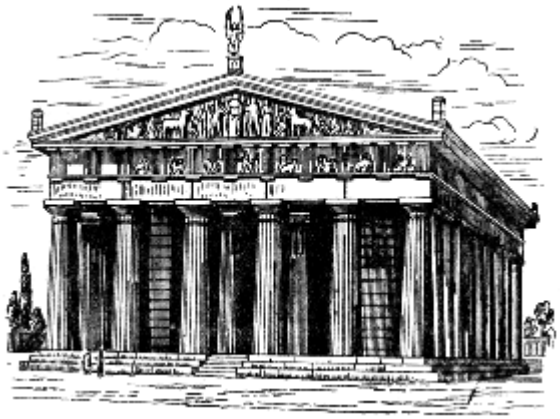
Softwarearchitektur

(Architektur: *αρχή* = Anfang, Ursprung + *tectum* = Haus, Dach)

11. Eingebettete Systeme Teil I



Vorlesung Wintersemester 2010 / 11

Technische Universität München
Institut für Informatik
Lehrstuhl von Prof. Dr. Manfred Broy

Dr. Klaus Bergner, Prof. Dr. Manfred Broy, Dr. Marc Sihling

Inhalt

- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Inhalt

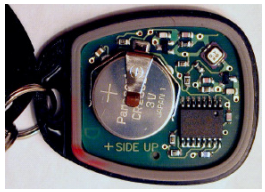
- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Definition: Eingebettetes System

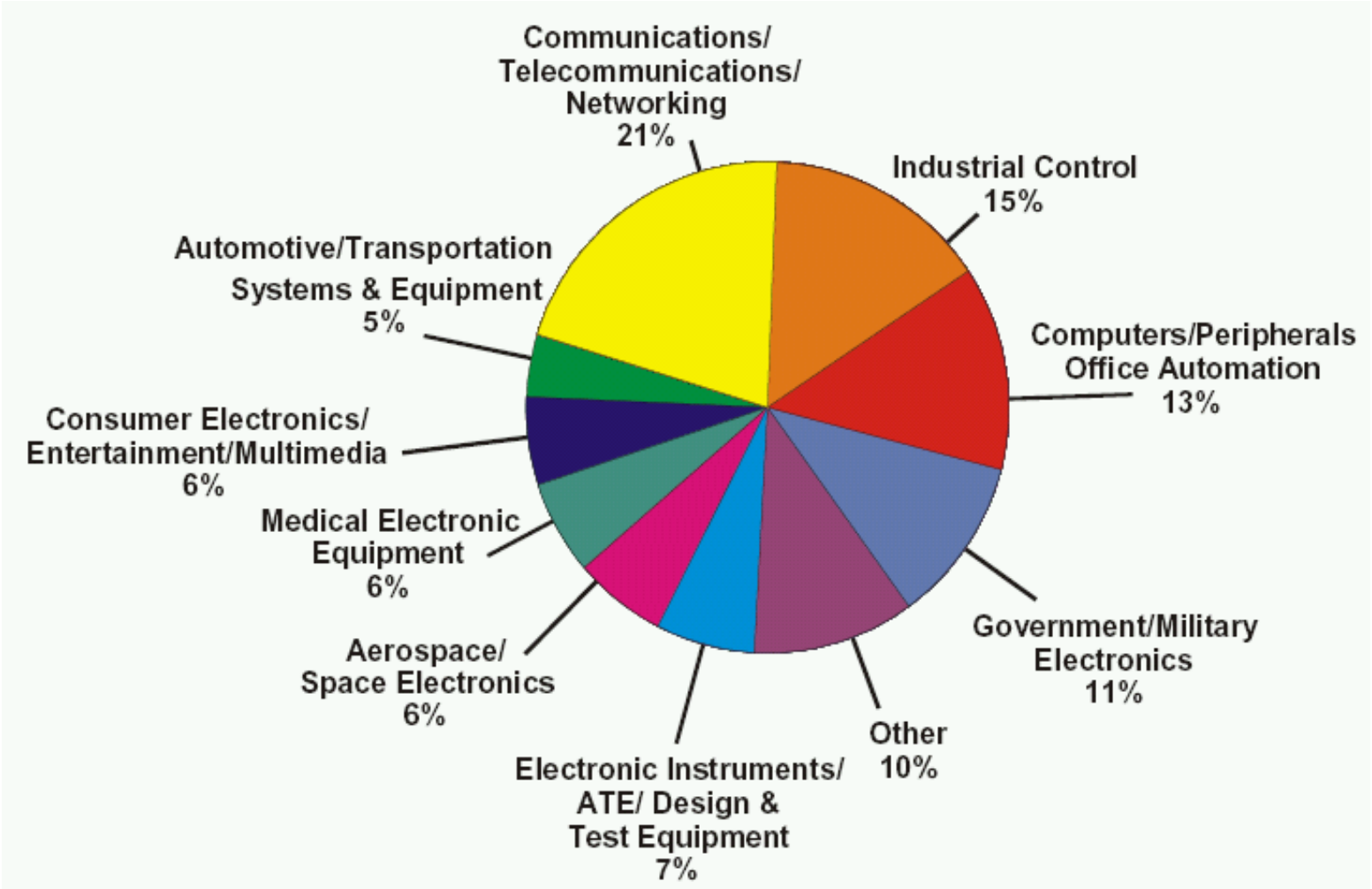
- „In contrast to a general-purpose computer, an embedded system performs a single well-defined task. It is a combination of custom-built hardware and software.“
- „Rechnersystem, das in einen technischen Prozess eingebettet ist und in diesem Rahmen ein oder mehrere Geräte steuert.“
- „Computersystem, das in ein umgebendes technisches System eingebettet und nach außen nicht als Computersystem erkennbar ist.“
- „The general purpose of an embedded hardware/software system is to control or to monitor a physical device by sending control commands to actuators in reaction to input signals from sensors or human users.“

Beispiele

- Werkzeugmaschinensteuerung
  - Robotersteuerung in Produktionsstraße
  - rechnergesteuerte Drehbank
- Fahrzeugsteuerung
  - ABS-Steuerung (Antiblockiersystem)
  - Steuerung der Zentralverriegelung
  - Motorsteuerung
- Steuerung für Haustechnik und Haushaltsgeräte
  - Liftsteuerung, Heizungssteuerung
  - Waschmaschine, Geschirrspüler
  - Fernseher, Videorekorder
- Mobile Geräte
  - Armbanduhr, MP3-Spieler, GPS-Navigator
  - Mobiltelefon, SmartCards
- ... und viele mehr.



Anwendungsdomänen



[laut Umfrage aus *Embedded Systems Programming*, 12/1998]

Typische Randbedingungen

- Geringe Größe, geringes Gewicht
  - Tragbare Geräte, Platzrestriktionen in Fahrzeugen
  - Gewichtsreduktion in der Luftfahrt
- Zeitkritisch
- Niedriger Energieverbrauch
  - Oft kein Stromnetz verfügbar, lange Batterielaufzeit nötig.
  - Ungenügende Kühlung
- Unfreundliche Umgebung
  - Stromschwankungen, Hitze, Vibrationen, Feuchtigkeit, ...
- Zwang zur Kostenminimierung
  - hohe Stückzahlen verlangen minimale Kosten (Beispiel: Auslieferung von Millionen von SmartCards)
- Oft sicherheitskritisch
- Lange Lebenszyklen
  - typischerweise 5 bis 50 Jahre

Basishardware

- Spezielle Prozessoren
  - Prozessoren mit minimalem Energieverbrauch (4-bit, embedded)
  - Micro-Controller mit zusätzlichen Steuerungsleitungen
  - oft Abwesenheit von Features wie MMU - Memory Management Unit, Floating-Point etc.
  - Prozessoren für Spezialanwendungen (z.B. Digitale Signalprozessoren, System-on-a-Chip, Kryptoprozessoren)
- Spezielle Speicher
  - meist keine magnetischen Massenspeicher
  - Software oft im ROM, PROM, oder EEPROM
  - oft minimaler RAM-Speicher
- Spezielle Zusatzhardware
  - speziell entwickelte Platinen, Schaltungen, ASICs
  - Sensoren, Aktuatoren
- Spezielle Kommunikationsbusse
  - deterministische Protokolle für verlässliche Auslieferung - Echtzeitsteuerung
- Spezielle Betriebsysteme
  - Echtzeitsteuerung (Scheduling)
  - Oft kein virtueller Speicher

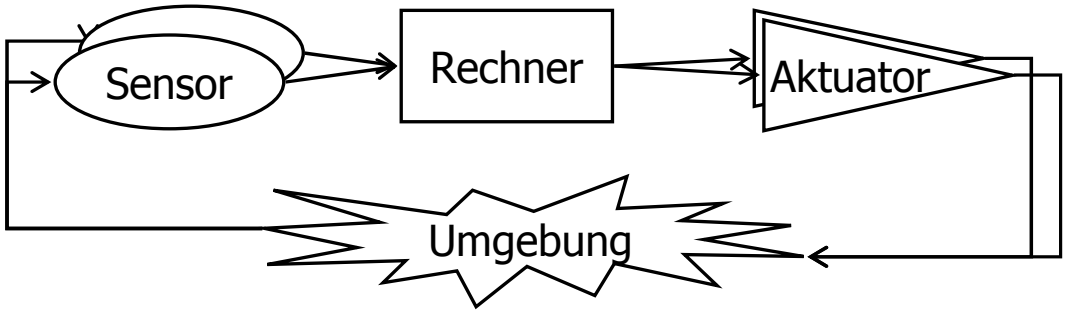
Echtzeitaspekte

- Art der Eingaben
  - Reaktion auf diskrete, asynchrone Eingaben (event-driven)
  - Reaktion auf zeit-synchrone Eingaben (time-driven)
- Reaktionszeit
  - harte Echtzeitanforderungen: jede Eingabe muss rechtzeitig verarbeitet werden (hard realtime constraints), die Zeitschranken sind für die Korrektheit der realisierten Funktion entscheidend
  - weiche Echtzeitanforderungen: die meisten Eingaben müssen rechtzeitig verarbeitet werden (Quality of Service, QoS), leichte Überschreitungen der Zeitschranken sind unerwünscht, aber tolerierbar
- Modelle
  - Kontinuierliche Modellierung (Reglungstechnik)
  - Diskrete Modellierung (discrete event systems)
- Lastprofil
  - statische Last: im Vorhinein bekannte, deterministische Last
  - dynamische Last: variable, nicht deterministische Last

Systeme sind häufig Mischformen.

