UNIVERSITÄT
DUISBURG
ESSEN

Seminar im Bereich Informatik der Systeme
Entwicklung verlässlicher komp.-basierter Systeme
Sommersemester 2008

Komponentenbasierte Eingebettete Systeme

Timm Linder

20. Juli 2008

Universität Duisburg-Essen · Fachbereich Ingenieurwissenschaften
Abteilung für Informatik und angewandte Kognitionswissenschaft
Arbeitsgruppe Software Engineering – Prof. Dr. M. Heisel
betreut durch: Dipl.-Inform. Denis Hatebur, M.Sc.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Motivation und Gliederung

Nicht nur im Kontext von Desktop-Computern oder Enterprise-Servern macht der Einsatz von komponentenbasiertem Software-Engineering Sinn. Auch in einem kleineren Rahmen, nämlich in Bezug auf die sogenannten *eingebetteten Systeme*, kann die Verwendung komponentenbasierter Designs zu einer Steigerung der Produktivität, Wiederverwendbarkeit und Wartbarkeit führen.

Bevor wir im zweiten Kapitel erörtern, wie man komponentenbasiertes Software-Engineering in diesem Umfeld gewinnbringend anwenden kann, definieren wir zunächst, was wir unter dem Begriff *Eingebettete Systeme* überhaupt verstehen und welche besonderen Anforderungen eingebettete Systeme an die eingesetzte Software stellen. In Kapitel 3 stellen wir ein Muster zum Entwurf eines komponentenbasierten Designs für eingebettete Systeme vor, welches wir schließlich im vierten Kapitel auf ein Beispiel aus dem Bereich der Automobilindustrie anwenden.

1.2 Was sind eingebettete Systeme?

Es fällt nicht leicht, eine allgemeingültige Definition für alle möglichen Arten von eingebetteten Systemen (engl.: „embedded systems“) zu finden. Nach Burns und Wellings [BW01] ist ein eingebettetes System ein Computersystem, das zur Erfüllung seiner Funktionalität Informationen verarbeiten muss, wobei die Verarbeitung von Informationen nicht seine Hauptaufgabe darstellt. Li und Yao [LY03] beschreiben eingebettete Systeme als für einen speziellen Zweck entworfene Computersysteme, deren Hard- und Software eng miteinander verknüpft sind. Der Wortbestandteil „eingebettet“ deutet dabei auf die Tatsache hin, dass diese Systeme oftmals Teil einer noch viel größeren Architektur sind. Dabei kann es in der Praxis durchaus vorkommen, dass in einem System mehrere ganz unterschiedliche eingebettete Systeme nebeneinander zum Einsatz kommen [LY03].

1.3 Konkrete Beispiele für eingebettete Systeme

Schon seit Jahren halten elektronische Geräte verstärkt Einzug in unser aller Alltagsleben. Deswegen ist es nicht schwierig, Beispiele für eingebettete Systeme zu finden – sei es im Haushalt, in der Freizeit, im Verkehr oder im professionellen Umfeld.

Nahezu alle heutigen Haushaltsgeräte, egal ob Mikrowellengeräte, Wasch- oder Geschirrspülmaschinen, enthalten elektronische Schaltungen in Form von Mikrocontrollern, welche bestimmte Abläufe regeln, ohne dass der Nutzer etwas davon erfährt. Auch Autos, Züge und Flugzeuge kommen heutzutage ohne Mikrocontroller nicht mehr aus. Moderne elektronische Systeme wie ABS und ESP überwachen mithilfe unzähliger Sensoren die Fahrzeugparameter und greifen dem Fahrer unter die Arme, wenn ihr Einsatz gefordert wird. Flugzeuge wie der Airbus A320 können bei schlechter Sicht vollautomatisch landen, ohne dass es eines Eingriffs der Piloten bedarf. Die NASA-Rover Spirit und Opportunity, die uns begeisternde Panoramaaufnahmen von unserem Nachbarplaneten Mars senden, treffen teilweise ganz ohne Zutun der Forscher Entscheidungen und arbeiten selbstständig ihr tägliches Programm ab. Alle diese Systeme funktionieren nur, weil sie über die entsprechende Software verfügen. Software, die in Echtzeit teilweise sicherheitsrelevante Entscheidungen übernehmen muss, bei denen es um Leben und Tod unzähliger Menschen gehen kann – aber Software, die gleichzeitig auch an die engen technischen Grenzen gebunden ist, die ihr von der Hardware auferlegt werden.

1.4 Besondere Anforderungen an Software für eingebettete Systeme

1.4.1 Ressourcenlimitierung durch die Hardware

Auch wenn die in eingebetteten Systemen eingesetzten Mikrocontroller immer leistungsfähiger werden, sind Software-Programme für eingebettete Systeme wesentlich stärker an die ihnen von der Hardware aufgelegten Begrenzungen gebunden als dies etwa im Desktop-Bereich der Fall ist. Während viele der heute verfügbaren Desktop-Computer über mehrere Prozessorkerne und einen Arbeitsspeicher (RAM) in der Größenordnung von zwei bis vier Gigabyte verfügen, liegt die Größe des Arbeitsspeichers von Mikrocontrollern auch heute üblicherweise noch im Kilobyte-Bereich[1]. Bei der Entwicklung von Software für eingebettete Systeme müssen also die geringen Ressourcen, die der Software letztendlich zur Verfügung stehen, in den Entwurfsprozess mit einbezogen werden.

