998) J.A. Stankovic, K. Ramamritham: Hard Real-Time Systems. IEEE Computer Society Press, 1988.

(Stauner, 2001) Thomas Stauner: Systematic Development of Hybrid Systems. Dissertation, Technische Universität München, 2001.

(Stelter und Ulrich, 2003) Philipp Stelter und Helle D. Ulrich: Control of spontaneous and damage-induced mutagenesis by SUMO and ubiquitin conjugation. Nature, Band 425, Seiten 188–191, 11. September 2003.

(Stieglbauer, 2003) Gerald Stieglbauer: Embedded Software Engineering: Model-Based Development of Embedded Control Systems with Giotto and Simulink. Diplomarbeit, Universität Salzburg, BMW Group München, 2003.

(Storey, 1996) N. Storey: Safety-Critical Computer Systems. Prentice-Hall, 1996.

(Tanenbaum, 2001) Andrew S. Tanenbaum: Computerarchitektur – Strukturen, Konzepte, Grundlagen. Pearson Education Deutschland, 2001.

(Timmermann, 1997) Martin Timmermann und Jean-Christophe Monfret: Windows NT Real-Time Extensions: an Overview. Real-Time Magazine, 2/1997.

(Windriver, 1993) WindRiver: VxWorks Product Information. WindRiver Systems Inc., 1993.

(Wolf, 2001) W. Wolf: Computers as Components: Principles of Embedded Computing System Design. Morgan Kaufmann Publishers, 2001.

(Zargham, 1996) Mehdi R. Zargham: Computer Architecture – Single and Parallel Systems. Prentice Hall, 1996.

Sachverzeichnis

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z