Geschlossene lokale Systeme

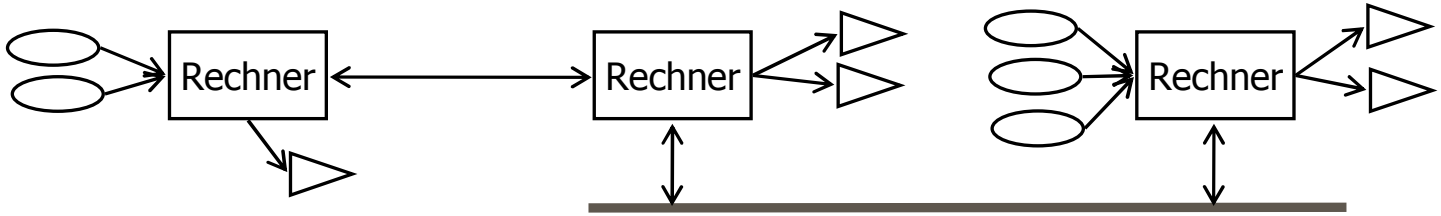
- Ein Steuerrechner kontrolliert (oft mehrere) Sensoren und Aktuatoren.
- Das System hat eine (selten mehrere) fest definierte Aufgabe(n).



- Typische Charakteristika:
  - zur Laufzeit fixe Konfiguration
  - meist fixe Anforderungen
  - hohe Zuverlässigkeit
  - Ereignissteuerung
  - Echtzeitbetrieb
- Typische Beispiele:
  - Haushaltsgeräte
  - Werkzeugmaschinen-steuerung
  - Medizinische Geräte
  - Unterhaltungselek-tronik

Geschlossene verteilte Systeme

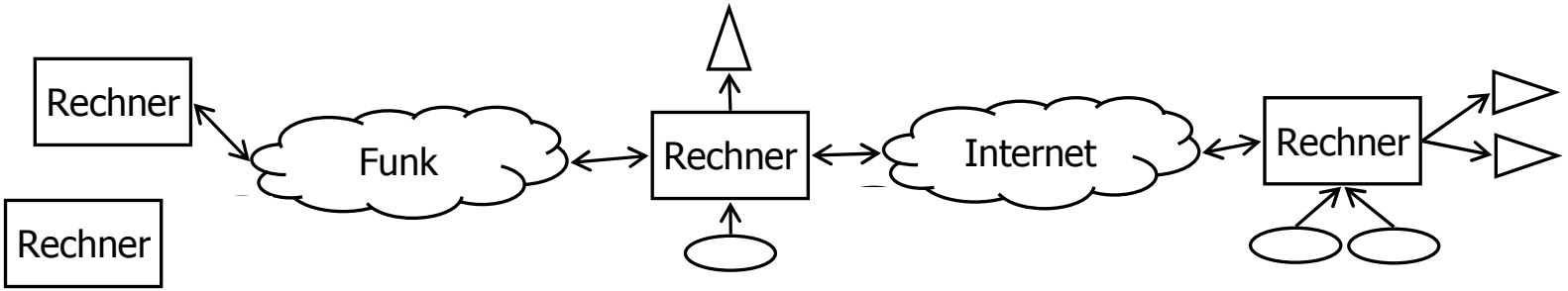
- Mehrere Steuerrechner kommunizieren über dedizierte Verbindungen oder (Feld-)Busse.
- Von Zeit zu Zeit kommen neue Anwendungen hinzu.



- Typische Charakteristika:
  - zur Laufzeit fixe Konfiguration
  - von Zeit zu Zeit zusätzliche Anforderungen
  - „graceful degradation" - fail safe
  - Ereignissteuerung
  - Echtzeitbetrieb
- Typische Beispiele:
  - Fahrzeugsteuerung
  - Produktionsstraße
  - Liftsteuerung

Offene verteilte Systeme

- Wechselnde Geräte kommunizieren über dynamisch aufgebaute Verbindungen (beispielsweise über Internet oder über Funk).
- Neue Anwendungen lassen sich zur Laufzeit installieren.



- Typische Charakteristika:
  - wechselnde Konfigurationen
  - schnell wechselnde Anforderungen
  - sichere Identifikation und sichere Verbindungen
  - Flexibilität und Robustheit
- Typische Beispiele:
  - Webcams
  - Bluetooth-Netze
  - intelligente Hausnetze
  - verteilte SmartCard-Anwendungen

Typische Anforderungen (-abilities)

Hauptanforderungen für die unterschiedlichen Klassen:

geschlossene Systeme

- Reliability: das Anwendungssystem liefert stets die erwarteten, richtigen Ergebnisse
- Availability: Dauer, in der ein System Dienste anbieten kann

offene Systeme

- Interoperability: Möglichkeit, auf Daten und Prozesse anderer Plattformen zuzugreifen
- Maintainability: Notwendige Änderungen können einfach durchgeführt werden
- Extensibility: die Funktionalität der Anwendung ist leicht zu erweitern
- Scalability: Fähigkeit, auch einer wachsenden Anzahl von Benutzern zu dienen

Kombination unterschiedlicher Systeme

- Trend: Geschlossene Systeme werden zunehmend zu Komponenten in offenen verteilten Systemen:
  - Beispiel: Einbindung der Fahrzeugsteuerung in übergreifende Verkehrsleitsysteme
- Die Querbeziehungen zwischen sicherheitskritischen und (eigentlich) unkritischen Komponenten stellen heute ein großes Problem dar.
  - Komfortfunktionen wie Radio sollten vom Motormanagement eigentlich strikt getrennt sein
  - Aber neue Anforderung: Regulierung der Radiolautstärke in Abhängigkeit vom Fahrgeräusch (abhängig z.B. von Sensor-Informationen über Geschwindigkeit, Reifendruck, Motorleistung)
  - Einschalten des Radios im Prototyp-Fahrzeug legte gesamtes Motormanagement lahm ...

Unterschiede zu betrieblichen Infosystemen

- Nicht die Daten des Systems und sein Durchsatz stehen im Vordergrund, sondern Ereignisse, Interaktion, Reaktionszeiten und vorhersagbares Verhalten.
- Komponenten können das korrekte Funktionieren des Systems nicht nur über explizite Schnittstellen beeinträchtigen, sondern auch implizit über die gemeinsame Nutzung knapper Ressourcen (z.B. Bandbreite).
- Parallelität und Nebenläufigkeit (Scheduling) müssen vom Architekten explizit entworfen werden (insbesondere bei Echtzeitsystemen).
- Die Anforderungen an Korrektheit, Ausfallsicherheit und Robustheit sind oft extrem hoch.
- Der Aufbau eines Testsystems und die Durchführung von Tests sind speziell bei verteilten Systemen mit Spezialhardware schwierig und langwierig (oft Cross-Development und Simulatoren erforderlich).
- Ressourcenknappheit erfordert oft Programmierung auf niedriger Ebene und Hand-Optimierung.
- Architekten und Entwickler müssen (zumindest heute noch in den meisten Fällen) ein umfassendes Verständnis der zu Grunde liegenden Hard- und Software und ihrer Eigenschaften haben.

Umfassende Architekturmodellierung

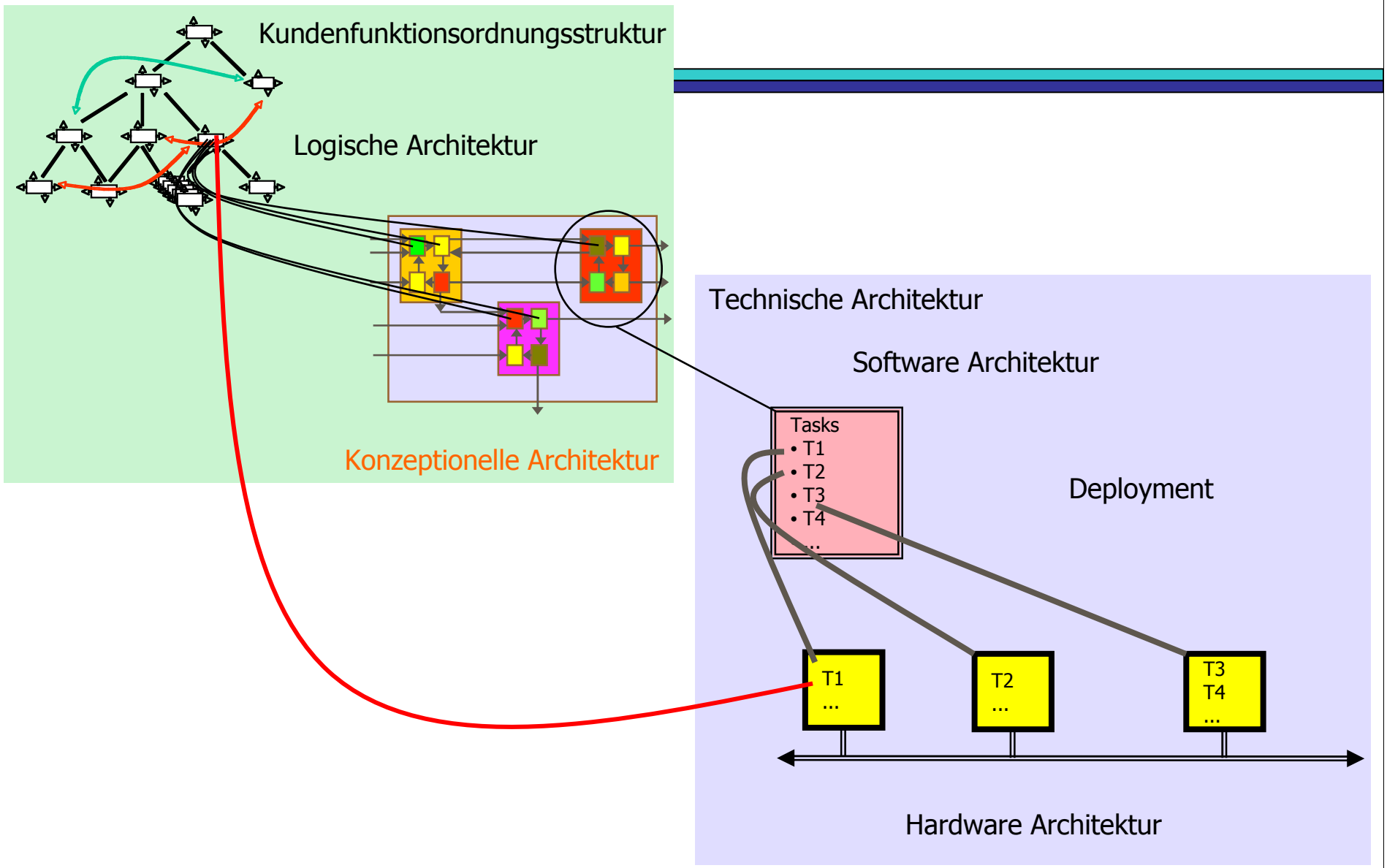
- Kunden/Nutzerfunktionsordnungsstruktur: Nutzungsebene - Kundenfunktionen (KF)
  - Multi-funktionale Systeme (K.Funktionshierarchien - feature hierarchies)
  - Abhängigkeiten/Beziehungen zwischen KFs: Feature interaction
- Logische Systemarchitektur (Funktionsnetz)
  - Hierarchische Zerlegung des Systems in logische Komponenten