1.4.2 Zeitverhalten

Ein weiterer wichtiger Punkt, den es bei der Entwicklung von Embedded Systems-Software zu beachten gibt, ist das Echtzeitverhalten der zu entwickelnden Anwendung. Während es etwa im Falle eines Wäschetrockners nur untergeordnet eine Rolle spielt, wie schnell die Software die Benutzereingaben und weitere sensorische Informationen verarbeitet, kann es fatal sein, wenn beispielsweise das Antiblockiersystem im Auto zu spät (oder gar nicht) reagiert, weil die Auswertung der vorliegenden Daten zu lange

[1] Weit verbreitete Mikrocontroller sind z.B. der Atmel ATmega16 mit 16 KB Flash-Speicher und 1 KB SRAM [ATM] oder der PIC16F84 der Fa. Microchip mit 1 KB Flash-Speicher und 68 Bytes RAM [PIC].

dauert oder die Software fehlerhaft ist. Bestimmte eingebettete Systeme, sog. *Echtzeitsysteme*, müssen also bei ihrer Entwicklung dahingehend optimiert werden, dass die notwendigen Berechnungen stets innerhalb einer vorgegebenen Zeitspanne abgeschlossen werden [CL02a]. Man unterscheidet hier zwischen den sogenannten *weichen* und *harten* Echtzeitsystemen.

Ein System wird als *hartes* Echtzeitsystem bezeichnet, wenn Resultate, die aus bestimmten Eingaben folgen, nutzlos bzw. falsch werden, sobald eine bestimmte Zeitspanne (Deadline) überschritten wird. Je nach Einsatzgebiet des Systems kann das Überschreiten der Deadline zu Fehlfunktionen oder sogar Unfällen führen [But04]. Bei einem *weichen* Echtzeitsystem hingegen können die errechneten Ergebnisse durchaus weiter verwendet werden, allerdings kann dies zu einer verminderten Ergebnisqualität führen (z.B. verworfene Frames bei einer Videoübertragung) [Wikd].

Dem benötigten Echtzeitverhalten der Software sollte bereits zu Beginn des Entwicklungsprozesses eine hohe Bedeutung zugemessen werden.

1.5 Entwicklung von Software für eingebettete Systeme

1.5.1 Aktueller Stand der Technik: Heute übliche Vorgehensweisen

In nur wenigen der heute existierenden eingebetteten Systeme kommt komponentenbasierte Software zum Einsatz. Stattdessen handelt es sich bei der Software meistens um monolithische, plattform-abhängige Systeme, die für eine sehr spezielle Umgebung entwickelt werden und gewisse Prinzipien, die wir aus dem komponentenbasierten Design kennen, z.B. die Abstraktion und Standardisierung von Interfaces, nicht oder nur ansatzweise umsetzen [CL02b].

1.5.2 Wie komponentenbasiertes Software-Engineering die Situation verbessern kann

Durch den Einsatz von komponentenbasiertem Software-Engineering könnte man zumindest einige der Schwächen heutiger Embedded Systems-Software beseitigen. Komponentenbildung erleichtert die Wiederverwendung von bereits bestehendem Code und erhöht somit die Produktivität. Durch geeignete Abstraktion und Standardisierung kann die Portabilität erhöht werden, sodass bestenfalls nur der sog. *low-level-Code* für jede Plattform neu geschrieben werden muss. All diese Faktoren führen zu einer Beschleunigung des Entwicklungsprozesses und damit zur Reduktion der damit verbundenen Kosten.

2 Umsetzung komponentenbasierter Designs in eingebetteten Systemen

Nachdem wir definiert haben, was wir unter dem Begriff der eingebetteten Systeme verstehen und wieso die Entwicklung von komponentenbasierter Software Sinn macht, wollen wir untersuchen, welche systembedingten Aspekte es dabei zu beachten gilt und wo ggf. sogar Abstriche gemacht werden müssen.

2.1 Probleme bei der Verwendung etablierter Komponententechnologien

Aus dem Software Engineering in Bezug auf Desktop- oder Web-Applikationen kennen wir moderne Komponenten-Technologien wie *Java-Beans*, *COM* oder *CORBA* [CL02c]. Diese Frameworks erreichen auf leistungsfähigen Systemen die Ziele, die wir zu erreichen versuchen, also z.B. die gewünschte Flexibilität und Wiederverwendbarkeit. Abhängig von der zum Einsatz kommenden Hardware kann die Nutzung solcher Technologien in Embedded Systems jedoch problematisch bzw. sogar unmöglich sein. Gerade im Bereich der Echtzeitanwendungen ist eine möglichst hohe Performance entscheidend, die durch den zusätzlichen Verwaltungsaufwand (Overhead) und Speicherbedarf, den diese Frameworks mit sich bringen, negativ beeinflusst wird.

