Komponentenbasierte eingebettete Systeme

Vortrag im Rahmen des Seminars "Entwicklung verlässlicher komponentenbasierter Systeme" (Sommersemester 2008)

Timm Linder

University Duisburg-Essen, Faculty of Engineering,
Department of Computational and Cognitive Sciences,
Working Group Software Engineering, Prof. Dr. M. Heisel

28. Juli 2008

Inhaltsübersicht

Einleitung/Motivation

Was sind „eingebettete Systeme“?

Definition: Eingebettete Systeme

- Computersystem, das Informationen verarbeitet, aber eine andere Hauptaufgabe als die der Informationsverarbeitung besitzt [BW01]
- für einen speziellen Zweck entworfene Systeme, bei denen Hard- und Software eng miteinander verknüpft sind [LY03]
- Wortbestandteil „eingebettet“ (engl. „embedded“): oftmals existieren in einem größeren System mehrere eingebettete Systeme nebeneinander

Einleitung/Motivation

Was sind „eingebettete Systeme“?

Beispiele für eingebettete Systeme

- **Haushalt:** Waschmaschine, Geschirrspüler, Mikrowelle
- **Freizeit:** Handy, Spielkonsole
- **Verkehr:** Auto (ABS, ESP), Flugzeug (Autopilot)
- **Industrie:** Fertigungsanlagen, Roboter
- **IT:** Router, Switches, Hubs
- **Wissenschaft:** Mars-Rover, Satelliten, Raumsonden

Einleitung/Motivation

Besondere Anforderungen an Software für eingebettete Systeme

Unterschiede zu „normaler“ Anwendungssoftware

- **limitierte Ressourcen:** Oft Systeme mit nur wenig Arbeitsspeicher (Kilobyte-Bereich), langsame CPU/Mikrocontroller
- **Zeitverhalten:** teilweise (sicherheitskritische) Echtzeitanwendungen

Ergo: Trotz der geringen verfügbaren Ressourcen müssen die meisten eingebetteten Systeme eine **gute Performance liefern** und gleichzeitig **zuverlässig sein** [CL02].

Einleitung/Motivation

Wieso CBSE in eingebetteten Systemen?

Allg. Vorteile komponentenbasierter Systeme

- Wiederverwendbarkeit
  ($\rightsquigarrow$ Effizienz, Kostensenkung)
- Wartbarkeit

Heutiger Stand der Technik bei eingebetteten Systemen

- überwiegend monolithische, unflexible Software
- Plattformabhängigkeit ($\rightsquigarrow$ geringe Wiederverwendbarkeit)
- kaum Abstraktion ($\rightsquigarrow$ schlechte Wartbarkeit)

Gliederung

CBSE in eingebetteten Systemen

Probleme bei der Verwendung etablierter Technologien

Gängige Frameworks

- COM, CORBA, Java-Beans/EJB
- schaffen **Flexibilität**, **Plattformunabhängigkeit**, erhöhen die **Wiederverwendbarkeit**

Nachteile

- Performance-Overhead durch Verwaltungs-Infrastruktur
- höherer Speicherbedarf
- fehlende Unterstützung für Embedded Systems

CBSE in eingebetteten Systemen

Schnittstellenspezifikationen

Erweiterung der Schnittstellenspezifikationen

Erweiterung der funktionalen Beschreibung der Komponentenschnittstellen um bestimmte **nicht-funktionale Parameter**, z.B.

- erwarteter Speicherbedarf
- max./durchschnittl. Anzahl an benötigten CPU-Zyklen zur Ausführung einer Funktion
- Laufzeitverhalten: *average/worst case execution time (WCET)*

Bestimmung derartiger Parameter ist nicht immer unproblematisch [HC01].

CBSE in eingebetteten Systemen

Komponentenkomposition I

Dynamische Komposition (*Loose Coupling*) von Komponenten

Erzielung höherer **Flexibilität**, indem man die Komponenten **erst zur Laufzeit** miteinander verknüpft.

Eine Möglichkeit dafür: Verwendung von XML-Dateien (oder anderen geskripteten Dateien) zur Konfiguration

$\rightsquigarrow$ leichte Anpassbarkeit
$\rightsquigarrow$ einfacher Austausch von Komponenten

CBSE in eingebetteten Systemen

Komponentenkomposition II

Verwendete Prinzipien bei dynamischer Komposition

- Late Binding
- Garbage Collection
- Reference Counting
- u.a.