Konzeptionelle Architektur

- Software-Architektur
  - Design Time Software Architektur
    - Applikationssoftware
    - Software Plattform (OSEK, Treiber für Bussysteme)
  - Laufzeit Software-Architektur
    - Tasks
    - Scheduling
- Hardware Architektur
  - Steuergeräte
  - Kommunikationseinrichtungen
  - Sensoren und Aktuatoren
- Deployment

Technische Architektur

Das umfassende Modell



Inhalt

- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Fachliche Architektur - Funktionsarchitektur

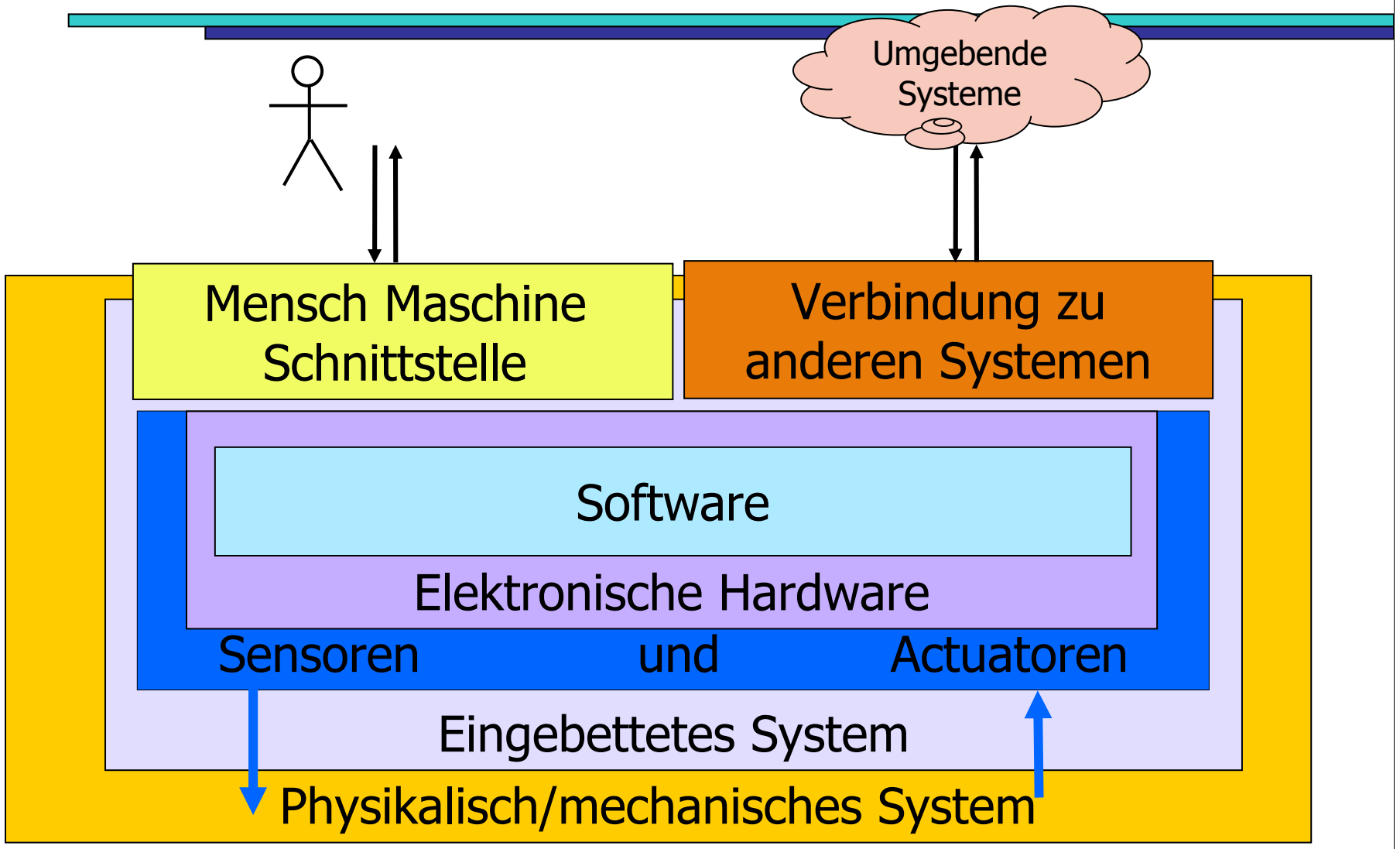
- Eine Funktion eines eingebetteten Systems
  - dient einem Zweck (Use Case)
  - manifestiert sich in der Interaktion zwischen
    - Eingaben (Sensorwerte, Nutzereingaben)
    - Ausgaben (Steuerwerte an Aktuatoren, Ausgaben an Nutzer)

    an den Schnittstellen des Systems
  - kann als Funktion/Relation (Abbildung) zwischen Strömen von Eingabewerten und Strömen von Ausgabewerten modelliert werden
- In den Strömen ist auch Zeitinformation (zu welchem Zeitpunkt wird welche Nachricht übertragen) erforderlich um die Echtzeiteigenschaften einer Funktion darzustellen

Funktionsarchitektur

- Ein eingebettetes System realisiert Funktionen an seinen Systemgrenzen
- Einfache eingebettete Systeme realisieren lediglich eine Funktion
  - Beispiel: Heizungssteuerung
- Multifunktionale Systeme realisieren eine Familie von Funktionen
  - Beispiel: Moderne Kraftfahrzeuge
  - Zwischen den Funktionen bestehen Abhängigkeiten

Zwiebelschalenartige Struktur eingebetteter Systeme

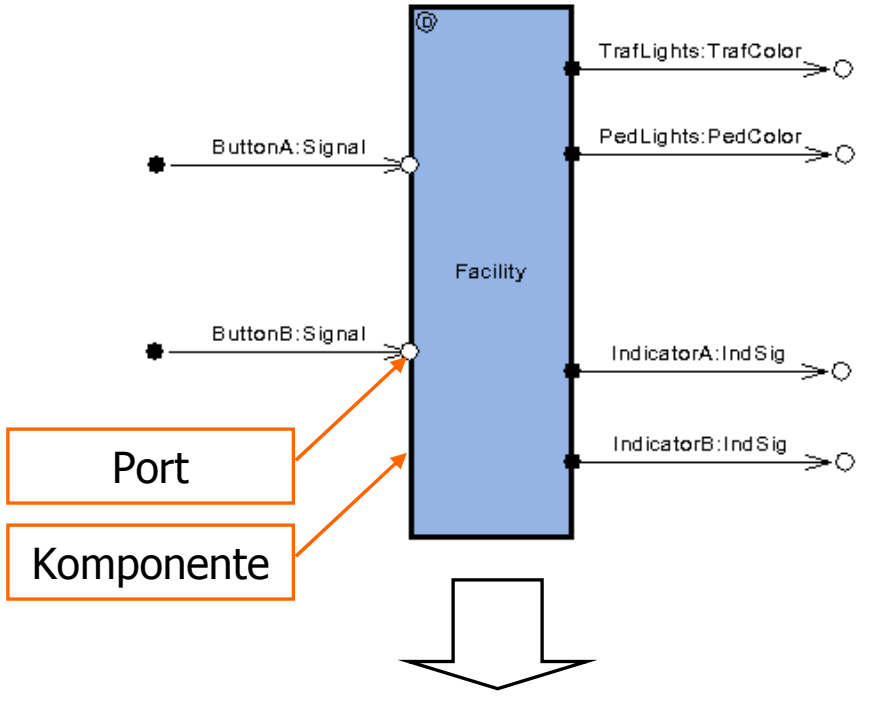


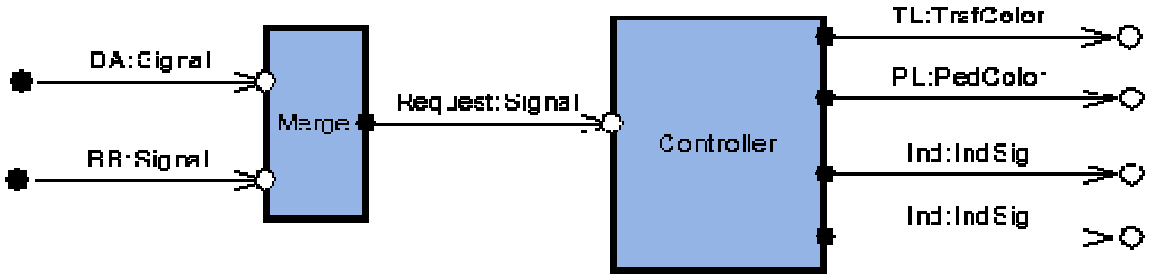
Fachliche Architektur - Funktionsarchitektur

- In geschlossenen Systemen steht nicht die Modellierung von Klassen und Operationen im Vordergrund, sondern
  - Konfigurationen der Komponenten (Datenflussarchitektur)
  - Ereignisse und Nachrichten
  - Zustände der Komponenten
  - Kommunikation und Interaktionsprotokolle
  - Wechselwirkung mit der physischen Umgebung
- Weitere wichtige Aspekte betreffen die nichtfunktionalen Anforderungen an das System, die von der technischen Architektur erfüllt werden müssen:
  - Realtime constraints (harte Echtzeit)
  - Quality of Service (QoS, weiche Echtzeit)
  - Zuverlässigkeit und Sicherheit