2.2 Komponenten-Komposition und -Einsatz

2.2.1 Nachteile dynamischer Komponenten-Komposition

Heute gängige Komponententechnologien wie COM oder Java-Beans bieten dem Entwickler nämlich die Möglichkeit, die Komponenten erst zur Laufzeit miteinander zu verknüpfen (z.B. mithilfe von geskripteten XML-Dateien, wie dies etwa beim im Enterprise-Bereich verwendeten Spring Framework [JHA+05] der Fall ist). Zur Umsetzung dieser Features kommen bestimmte Mechanismen wie z.B. Referenzzähler, Garbage Collection oder das Prinzip des Late Bindings zum Einsatz. Der Nachteil bei der Verwendung solcher Verfahren liegt allerdings insbesondere im höheren Speicherbedarf und der stärkeren Prozessorauslastung. Da ein zeitnahes Reaktionsverhalten vor allem bei Echtzeitanwendungen von großer Bedeutung ist, empfiehlt [CL02b], im Hinblick auf die Performance auf eine dynamische Komponenten-Komposition zu verzichten.

Eine andere Position vertritt hier [HC01], welcher die dynamische Komposition von Komponenten sogar als eine wichtige Voraussetzung für ein komponentenbasiertes Software-Design im Kontext von Embedded Systems betrachtet. Im Endeffekt muss man hier wohl abwägen zwischen der zusätzlichen Flexibilität, die das *Loose Couling* der Komponenenten bietet, und der besseren Performance, welche durch eine statische Komponentenkomposition ermöglicht wird.

2.2.2 Optimierungsmöglichkeiten bei statischer Komposition

Im Falle statischer eingebetteter Systeme, wie sie z.B. in Automobilen zum Einsatz kommen, stellt die geringere Flexibilität jedoch nur einen geringen Nachteil dar. Zwar ermöglicht eine dynamische Komposition in Bezug auf die Automobilbranche eine einfachere Aktualisierung einzelner Softwarekomponenten etwa im Rahmen eines Werkstattaufenthaltes. Dafür können jedoch bei der statischen Komposition der Komponenten, der sog. *design-time composition*, während der Kompilierung bereits bestimmte auf die Performance ausgerichtete Optimierungen vorgenommen werden. Dazu zählen unter anderem das sog. *Function Inlining*[1], die Ersetzung nachrichtengesteuerter indirekter Funktionsaufrufe durch direkte Aufrufe oder die Verlagerung konstanter Werte in den (kostengünstigeren) *Read-Only Memory (ROM)*. Auch die Verwendung bestimmter von der Zielplattform bereitgestellter Instruktionen, die die Performance verbessern, wäre hier sinnvoll. Um all diese Optimierungen sinnvoll durchführen zu können, muss jedoch der Quellcode der Komponenten vorliegen (Stichwort *white-box reuse*) [CL02b].

2.3 Schnittstellen-Spezifikationen

Aufgrund der technischen Gegebenheiten sollte man die Spezifikationen der Interfaces, die man üblicherweise während des komponentenorientierten Design-Prozesses erstellt, im Falle der eingebetteten Systeme um bestimmte *nicht-funktionale* Parameter erweitern. Beispiele für solche Parameter sind der erwartete Speicherbedarf einer Funktion bzw. Methode oder aber auch ihr Laufzeitverhalten.

Bei Echtzeitanwendungen kommt hier der Begriff der *worst-case execution time (WCET)*, also der schlimmstmöglichen Ausführungsdauer, ins Spiel, wohingegen bei weniger zeitkritischen Anwendungen eher die durchschnittliche Ausführungsdauer von Belang ist. [HC01] erläutert die Problematiken, auf die man bei der Bestimmung des Ressourcenbedarfs einer Komponente stößt.

[1] Gemeint ist hiermit die Vermeidung unnötiger Funktionsaufrufe und des damit verbundenen Overheads, indem der Code der aufzurufenden Funktion direkt in die aufrufende Funktion mit einkompiliert wird.

2.4 Kontextabhängigkeiten

2.4.1 Abhängigkeit von der Programmiersprache

Moderne Komponentenmodelle wie etwa CORBA sind unabhängig von der verwendeten Programmiersprache bzw. bieten für jede der gängigen Programmiersprachen eine eigene Implementierung an. Dazu spezifiziert der Entwickler die Schnittstellen der Komponenten (in CORBA etwa mithilfe von *IDL*[2]) und erzeugt aus dieser abstrakten Schnittstellenspezifikation mithilfe eines Compilers den Code für die jeweilige Sprache. Im Allgemeinen ist es in Bezug auf eingebettete Systeme jedoch sinnvoll, auf diese Flexibilität zugunsten einer höheren Performance zu verzichten [CL02b].