Nachteile von dyn. Komposition bei Embedded Systems

Höherer **Speicherbedarf** und stärkere **CPU-Auslastung** durch zusätzlichen Verwaltungsaufwand. Zum Teil Verschwendung kostbarer Ressourcen durch zu langsame Garbage Collection.

CBSE in eingebetteten Systemen

Komponentenkomposition III

Optimierungsmöglichkeiten bei statischer Komposition

Design-time composition ermöglicht zusätzliche Optimierungen während der Kompilierung, z.B.

- *Function Inlining*
- Verlagerung konstanter Werte in den kostengünstigeren *Read-Only Memory (ROM)*
- Ersetzung nachrichtengesteuerter Funktionsaufrufe durch direkte Aufrufe

Aber: Dazu muss der Quellcode vorliegen ($\rightsquigarrow$ *white-box reuse*)

CBSE in eingebetteten Systemen

Komponentenkomposition IV

Statische oder dynamische Komposition?

Man muss deshalb, abhängig von der Leistungsfähigkeit der Hardware, zwischen **Performance** und **Flexibilität** abwägen.

Statische Komposition ermöglicht eine **zusätzliche Optimierung** des entstehenden Codes [CL02].

CBSE in eingebetteten Systemen

Kontextabhängigkeiten I

Abhängigkeit von der Programmiersprache

COM, CORBA und Co. sind unabhängig von der Programmiersprache (z.B. IDL bei CORBA).

Bei eingebetteten Systemen (größtenteils statisch) kann man auf diese Funktionalität verzichten [CL02].

CBSE in eingebetteten Systemen

Kontextabhängigkeiten I

Plattformabhängigkeit

Plattformabhängigkeit schränkt Wiederverwendbarkeit von Komponenten ein. Deshalb:

- Komponenten abstrahieren, eigene Komponenten für plattformabhängige low-level-Operationen
- Rückgriff auf bestehende Multi-Plattform-Bibliotheken
- alternativ: Verwendung einer **virtuellen Maschine** (vgl. Java) $\rightsquigarrow$ Performance!?

Aus Performancegründen: Gewährleistung einer rein **quellcode-bezogenen Portabilität** [CL02]. Komponenten müssen für jede Plattform neu kompiliert werden. **Binäre Portierbarkeit** zwar wünschenswert, aber nicht notwendig.

Gliederung

Ein Entwurfsmuster
Grundidee

Vier verschiedene Ebenen, die für uns eine Rolle spielen:

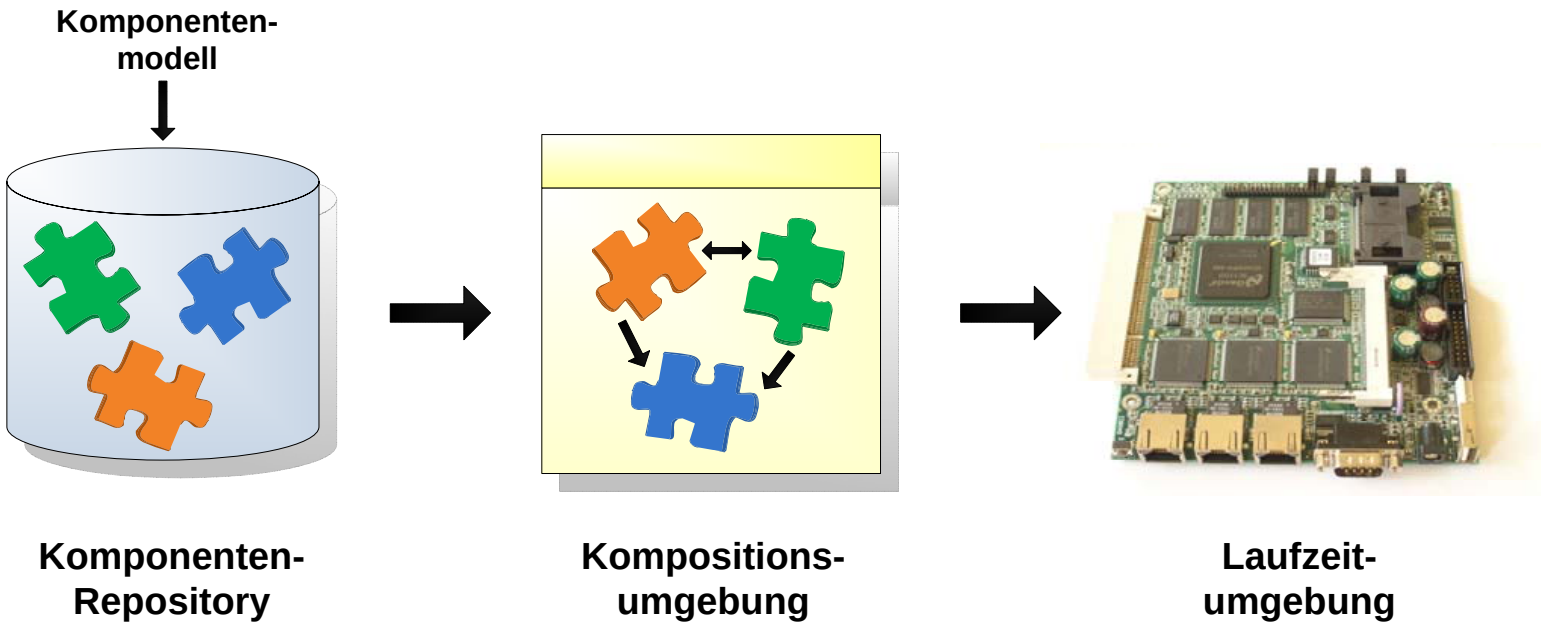

Abbildung: Komp.-basiertes Entwurfsmuster für Embedded Systems

Ein Entwurfsmuster
Das Komponentenmodell

Inhalt des **Komponentenmodells**

- erlaubt präzise Beschreibung der **Komponentenschnittstellen**
- Beschreibung der funktionalen und nicht-funktionalen Parameter (z.B. Laufzeitverhalten, Speicherbedarf)
- nicht-funktionale Parameter ggf. besser als „Komponentenbeziehung“ innerhalb des *Repositorys* darstellen, wenn die Parameter die **Interaktion** der Komponenten betreffen

Ein Entwurfsmuster
Komponenten-Repository

Sinn und Zweck des **Komponenten-Repositorys**

- dient dem **Ablegen der Komponenten** mitsamt Ihrer (plattformabhängigen) Implementierungen
- Speicherung von **Versionsinformationen** und **nicht-funktionalen Parametern** (z.B. sog. *end-to-end constraints*)
- Ablegen von **Architekturstilen**

Ein Entwurfsmuster
Kompositionsumgebung

Aufgaben der **Kompositionsumgebung**

- **Verknüpfung** der fertigen Komponenten miteinander
- entweder **skriptbasiert** oder per **grafischer Benutzeroberfläche** (wünschenswert!)
- erzeugt aus den Einzelkomponenten den finalen Code
- Überprüfung der Einhaltung von Beschränkungen (zeitlich, Speicherbedarf, ...)

Ein Entwurfsmuster
Laufzeitumgebung

Die **Laufzeitumgebung**

- dient der **Ausführung** des fertigen Codes
- **Leistungsfähigkeit** sehr entscheidend, sollte speicherschonend sein
- enthält ggf. eine **virtuelle Maschine**
- sollte eine **Abstraktionsschicht** für Low-Level-Operationen bieten (z.B. I/O)
- für Echtzeit-Systeme: Bereitstellung eines passenden Betriebssystems (**Real-Time OS**)

Gliederung

Anwendungsbeispiel: Antiblockiersystem

Technische Funktionsweise

Grundprinzip von ABS [Wik]

- erhöht die **Fahrsicherheit**, indem das Blockieren der Räder bei starkem Bremsen verhindert wird
- **Sensoren** an den Rädern überwachen Rotationsgeschwindigkeit der Räder
- Reduktion des Bremsdrucks, wenn sich eines der Räder wesentlich langsamer dreht als die anderen
- wichtige Komponenten: **ECU** (Electronic Control Unit) und **HCU** (Hydraulic Control Unit)