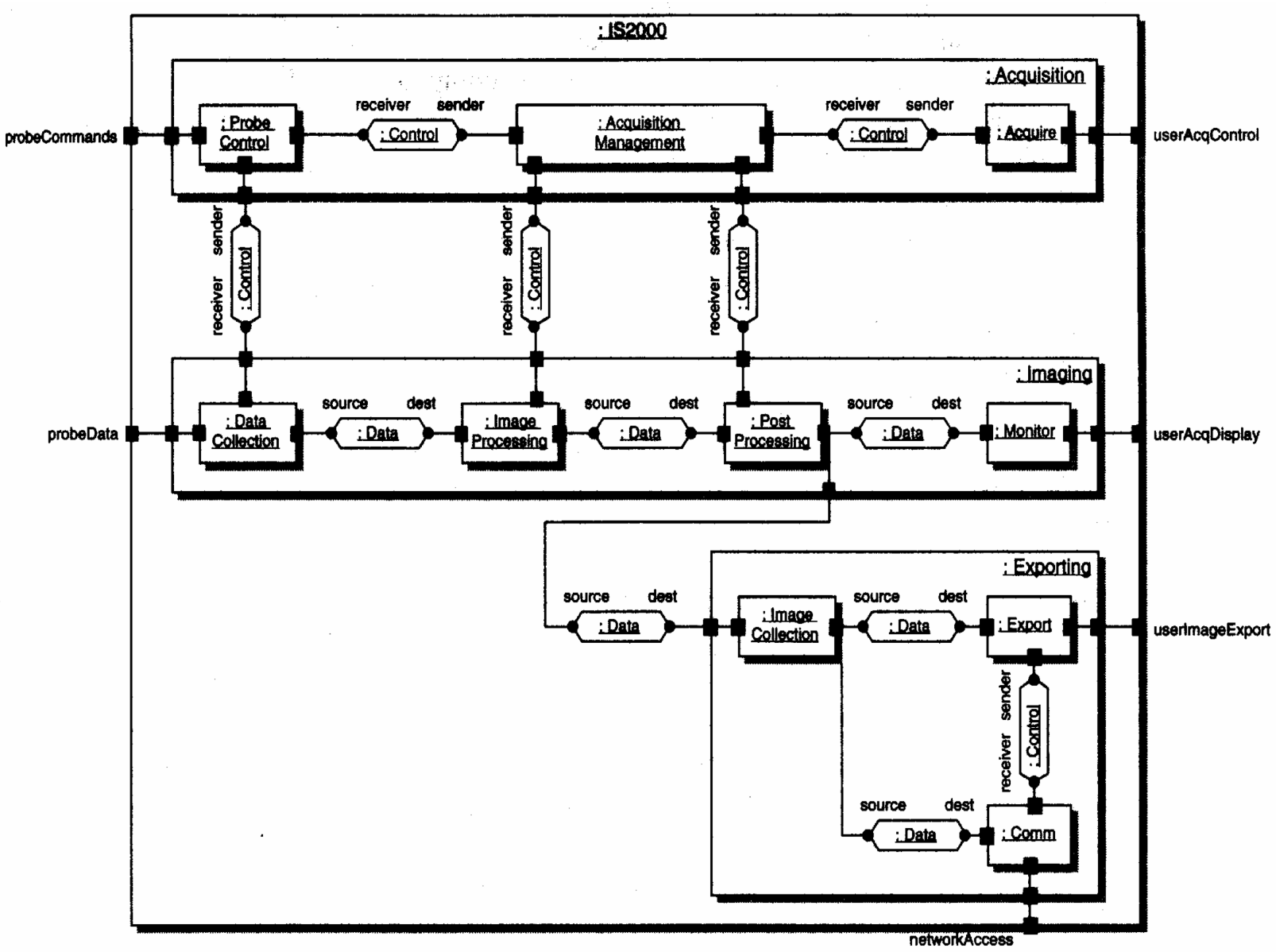
Spezifikation fixer Konfigurationen von Instanzen

- Die Spezifikation erfolgt meist mit Hilfe von speziellen Instanzendiagrammen.
- Dabei werden typischerweise Komponenten und deren Schnittstellen unterschieden (sogenannte Ports).
- Komponenten können in den meisten Ansätzen verfeinert werden.





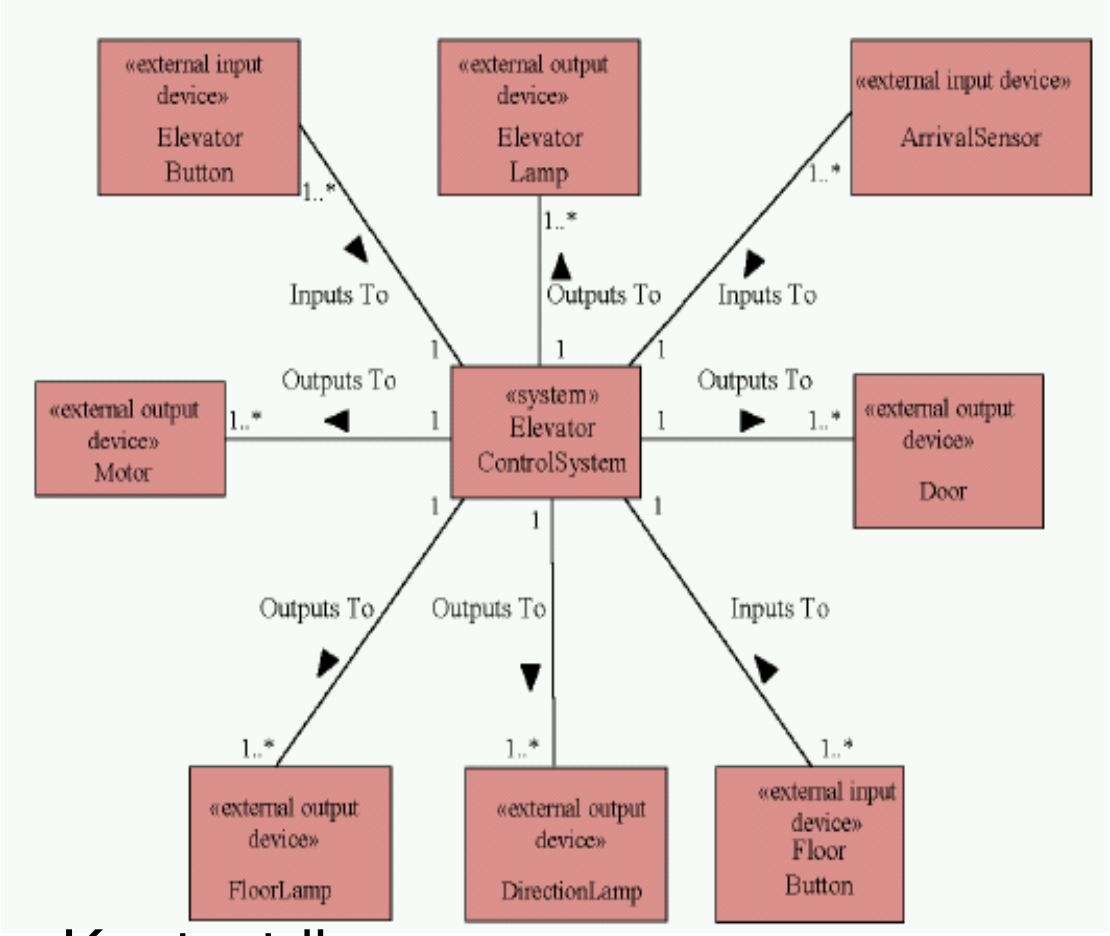
Beispiel: Ampelsteuerung aus [AF03]



Beispiel: Bilderfassungssystem aus [HNS99]

Spezifikation von Konfigurationen von Instanzen

- Falls mehrere gleichartige Objekte in einem System vorkommen, können Klassendiagramme verwendet werden.
- Im Gegensatz zu betrieblichen Informationssystemen lassen sich die Konfigurationen meist direkt erschließen (statt einer Instanz sind mehrere auf die gleiche Art und Weise verbunden).

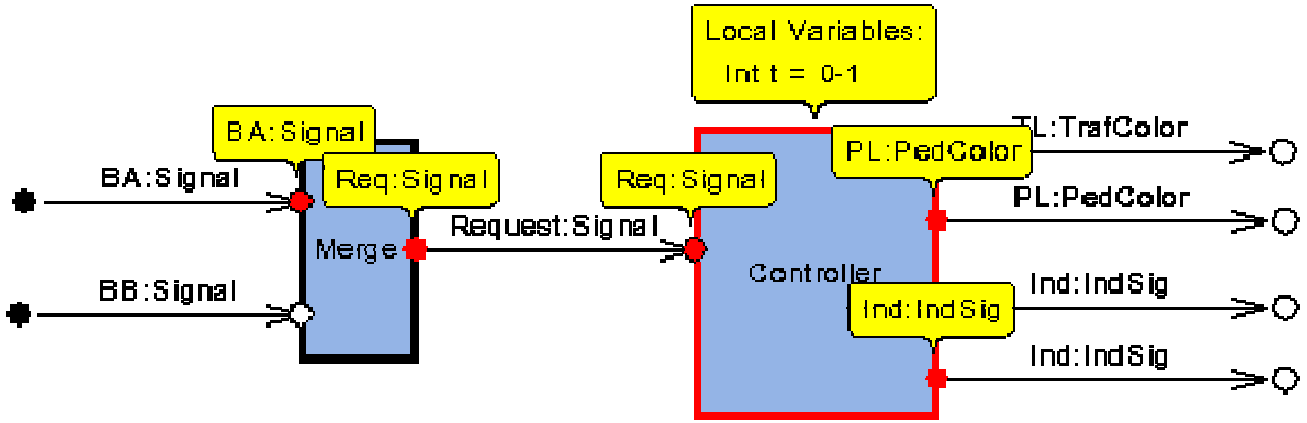


Kontextdiagramm

Beispiel: Liftsteuerung aus [DS01]

Schnittstellen von Instanzen

- An den Schnittstellen wird jeweils angegeben, welche Nachrichten ein- oder ausgehen können.
- Im Gegensatz zu betrieblichen Informationssystemen werden überwiegend keine Aufrufe mit Parametern und Ergebnis angegeben, sondern nur einzelne Nachrichten.

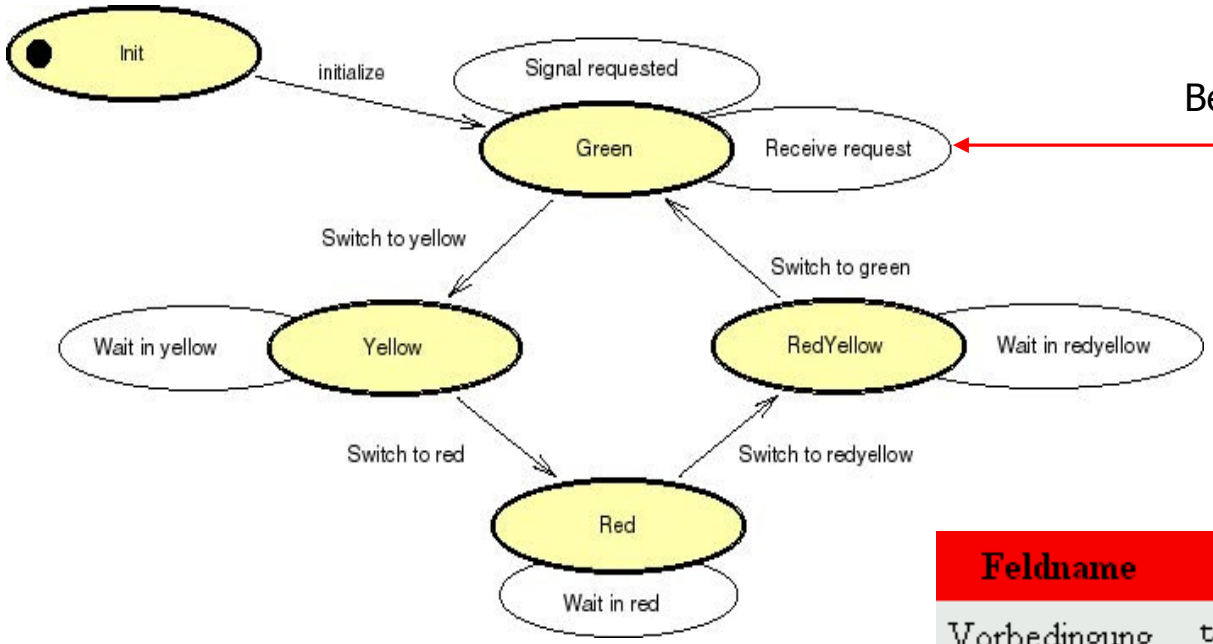


Beispiel: Ampelsteuerung aus [AF03]

Ereignisse und Nachrichten

- Komponenten in eingebetteten Systemen sind ereignisgetrieben.
- Sie reagieren auf Ereignisse von Sensoren oder Uhren, indem sie ihren Zustand ändern und Nachrichten an andere Komponenten oder Aktuatoren verschicken („active objects“).
- Die Zustandsübergänge sowie das Verschicken von Nachrichten bei Zustandsübergängen von Komponenten werden typischerweise mit Hilfe von Zustandsdiagrammen beschrieben.
- Mögliche Informationen an den Transitionen zwischen Zuständen:
  - Vorbedingung: Macht Aussage über Zustand der Komponente. Muss erfüllt sein, damit Transition schalten kann.
  - Eingabemuster: Nachrichten, die an den Schnittstellen anliegen.
  - Ausgabemuster: Nachrichten, die als Reaktion verschickt werden.
  - Nachbedingung: Macht Aussage über Zustand der Komponente nach dem Schalten der Transition.