2.4.2 Plattform-Abhängigkeit

Die Wiederverwendbarkeit ist eines der Hauptargumente für die Wahl eines komponentenorientierten Softwaredesigns. Um sie zu erhöhen, sollten Softwarekomponenten nach Möglichkeit unabhängig von der verwendeten Plattform sein. Gerade im Bereich der Embedded Systems ist dies von großer Bedeutung, da die Entwicklung der Hardware sehr rasant vonstatten geht, während die Software oft über Jahre hinweg weiterentwickelt wird. In Hinsicht auf eine größtmögliche Wiederverwendbarkeit ist es prinzipiell ratsam, die Komponenten so abstrakt wie möglich zu halten[3]. Ein anderer Weg wäre die Nutzung einer virtuellen Maschine, wie wir sie von der Programmiersprache Java her kennen. Beiden Ansätzen ist jedoch wieder der recht hohe Overhead gemein, den es insbesondere in zeitkritischen Anwendungen um jeden Preis zu vermeiden gilt.

[CL02b] rät daher zu einer rein quellcode-bezogenen Portabilität, sodass die Software-Komponenten für jede Zielplattform neu kompiliert werden müssen. Doch auch hier gilt es, bestimmte Einschränkungen zu treffen, um überhaupt eine Portierung zu ermöglichen: So sollte man sich auf eine bestimmte Programmiersprache (z.B. ANSI-C) einigen, die von allen denkbaren Zielplattformen unterstützt wird. Üblicherweise bieten derartige Programmiersprachen bestimmte Standardbibliotheken, die grundlegende Aufgaben, z.B. in Bezug auf Ein- und Ausgabe, übernehmen. Die etwa von Java bekannte binäre Portierbarkeit wäre natürlich wünschenswert, ist aber nicht unbedingt erforderlich.

[2] Interface Definition Language

[3] Low-level-Code innerhalb der Komponenten sollte also vermieden werden, oder aber ausgelagert werden in dedizierte Komponenten, die hardwarenahe Aufgaben übernehmen.

3 Ein Entwurfsmuster für die Entwicklung von Software für eingebettete Systeme

Nun, wo wir die wichtigsten Aspekte kennen, die es bei der Entwicklung von Software für eingebettete Systeme zu beachten gilt, wollen wir eine Vorgehensweise beschreiben, die unter Berücksichtigung dieser Aspekte zu erläutern versucht, wie ein komponentenbasiertes Design von Embedded Systems-Software aussehen sollte. Dieses Entwurfsmuster setzt sich aus einer Reihe von unterschiedlichen Elementen zusammen, wie wir sie teilweise schon aus dem komponentenbasierten Software-Engineering für Desktop- und Enterprise-Anwendungen kennen.

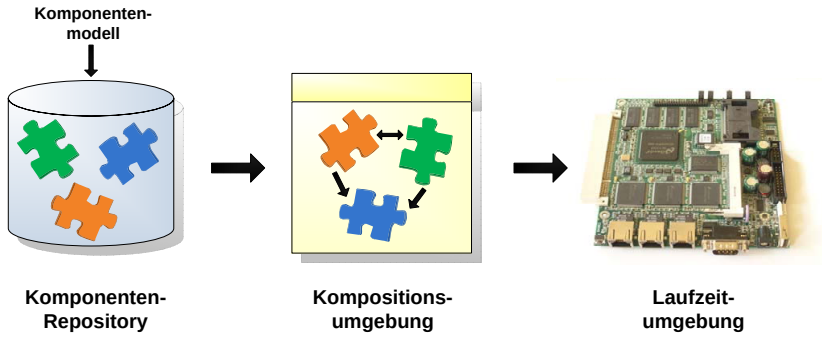

Abbildung 3.1: Komponentenbasiertes Entwurfsmuster für Embedded Systems. (Foto: Soekris net4801, Embedded System für Netzwerkanwendungen [Wikc])

3.1 Komponentenmodell

Das Komponentenmodell stellt die unterste Schicht in unserem Entwurfsmuster dar. Es erlaubt uns, die von den jeweiligen Komponenten verwendeten Interfaces möglichst effizient, aber vor allem exakt zu beschreiben, was nichtfunktionale Parameter wie etwa die *worst-case execution time* betrifft. [HC01] schlägt hier die Einführung sogenann-

ter *end-to-end constraints*[1] vor, also zeitliche Beschränkungen, die bei der Interaktion bestimmter Komponenten erfüllt sein müssen. Dabei sollte darauf geachtet werden, dass diese Beschränkungen möglichst unabhängig von der verwendeten Plattform spezifiziert werden. Auch erscheint es sinnvoll, diese Beschränkungen nicht innerhalb der Komponenten selbst, sondern eher zwischen den Komponenten anzusiedeln, da sie die Interaktion der Komponenten miteinander betreffen [HC00].

3.2 Komponenten-Repository

Ein wichtiger Faktor für die effiziente Umsetzung eines komponentenbasierten Designs ist das *Komponenten-Repository*. Hier werden die Implementierungen aller Komponenten – abhängig von der jeweiligen Zielplattform – mitsamt der nötigen Versionsinformationen und Schnittstellenbeschreibungen abgelegt. Zu dieser Beschreibung gehören, wie zuvor bereits erwähnt, auch nicht-funktionale Parameter, die zum Beispiel über das Laufzeitverhalten der jeweiligen Komponenten Auskunft geben. Auch ganze Architekturstile können hier hinterlegt werden.