Anwendungsbeispiel: Antiblockiersystem

Hardware

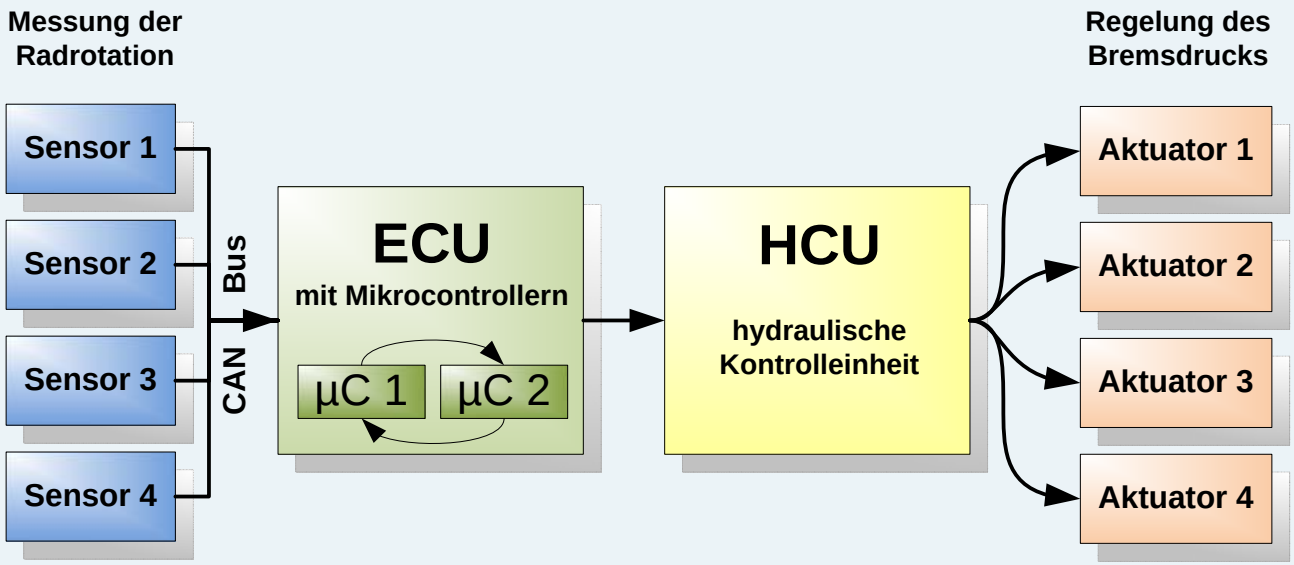

Abbildung: Schematische Darstellung der ABS-Hardware

Anwendungsbeispiel: Antiblockiersystem

Software

Aufgabenbereiche der ABS-Software

- Aufbereitung der einkommenden **Signale** vom Bus
- Auswertung der Daten und Berechnung des Reaktionsverhaltens (eigentlicher **Kernalgorithmus**)
- Ansteuerung der **hydraulischen Kontrolleinheit** (HCU)
- außerdem: **Diagnosefunktionen**

Anwendungsbeispiel: Antiblockiersystem

Besondere Anforderungen

Echtzeitverhalten

Kernroutinen stellen ein *hartes* Echtzeitsystem dar. Ergebnisse müssen vor Eintreten einer Deadline vorliegen, sonst nutzlos. **Performance** überaus wichtig, da sicherheitskritisch.

Diagnosefunktionen haben geringere Priorität (*weiches* Echtzeitsystem).

Verfügbare Systemressourcen

Nur **geringe Systemressourcen** vorhanden (wenig leistungsfähige Mikrocontroller) zur Kostenreduktion und Verringerung des Ausfallrisikos (Abwärme) [Wik].