Beispiel für Zustandsübergangsdiagramm



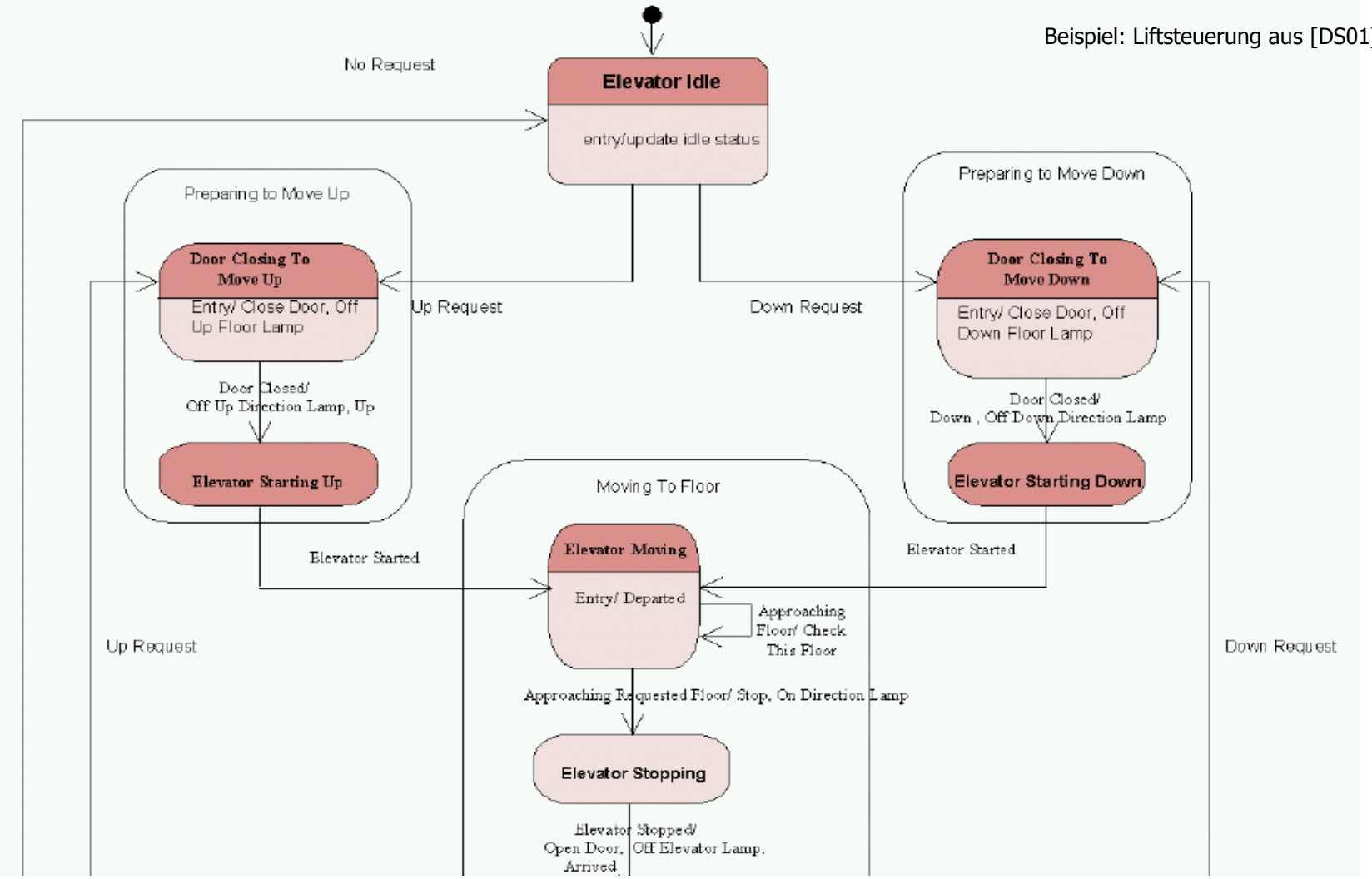
Beschriftung an Transition

Feldname	Wert
Vorbedingung	`t == (0 - 1)`
Eingabemuster	`Req?Present`
Ausgabemuster	`TL!Green; PL!Stop; Ind!On`
Nachbedingung	`t = TGreen`
Beschriftung	`Receive request`
Kommentar	

Beispiel: Ampelsteuerung aus [AF03]

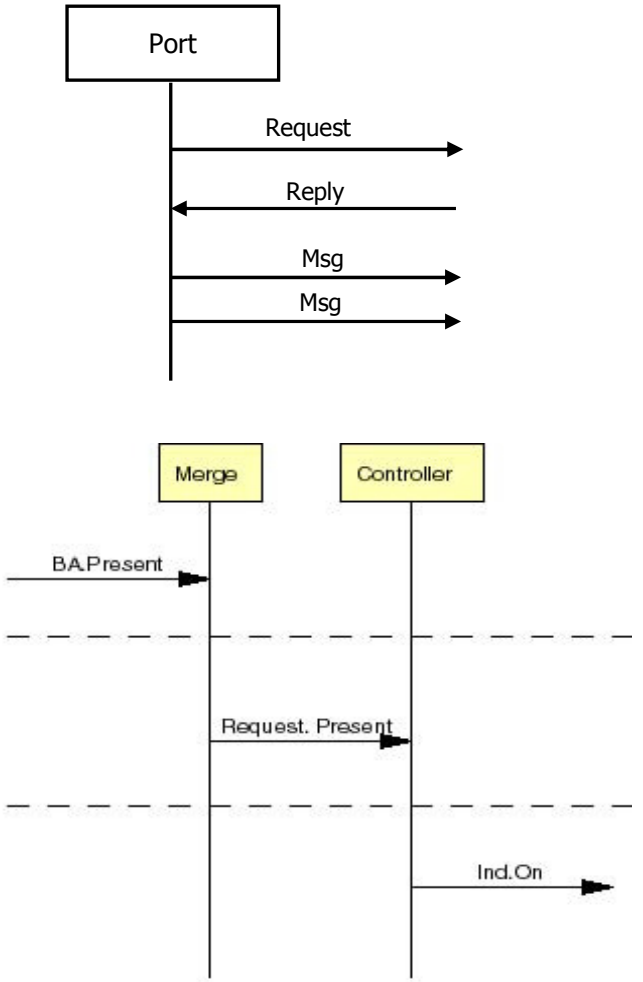
Beispiel für hierarchisches Zustandsdiagramm

Beispiel: Liftsteuerung aus [DS01]



Interaktion und Kommunikation

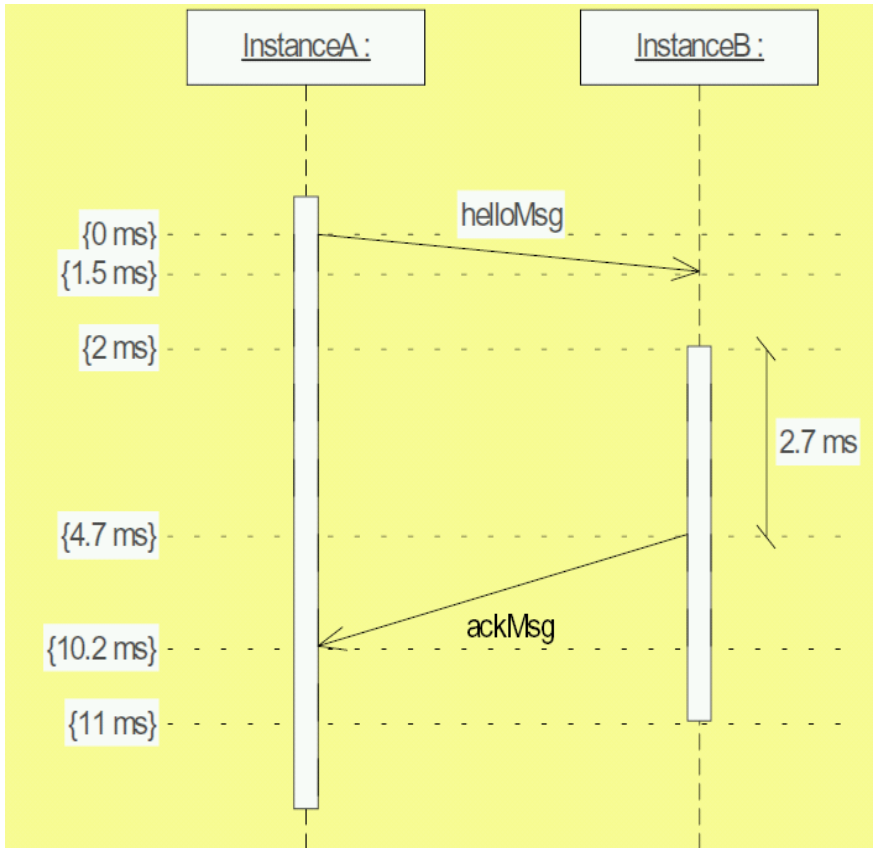
- Wie in betrieblichen Informationssystemen können Protokolle und Interaktionsszenarien durch Sequenzdiagramme dargestellt werden.
- Oft werden diese Diagramme (wie auch Zustandsdiagramme) mit zusätzlichen Annotationen angereichert, um beispielsweise zeitliche Bedingungen oder andere nichtfunktionale Anforderungen zu spezifizieren.