3.3 Kompositionsumgebung

Nachdem die verschiedenen Komponenten erstellt und ggf. getestet worden sind, müssen sie auf irgendeine Weise miteinander verknüpft werden. Dazu sind verschiedene Techniken denkbar. Neben einer skriptbasierten Komponentenkomposition erachtet [HC01] aus Gründen der Effizienz insbesondere die Nutzung grafischer Benutzeroberflächen als sinnvoll. Die Kompositionsumgebung hat zudem dafür Sorge zu tragen, dass bestimmte Vorgaben, die von den einzelnen Komponenten gemacht werden, eingehalten werden (z.B. in Bezug auf das Zeitverhalten). Auch die Visualisierung ressourcentechnischer Engpässe wäre in Hinsicht auf Embedded Systems ein sinnvolles Feature.

3.4 Laufzeitumgebung

Der fertig erzeugte Code kommt schließlich in der Laufzeitumgebung zum Einsatz. Vor allem bei Echtzeitsystemen ist hier die Leistungsfähigkeit der Laufzeitumgebung entscheidend. So sollte die Laufzeitumgebung selber recht sparsam mit dem zur Verfügung stehenden Speicher umgehen, damit dieser vom eigentlichen Anwendungsprogramm verwendet werden kann. Die Laufzeitumgebung sollte hardwarenahe Programmiersprachen wie z.B. C unterstützen, gleichzeitig aber auch eine passende Abstraktionsschicht bieten, sodass plattformabhängige Operationen (z.B. Ein-/Ausgabe) von speziellen Bibliotheken übernommen werden und nicht von der Anwendung selber implementiert werden müssen.

[1] Gemeint ist hiermit die Zeit, die verstreicht, bis auf eine Stimulation der Komponente von außen eine Reaktion der Komponente erfolgt.

4 Ein Anwendungsbeispiel: Antiblockiersysteme in Automobilen

Am Beispiel des Antiblockiersystems (ABS), das in praktisch jedem heutigen Automobil zur Erhöhung der Fahrsicherheit in Gefahrensituationen zum Einsatz kommt, wollen wir zeigen, wie ein komponentenbasiertes Softwaredesign für eingebettete Systeme aussehen könnte. Dazu erläutern wir zunächst kurz die technischen Hintergründe und gehen dann auf die in Kapitel 3 besprochenen Aspekte ein.

4.1 Technische Funktionsweise

Durch zu starke Betätigung der Bremsen im Auto kann es passieren, dass die Räder blockieren und das Fahrzeug somit nicht mehr auf Lenkbewegungen des Fahrers reagiert. Bevor das ABS erfunden wurde, musste der Fahrer hier den Bremsdruck manuell reduzieren bzw. die Bremse pulsieren lassen, um einerseits die Kontrolle über das Fahrzeug zu behalten, andererseits aber den benötigten Bremsweg möglichst gering zu halten. Seit den 1970er Jahren übernimmt das Antiblockiersystem diese Aufgabe und ist aus heutigen Serienfahrzeugen nicht mehr wegzudenken. [Wika]

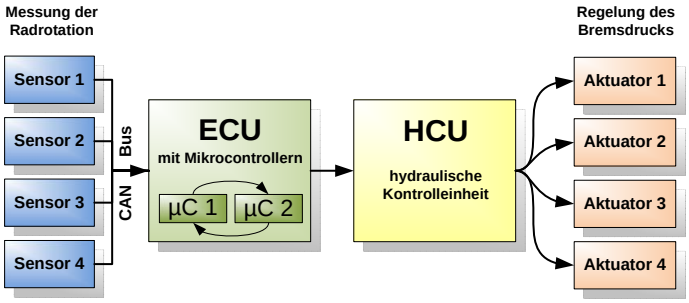

Abbildung 4.1: Schematische Darstellung der Hardware eines Antiblockiersystems

4.1.1 Hardware

Die Hardware des ABS besteht im wesentlichen aus einer elektronischen Kontrolleinheit, der sogenannten *ECU (Electronic Control Unit)*, vier *Sensoren*, die die Rotationsgeschwindigkeit der einzelnen Räder messen, sowie einem *System aus hydraulischen Ventilen und Aktuatoren*, die zur Regulation des Bremsdrucks dienen. Die Aktuatoren werden von einer hydraulischen Kontrolleinheit *(HCU)* angesteuert. [Wika]

Die ECU überwacht die Geschwindigkeit, mit der sich die Räder drehen. Sollte eines der Räder durch zu starkes Bremsen blockieren, d.h. die Rotationsgeschwindigkeit weicht von der der anderen Räder sehr stark nach unten ab, so reduziert das System durch Öffnen eines Ventils automatisch den Bremsdruck, der an dem betroffenen Rad anliegt. Dadurch erhöht sich die Rotationsgeschwindigkeit des Rades wieder. Sollte sich das Rad im Vergleich zu den anderen Rädern dann zu schnell drehen, wird der hydraulische Bremsdruck durch Schließen des Ventils wieder gesteigert [Wika]. Dieses Öffnen und Schließen der Ventile erfolgt bei heutigen Systemen bis zu 20 mal in der Sekunde, wodurch das charakteristische Pulsieren des Bremspedals bei Vollbremsungen zustande kommt.