Anwendungsbeispiel: Antiblockiersystem

Vorschlag für ein komponentenbasiertes Softwaredesign I

Unterteilung in Komponenten

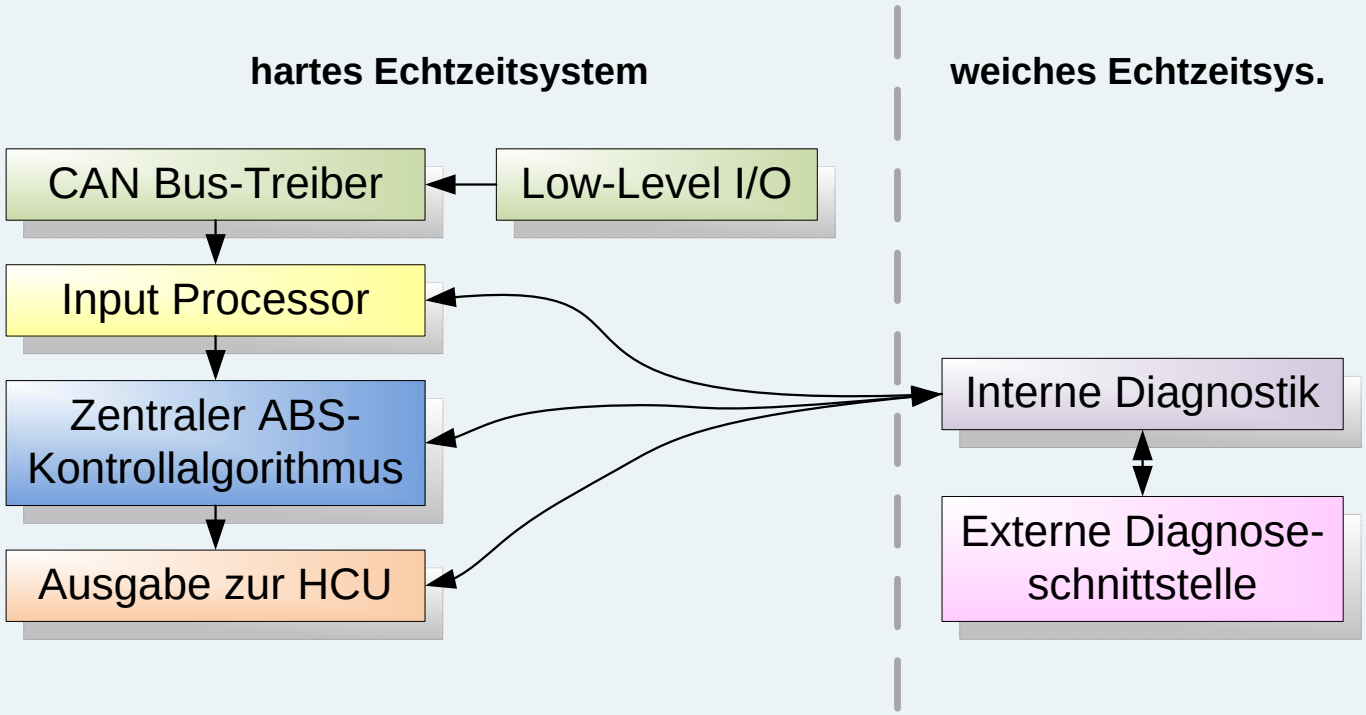

Abbildung: Unterteilung der ABS-Software in Komponenten

Anwendungsbeispiel: Antiblockiersystem

Vorschlag für ein komponentenbasiertes Softwaredesign II

Einrichtung des Komponenten-Repositorys

- Ablegen von **nicht-funktionalen Parametern** (z.B. Latenzzeit des CAN-Bus-Treibers) zusammen mit den Komponenten im Repository
- **Kernroutinen** (Logik) abstrakt halten, um sie z.B. in anderen Fahrzeugen wiederverwenden zu können
- **Fahrzeugspezifische Parameter** in einzelnen Dateien speichern und bei der Kompilierung einbinden

Anwendungsbeispiel: Antiblockiersystem

Vorschlag für ein komponentenbasiertes Softwaredesign III

Überlegungen bzgl. der Laufzeitumgebung

Probleme:

- **Synchronisierung** der beiden Mikrocontroller
- **Prioritätenverteilung** (Kernalgorithmus / Diagnose-Funktionen)

$\rightsquigarrow$ evtl. Nutzung einer *synchronen Programmiersprache* wie **Esterel** [Hal98] angebracht

Gliederung

Schlussfolgerungen

Welche Schlüsse ziehen wir aus unseren Beobachtungen?

- heutzutage **kaum komponentenbasiertes Software-Engineering** (CBSE) im Bereich der eingebetteten Systeme
- CBSE kann **Effizienz** und **Wartbarkeit** steigern und zur **Kostenreduktion** beitragen
- durchaus realisierbar, aber vor allem bzgl. der **Flexibilität** müssen **Abstriche** gemacht werden
- letztendlich entscheidet die **Leistungsfähigkeit der Hardware**
- **Wiederverwendbarkeit** ist aber durchaus zu erreichen.

Fragen/Diskussion

Fragen!?

Danke für die Aufmerksamkeit!

Bibliographie I

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.

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.

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

Bibliographie II


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.


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

Bibliographie III

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