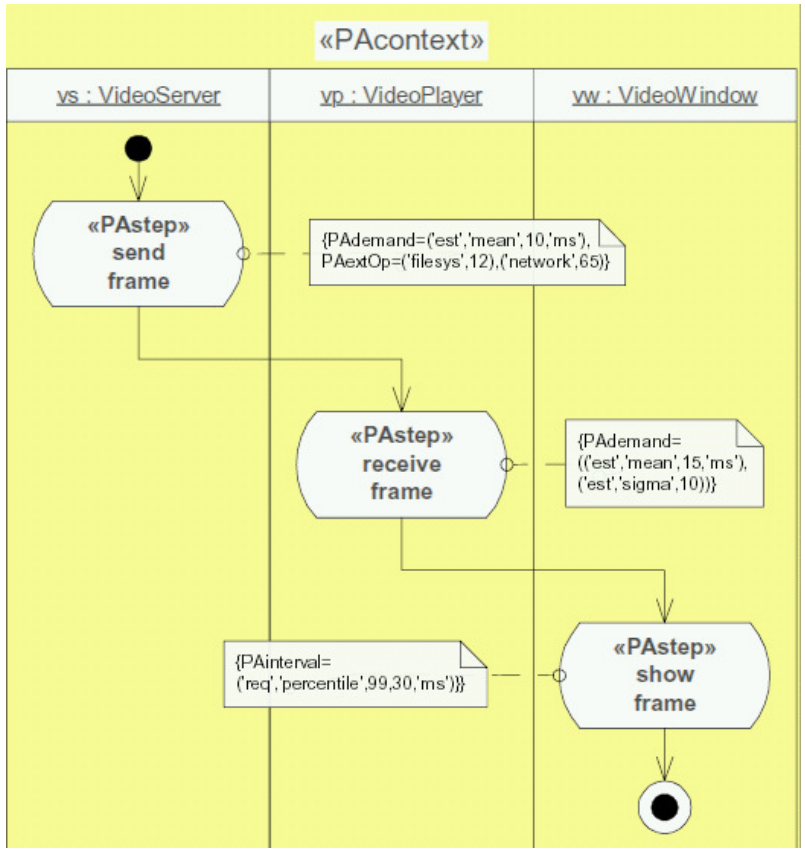


Beispiel: Ampelsteuerung aus [AF03]

Spezifikation von nichtfunktionalen Anforderungen

- Beispiele für Annotationen zur Spezifikation von
  - zeitlichem Verhalten
  - Quality-of-Service (QoS)





Beispiele aus [SW01]

Inhalt

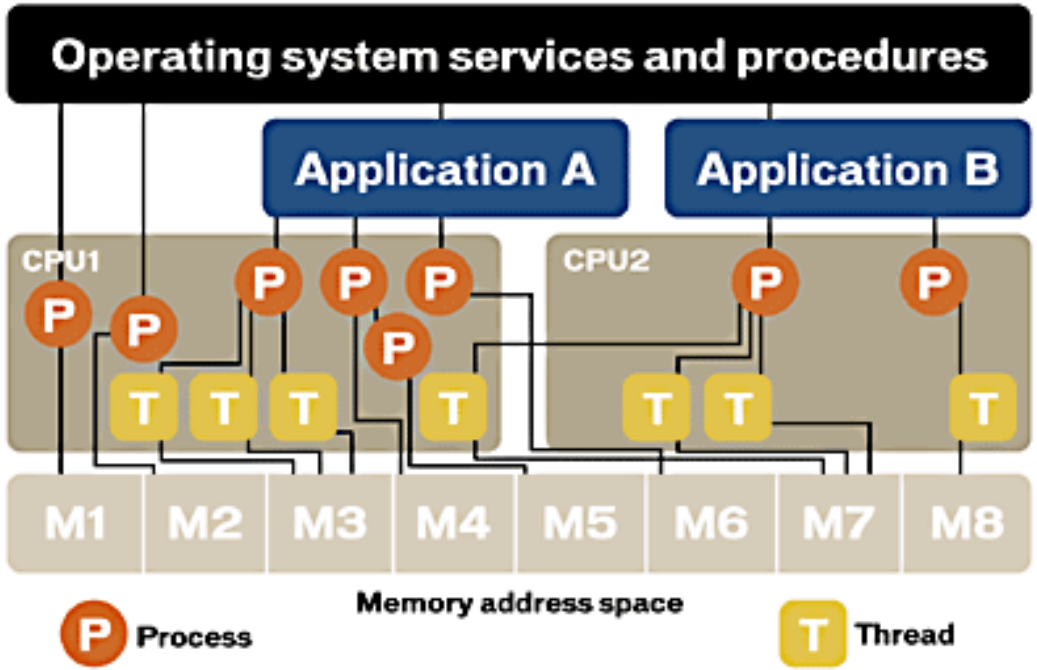
- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Technische Architektur

- Zur Umsetzung der Konzepte der fachlichen Architektur gibt es eine Reihe von Basiskonzepten und den dazu gehörigen Realisierungen, insbesondere für
  - Task-Verwaltung und Scheduling
  - Reaktion auf Ereignisse und Synchronisation
  - Kommunikation und Interaktion
- Die Implementierungen in Form von technischen Infrastrukturkomponenten sind jedoch meist proprietär und unterscheiden sich in vielen Details. Insgesamt
  - haben sich zwar einige weitgehend verwendete Basiskonzepte herauskristallisiert.
  - ist die Standardisierung lange nicht so weit fortgeschritten wie bei den betrieblichen Informationssystemen.

Task-Verwaltung: Prozesse und Threads

- Anwendungen können aus mehreren Prozessen bestehen, die wiederum jeweils aus mehreren Threads bestehen können.
- Sowohl Prozesse als auch Threads laufen quasiparallel ab (oder echt parallel, wenn genügend Prozessoren vorhanden sind). Ein Scheduler bestimmt, wann welcher Task abgearbeitet wird.



Charakteristika von Prozessen und Threads

Prozesse

- eigener Adressraum, damit Speicherschutz
- verwaltet vom Betriebssystem
- typischerweise schwergewichtiger als Threads
- Inter-Prozess-Kommunikation über Mechanismen wie Shared Memory, Queues, Sockets etc.
- Programmierung über Systembibliotheken
- Nachteile: oft proprietäre Schnittstellen, schwergewichtig

Threads

- kein eigener Adressraum, damit kein Speicherschutz
- verwaltet vom Betriebssystem oder Laufzeitsystem der Sprache
- typischerweise sehr leichtgewichtig (schnellere Taskwechsel)
- Inter-Thread-Kommunikation über gemeinsam genutzten Speicher bzw. gemeinsam genutzte Objekte
- Programmierung über Systembibliotheken oder Sprachmittel
- Nachteile: fehlender Speicherschutz erzwingt sorgfältige Synchronisation, überflüssige Synchronisationen beeinträchtigen Performance

Echtzeitsprachen am Beispiel von RT-Java (I)

Über die standardmäßigen Konzepte von Java hinaus bietet RT-Java [RJ01] aufwärtskompatible Erweiterungen in Form zusätzlicher Klassen und Frameworks. Sie betreffen folgende fünf Bereiche:

- Thread-Verwaltung und Scheduling
  - neue Klasse RealtimeThread mit Framework zum Einhängen von beliebigen Scheduling-Algorithmen durch Anbieter
  - präemptiver, prioritätsgesteuerter Standard-Scheduler
- Speicherverwaltung
  - Verwaltung unterschiedlicher Arten von Speicher (z.B. für Flash-RAM)
  - flexiblere Steuerung des Garbage-Collectors
  - expliziter Zugriff auf Speicheradressen
- Echtzeit-Synchronisationsmechanismen
  - beispielsweise Auflösung von Priority Inversion (muss ein Thread einer höheren Priorität auf einen Thread niedrigerer Priorität warten, so kann er ihn „anschieben", indem er ihm temporär seine Priorität gibt)

Echtzeitsprachen am Beispiel von RT-Java (II)

- Behandlung exakter Zeit
  - neue Klassen wie z.B. HighResolutionTime und RelativeTime
  - neue Klassen für Uhren wie z.B. OneShotTimer und PeriodicTimer
- Behandlung asynchroner Nachrichten und Ereignisse
  - neue Klassen AsyncEvent und AsyncEventHandler erlauben die schnelle Reaktion auf Ereignisse, die von außen kommen
  - der betreffende Code im AsyncEventHandler unterliegt dem Scheduler
  - neue Möglichkeiten, Threads im Notfall asynchron zu verlassen bzw. zu terminieren
- Ähnlicher Ansatz: Viele Embedded-C-Dialekte, die Echtzeitfeatures mit Hilfe von Bibliotheken und einem angepassten Laufzeitsystem realisieren.
- Echtzeitsprachen wie PEARL, CHILL oder ADA95 bieten vergleichbare Features mit einer speziellen Syntax an.

Echtzeitbetriebssysteme

Echtzeitbetriebssysteme als Infrastruktur bieten im Vergleich zu „normalen“ Betriebssystemen

- Effizientes Task-Management mit schnellem Taskwechsel (oft nur auf Basis von Threads, wenn die Hardware keine echten Prozesse mit eigenen Adressräumen unterstützt)
- Erweiterte Möglichkeiten für das Scheduling (streng präemptiv mit Prioritäten) und die Unterbrechungsbehandlung
- Mechanismen zur Inter-Task-Kommunikation (Nachrichten, Ereignisse, Semaphore, Warteschlangen, Interrupts etc.)
- Uhren für die zeitgesteuerte oder periodische Aktivierung von Tasks
- Effiziente Kommunikation mit externen Geräten und Feldbussen
- Konfiguration eines „maßgeschneiderten“ Betriebssystems, bei dem alle unnötigen Komponenten weggelassen werden

Beispiele: FLEXOS, LynxOS, QNX, Windows-CE und viele andere

Kommunikation in eingebetteten Systemen

- Die unterschiedlichen Hardware- und Software-Plattformen bieten eine Vielzahl unterschiedlicher Kommunikationsmechanismen an, die speziell auf die Anforderungen von eingebetteten Systemen eingehen.
- Oft gibt es nur low-level-Software-Schnittstellen, die sich zudem noch von Hersteller zu Hersteller unterscheiden.
- Eine Spezifikation bzw. Standardisierung erfolgt vielfach nur auf der Ebene der Hardware-Protokolle (z.B. serielle Schnittstelle, IEEE1394, Ansteuerung von Sensoren über Memory-Ports).
- Neuere Ansätze wie Realtime-CORBA versuchen, diese Vielfalt hinter Standardschnittstellen zu kapseln und einheitliche Basisdienste zur Verfügung zu stellen.