Das Herzstück des Systems, die elektronische Kontrolleinheit, besteht – wie in Abbildung 4.1 ersichtlich – heutzutage aus mindestens zwei Mikrocontrollern, die immer gleichzeitig aktiv sind und sich gegenseitig überwachen. Dadurch wird eine gewisse Redundanz erzielt, die bei einem derart sicherheitskritischen System natürlich erwünscht ist. Die Anbindung der Sensoren an die ECU erfolgt bei den meisten heutigen Antiblockiersystemen über einen speziellen Datenbus, den sog. *CAN-Bus*[1], während die HCU auf direktem Wege mit der ECU verbunden ist [Wika].

4.1.2 Software

Innerhalb der ECU kommt ein Softwaresystem zum Einsatz, das teils ganz verschiedene Aufgaben übernimmt. Zunächst müssen natürlich die Signale, welche die vier Sensoren an den Rädern liefern, eingelesen und aufbereitet werden. Anschließend werden die eingehenden Daten verarbeitet. Ein Algorithmus vergleicht die Drehgeschwindigkeiten der Räder miteinander und teilt dann bei Bedarf dem Steuerungssystem für die hydraulische Kontrolleinheit mit, ob die jeweiligen Ventile geöffnet oder geschlossen werden müssen. Dieses System steuert dann die jeweiligen Ausgänge an, die über einen Stromkreis direkt mit den Aktuatoren verbunden sind [Wika].

Parallel dazu läuft auf beiden ECU-Mikrocontrollern ein Diagnosesystem, welches Fehler innerhalb des Systems aufspüren und dann ggf. die passenden Maßnahmen ergreifen soll [Wika]. Ein solcher Fehler könnte z.B. ein Ausfall eines Sensors sein (welcher sich durch eine kontinuierliche Rotationsgeschwindigkeit von 0 bemerkbar machen würde). Auch ungültige Anweisungen an die hydraulische Kontrolleinheit könnte das System

[1] **C**ontroller-**A**rea **N**etwork Bus, 1983 von der Firmen Bosch und Intel zur Vernetzung von Mikrocontrollern in Automobilen entwickelt [Wikb].

feststellen und dann z.B. zur Sicherheit das komplette ABS deaktivieren (was der Fahrer an einer aufleuchtenden Kontrollleuchte erkennen würde). Eine Schnittstelle innerhalb des Systems ermöglicht es zudem, externe Diagnosegeräte anzuschließen, um etwa die Fehlersuche in der Werkstatt zu erleichtern.

4.2 Besondere Anforderungen

4.2.1 Echtzeitverhalten

Beim Antiblockiersystem handelt es sich offensichtlich um ein höchst sicherheitsrelevantes Fahrzeugsystem. Aus diesem Grunde muss das System äußert ausfallsicher sein, sodass – wie oben erwähnt – u. a. Redundanz eine große Rolle spielt. Da sich Gefahrensituationen im Straßenverkehr aber oftmals innerhalb von Sekundenbruchteilen abspielen, ist es außerdem von großer Wichtigkeit, dass das System möglichst zeitnah reagiert. Technisch gesehen ist es sogar zwingend notwendig, dass die passenden Reaktionen auf die Sensor-Inputs innerhalb einer bestimmten Zeitspanne zur Verfügung stehen, da die Reaktionen ansonsten nutzlos würden: Das System muss nämlich frühzeitig erkennen, wann ein Rad zu blockieren beginnt, und dann sofort das passende Ventil öffnen. Wenn das Rad bereits blockiert, kann es unter Umständen bereits zu spät sein, weil der Fahrer dann schon die Kontrolle über das Fahrzeug verloren hat [Wika].

Aus diesem Grund handelt es sich bei den Kernkomponenten des Antiblockiersystems um ein *hartes Echtzeitsystem*. Dem gegenüber stehen die Diagnosefunktionen des Systems, die eine geringere Priorität besitzen [Wika] und deshalb zu den *weichen* Echtzeitsystemen zählen.

4.2.2 Geringe Systemressourcen

Hardwaremäßig stehen der Software eines Antiblockiersystems nur eng begrenzte Ressourcen zur Verfügung. Neben Maßnahmen zur Kostensenkung ist hier nämlich auch die Ausfallsicherheit ein wichtiger Faktor: Die in der ECU verwendeten Mikrocontroller sind, was ihre Leistungsfähigkeit betrifft, oftmals sehr klein dimensioniert, um die entstehende Verlustwärme und damit das Ausfallrisiko zu reduzieren. Zudem sollte ein Antiblockiersystem niemals an die Grenzen der Hardware stoßen, was mitunter ein Fehlverhalten oder zu späte Reaktionen des Systems zur Folge haben könnte [Wika].