Kommunikation: Beispiele für Basismechanismen

Beispiele für spezielle Kommunikationsmechanismen

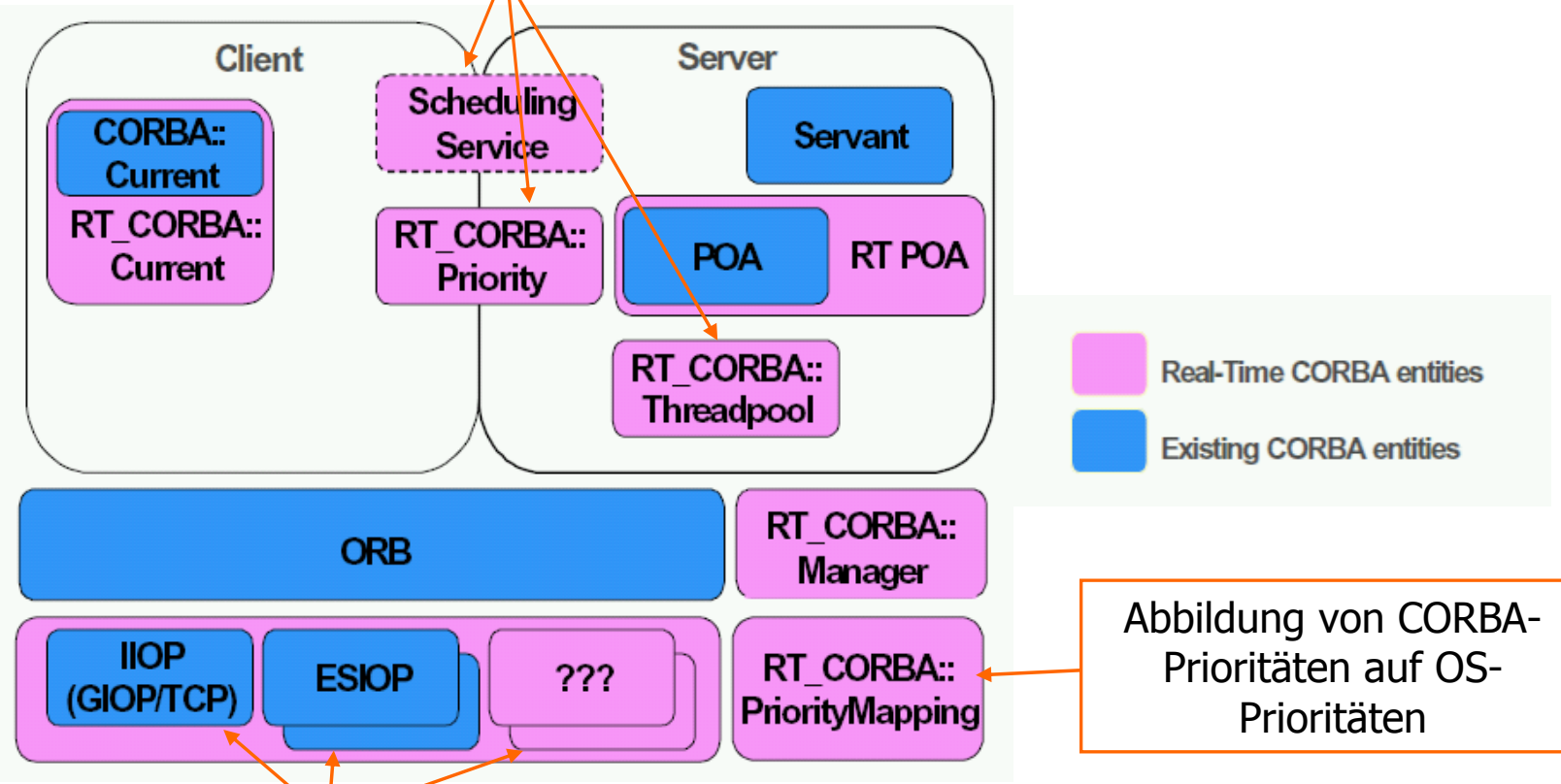
- lokale Prozesskommunikation über Shared Memory
  - Vermeidung von Kopier- und Transfervorgängen durch Nutzung von gemeinsamem Speicher durch mehrere Prozesse
  - muss von Hardware (und Betriebssystem) unterstützt werden
  - Programmierung erfolgt im Allgemeinen auf der Ebene von Betriebssystemaufrufen und Speicherblöcken – Objekte werden typischerweise nicht unterstützt
- entfernte Kommunikation über Feldbusse (z.B. CAN, MAP, FIP etc.)
  - Feldbusse stellen typischerweise eine low-level-Programmier-schnittstelle zur Verfügung, über die Datenpakete effizient und prioritätsgesteuert verschickt werden können
  - Sie bieten eine zuverlässige Übertragung mit deterministischem Zeitverhalten, geeignet für verteilte Echtzeitsysteme

Realtime-CORBA

- Realtime-CORBA ist eine echte Erweiterung von CORBA und setzt auf dessen Schnittstellen auf.
- Ziel ist das Erreichen von „End-to-End-Predictability" – also eines vorhersagbaren, deterministischen Zeitverhaltens.
- Dazu muss der ORB die verfügbaren Ressourcen (insbesondere auch die Threads) adäquat verwalten können (inklusive Scheduling-Policies).
- Um die Eigenheiten spezieller Kommunikationsmechanismen zu kapseln, können unterschiedliche Protokolle unter den ORB gehängt werden, ohne dass die Anwendungen geändert werden müssen.
- Der ORB kann mehrere Protokolle gleichzeitig nutzen, um Nachrichten jeweils gemäß ihrer Priorität über den günstigsten Kommunikationsmechanismus zu verschicken.

Realtime-CORBA: Architektur

detaillierte, prioritätsgesteuerte Kontrolle von Threads und Kommunikation



unterschiedliche Kommunikationsmechanismen gleichzeitig verwendbar

Inhalt

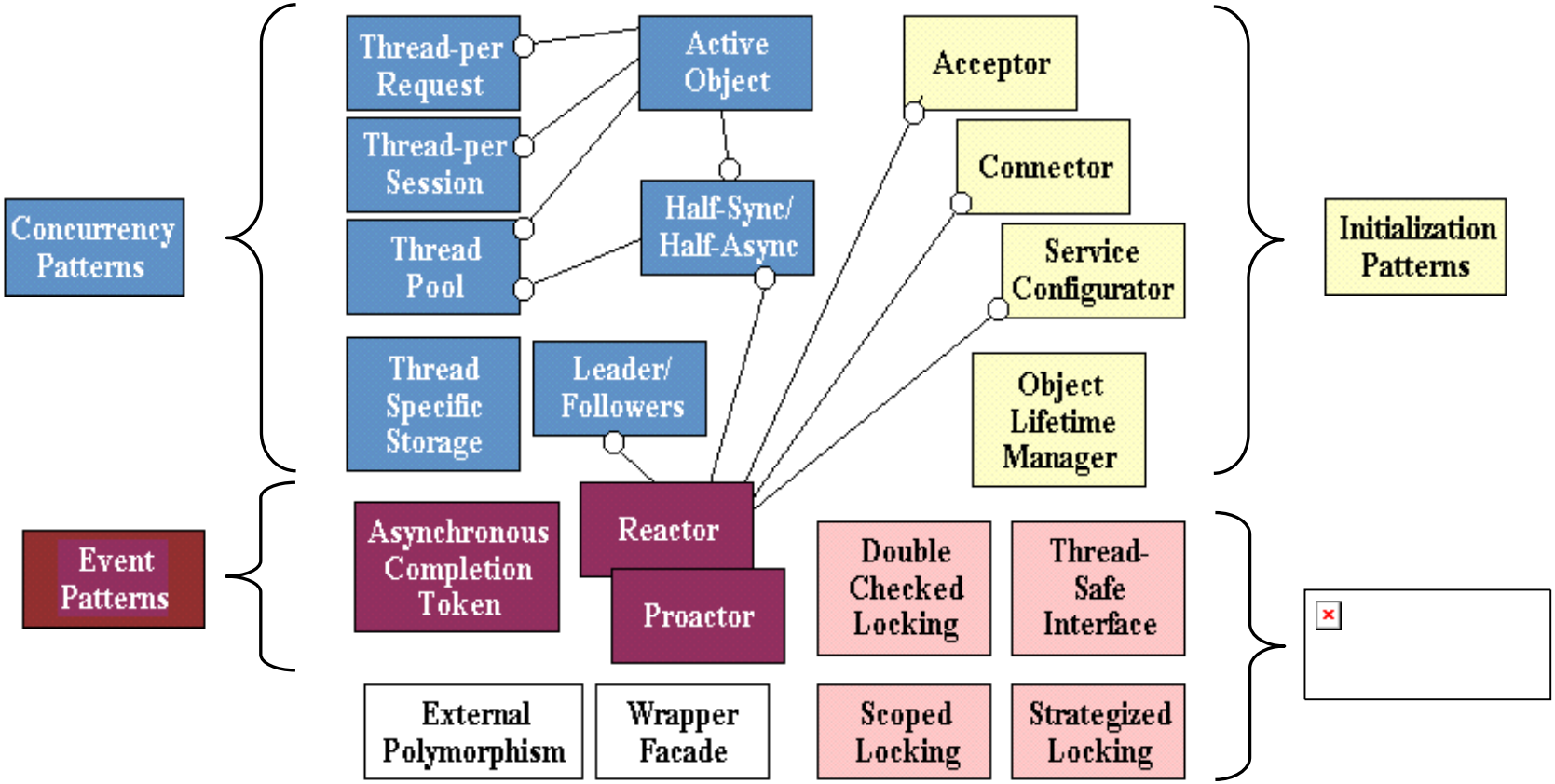
- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Architektur- und Entwurfsmuster

- Für die Entwicklung von eingebetteten Systemen gibt es erprobte Vorgehensweisen, die jedoch meist nur implizit in existierenden Systemen repräsentiert sind.
- Viele grundlegende Architekturmuster wie etwa Layers, Broker lassen sich grundsätzlich auch für eingebettete Systeme nutzen.
- Die meisten speziellen Muster sind auf relativ niedriger Ebene angesiedelt und versuchen, die fehlerträchtige parallele Programmierung zu erleichtern.
- Daneben gibt es Muster für spezielle Vorgehensweisen, beispielsweise beim Handling knapper Ressourcen und für die Erfüllung erhöhter Sicherheitsanforderungen.
- In einigen Anwendungsbereichen oder Unternehmen haben sich „lokale Standardarchitekturen" herausgebildet. Diese sind allerdings meist auf eine ganz bestimmte technische Basis und bestimmte Produkte zugeschnitten und sehr spezifisch.

Musterkataloge für parallele Programmierung

- Beispiel: Musterkatalog von Douglas Schmidt et al [D+02]



Beispielmuster aus [D+02]

- Thread-Safe Interface
  - Minimiert den Sperr-Aufwand und stellt sicher, dass es beim rekursiven Aufruf keinen Selbst-Ausschluss gibt.
- Double-Checked Locking
  - Vermindert den Synchronisations-Aufwand, wenn kritische Abschnitte nur einmal während der Programmausführung Sperren setzen müssen.
- Thread-Specific Storage
  - Stellt Programmen einen thread-lokalen Speicher auf transparente Art und Weise zur Verfügung. Erleichtert damit die Portierung von single-threaded Code.
- Active Object
  - Beschreibt die effiziente Realisierung eines Objekts mit einem eigenen, ihm zugeordneten Thread, das über einen Eingabepuffer mit Aufträgen versorgt wird und diese asynchron abarbeitet.