4.3 Vorschlag für ein komponentenbasiertes Softwaredesign

Nachdem wir die Funktionsweise und die besonderen Ansprüche eines Antiblockiersystems erläutert haben, wollen wir untersuchen, wie man das Prinzip des komponentenbasierten Software-Engineerings auf die Software eines solchen Systems erfolgreich

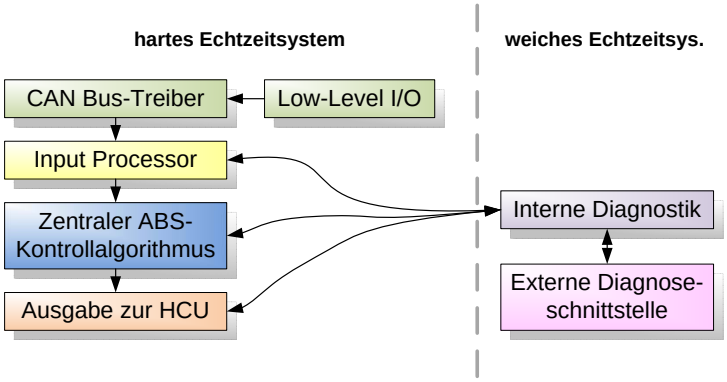

Abbildung 4.2: Unterteilung der ABS-Software in Komponenten

anwenden könnte.

4.3.1 Unterteilung der Software in Komponenten nach Funktionalität

Um unser Entwurfsmuster aus Kapitel 3 verwenden zu können, gilt es zunächst zu entscheiden, in welche Komponenten man das Softwaresystem unterteilt. Es erscheint hier sinnvoll, jedem Aufgabengebiet der ECU-Software eine oder mehrere eigene Komponente(n) zuzuordnen. Abbildung 4.2 veranschaulicht die verschiedenen Teilsysteme unserer Softwarearchitektur. Wichtig ist, dass hardwarespezifische Subsysteme, z.B. die Verarbeitung der Sensor-Inputs und die damit verbundene Kommunikation über den CAN-Bus, in separate Komponenten ausgelagert werden, um die Wiederverwendbarkeit der Komponenten zu erhöhen.

4.3.2 Einrichtung des Komponenten-Repositorys

Legt man etwa den CAN-Bus-Treiber als separate Komponente im Repository ab, könnte dieser in ganz anderen Systemen des Fahrzeugs erneut Verwendung finden. Dazu müssen natürlich die entsprechenden nicht-funktionalen Parameter, wie z.B. die durch die im Treiber ablaufenden internen Prozesse verursachte Latenzzeit, mit abgespeichert werden, sodass an anderer Stelle überprüft werden kann, ob die gegebenen *Real-Time Constraints* überhaupt eingehalten werden können. Auch sollte die Logik des eigentlichen ABS-Kontrollalgorithmus möglichst abstrakt gehalten werden, um etwa einen Einsatz in anderen Fahrzeugtypen zu gestatten und die Notwendigkeit einer kompletten Neuentwicklung von vorneherein zu vermeiden. Fahrzeugspezifische Parameter (Raddurchmesser etc.), die mit in die Berechnung einfließen, könnten z.B. über Skripte bei der Komposition der Komponenten mit eingebunden werden. Der Einsatz dynamischer Konfigurationsdateien, etwa im XML-Format, macht hier aus performan-

cetechnischen Gründen keinen Sinn, ist aufgrund der statischen Natur des Systems aber auch nicht erforderlich.

4.3.3 Überlegungen bezüglich der Laufzeitumgebung

Wie in 4.2.1 beschrieben, besteht unsere Softwarearchitektur nicht nur aus den Ein- und Ausgabe-Komponenten sowie dem eigentlichen Kontrollalgorithmus, sondern auch aus bestimmten Diagnosefunktionen, welche von der zu veranschlagenden Rechenzeit her eine weitaus geringere Priorität einnehmen als die Hauptkomponenten. Da das Diagnosesystem die anderen Komponenten ständig und ununterbrochen überwachen soll, macht hier eventuell der Einsatz eines Multi-Threading-Systems[2] Sinn.

Synchrone Programmiersprachen[3] wie *Esterel* oder *Lustre* (beide [Hal98]) können bei der Implementation nebenläufiger Systeme von Nutzen sein. Dazu wird der reaktive Teil der Software, welcher auf ankommende Signale reagiert, in einer solchen synchronen Sprache spezifiziert und anschließend etwa in Form eines endlichen Automaten in den Code einer Host-Sprache wie z.B. C übersetzt. Synchrone Softwarekomponenten können, da sie ausschließlich über Signale miteinander kommunizieren, auch existieren, ohne das jeweilige Gegenüber zu kennen [CL02b]. Weil der etwa vom Esterel-Compiler erzeugte Code insbesondere im Falle von Nebenläufigkeit wesentlich umfangreicher ist als handgeschriebener C-Code, muss man hier jedoch den Nachteil des höheren Speicherbedarfs in Kauf nehmen [CL02b].

[2] Man lässt also mehrere Ausführungsstränge der Anwendung gleichzeitig ablaufen. Während die Berechnung der neuen Parameter, die an die hydraulische Kontrolleinheit gesendet werden sollen, im Gange ist, überprüft gleichzeitig das Diagnosesystem alle vorhandenen Parameter und greift ggf. ein.

[3] Synchrone Programmiersprachen basieren auf der sog. *synchronen Hypothese*. Diese besagt im Wesentlichen, dass Berechnungen unendlich schnell vonstatten gehen und dass jede Reaktion auf eine Eingabe atomar ist und im selben Moment wie die Eingabe selber erfolgt [CL02b].

5 Fazit und Ausblick

In dieser Ausarbeitung haben wir uns mit der Frage auseinandergesetzt, wie man das im Desktop- und Enterprise-Bereich bereits weit verbreitete komponentenbasierte Software-Design auch auf das Umfeld der eingebetteten Systeme übertragen kann. Dabei haben wir die unterschiedlichen Problemstellungen, die es hierbei zu beachten gibt, beleuchtet, mögliche Lösungswege aufgezeigt und diese schließlich auf ein Fallbeispiel angewendet. Der am schwersten wiegende Faktor ist hier sicherlich die Limitierung der Ressourcen, welche die Hardware von eingebetten Systemen der zugehörigen Software auferlegt. Um trotzdem komponentenbasiertes Software-Engineering im Bereich der Embedded Systems betreiben zu können, gilt es, hier einige Abstriche bezüglich der Flexibilität unserer Softwarearchitekturen in Kauf zu nehmen, ohne aber die Wiederverwendbarkeit der Komponenten als solche stärker in Mitleidenschaft zu ziehen. Gerade der letztgenannte Aspekt der Wiederverwendbarkeit ist einer der Hauptgründe, die dafür sprechen, die Entwicklung komponentenbasierter Designs weiter voranzutreiben.

Legt man seinen Fokus auf Echtzeitsysteme, so sind zur erfolgreichen Umsetzung eines komponentenbasierten Designs weitere Punkte zu beachten. Die zusätzlichen Probleme, die sich unter anderem bei der Nutzung von Nebenläufigkeit ergeben, lassen sich mitunter durch Verwendung einer synchronen Programmiersprache wie Esterel lösen. Um miteinander verbundene Systeme mit völlig unterschiedlichen Real-Time Constraints in Einklang zu bringen, wie etwa die hydraulische Kontrolleinheit und das Diagnosesystem im Fallbeispiel des Antiblockiersystems, kann die Anwendung des *Strategy-Patterns* [Bus98] sinnvoll sein, auf das wir hier jedoch nicht genauer eingehen können.

6 Literaturverzeichnis

[ATM] Atmel ATMega16. Data Sheet, http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf.

[Bus98] F. Buschmann. Real-time constraints as strategies, 1998.

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CL02a] Ivica Crnkovic and Magnis Larson, editors. *Builing reliable component-based software systems*, chapter 14, pages 265–280. ARTECH HOUSE, INC., 2002. Testing Reusable Software Components in Safety-Critical Real-Time Systems.

[CL02b] Ivica Crnkovic and Magnis Larson, editors. *Builing reliable component-based software systems*, chapter 16, pages 299–324. ARTECH HOUSE, INC., 2002. Component-Based Embedded Systems.

[CL02c] Ivica Crnkovic and Magnis Larson, editors. *Builing reliable component-based software systems*, chapter 4, pages 57–86. ARTECH HOUSE, INC., 2002. Component Models and Technology.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.

[HC00] D. K. Hammer and M. R. V. Chaudron. Towards component-based architecting for resource constraint systems. In *Proc. 4rth International Software Architecture Workshop (ISAW-4)*, Limerick, Ireland, 2000.

[HC01] D. K. Hammer and M. R. V. Chaudron. Component-based software engineering for resource-constraint systems: What are the needs? In *WORDS '01: Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, page 91, Washington, DC, USA, 2001. IEEE Computer Society.

[JHA+05] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.

[LY03] Qing Li and Caroline Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, San Francisco, USA, 2003.

[PIC] Microchip PIC16F8x. Data Sheet, http://www.datasheetarchive.com/pdf/2757692.pdf.

[Wika] Wikipedia. *Anti-lock braking system*. http://en.wikipedia.org/w/index.php?title=Anti-lock_braking_system&oldid=224720809.

[Wikb] Wikipedia. *Controller-Area Network*. http://en.wikipedia.org/w/index.php?title=Controller-area_network&oldid=225797003.

[Wikc] Wikipedia. *Embedded system*. http://en.wikipedia.org/w/index.php?title=Embedded_system&oldid=225828757.

[Wikd] Wikipedia. *Real-time computing*. http://en.wikipedia.org/wiki/Real-time_computing.