Musterkataloge für eingebettete Systeme

- Beispiel: Musterkatalog von Bruce Powell [BP03]

Homogeneous Redundancy	System mit mehreren identischen Komponenten, die bei zufälligen Fehlern füreinander einspringen.
Diverse Redundancy	System mit mehreren verschiedenartigen Komponenten als Sicherung gegen Fehler bei der Entwicklung.
Sanity Check	Diverse Redundancy: Eine Komponente realisiert Funktionalität, eine zweite prüft die Plausibilität.
Monitor-Actuator	Diverse Redundancy: Eine Komponente steuert einen Aktuator, eine andere überwacht die Performance.
Watchdog	Eine Wachhund-Komponente muss regelmäßig angestoßen („gefüttert“) werden. Andernfalls beißt sie zu (stößt eine Ausnahmebehandlung an).
Safety Executive	Eine zentrale, hochsichere Komponente koordiniert die Identifikation von Fehlern sowie das Wiederaufsetzen.

Inhalt

- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Umsetzung der Fachlichkeit

- Methoden wie Focus, ROOM oder ASCET/UML mit den entsprechenden Beschreibungstechniken wie RT-UML oder in Zukunft UML 2.0 machen Vorgaben für das Vorgehen bei der Abbildung der Fachlichkeit auf die Technik (inklusive Hardware-Software-Mapping).
- Erste Tools für derartige Ansätze existieren (z.B. Rose-RT von Rational, AutoFocus [AF03], Rhapsody). Für spezielle Ziel-Architekturen gibt es auch schon Code-Generatoren.
- In der Praxis ist das Vorgehen meist wesentlich stärker technikgetrieben. Gründe dafür sind unter anderem:
  - Komplexität der technischen Infrastruktur macht einfache Abbildung unmöglich
  - Nötige Optimierungen durch Nutzung plattformspezifischer Mittel
  - Unzureichende Fähigkeiten der Modellierungswerkzeuge bei der Entwicklung von verteilten, heterogenen Systemen

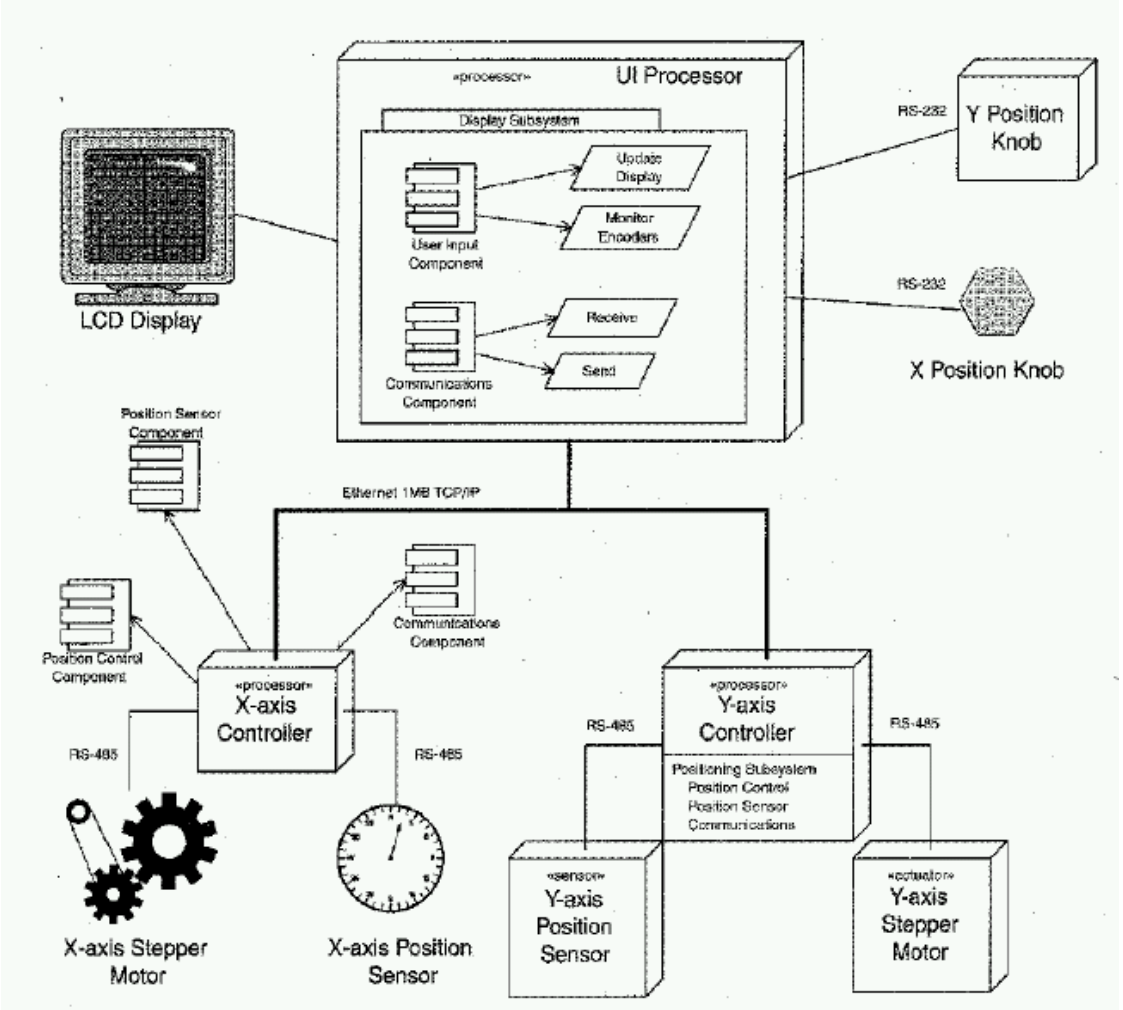
Entwurfsschritte bei der Abbildung der Fachlichkeit

- Auswahl der Hardware
  - meist vorgegeben oder stark eingeschränkt durch Kosten, Größe, Stromverbrauch, Performance
- Auswahl der Basissoftware
  - Auswahl von Basiskomponenten, insbesondere eines RT-OS (falls überhaupt geeignete Komponenten existieren)
- Entwurf der Tasks
  - Identifikation der Tasks
  - Zuordnung von Tasks zu Prozessoren (damit implizit auch Zuordnung von Kommunikationsverbindungen zu Kommunikationsmedien)
  - Zuordnung von Objekten zu Tasks
- Entwurf der Interaktion und Kommunikation
  - Auswahl und Entwurf der Synchronisations- und Kommunikations-mechanismen
  - Zuordnung von Prioritäten zu Tasks

Beispiel für Zuordnung von Tasks zu Prozessoren

Beispiel aus [BP99]

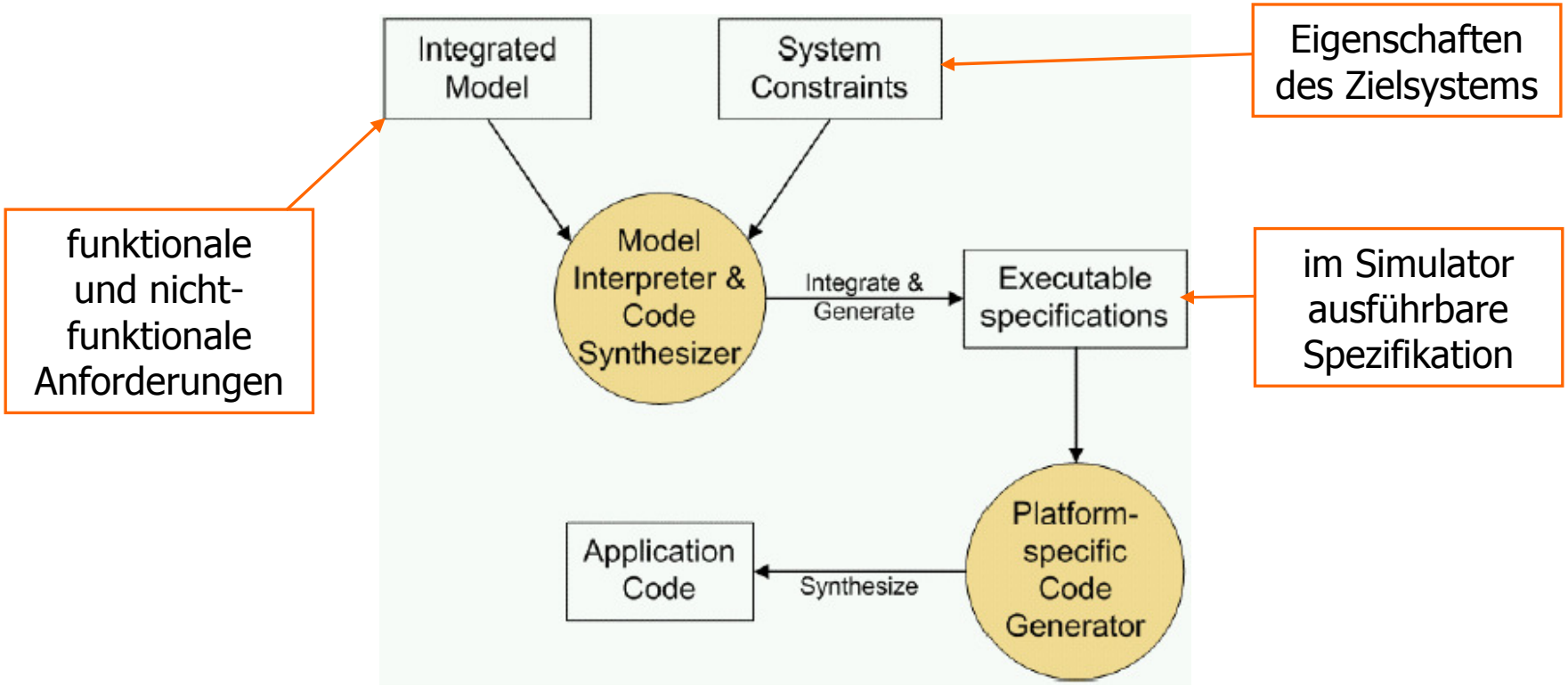
- Verteilungssicht für eine Teleskopsteue-rung, zeigt
  - Hardware-Software-Mapping
  - Aufteilung in Tasks
- Verteilungs- und Deployment-Sicht sind bei statischen eingebetteten Systemen meist isomorph.



Beispiel aus [BP99]

OMG MDA für eingebettete Systeme

- Die Erweiterung der Model-Driven Architecture für eingebettete Systeme ist bisher noch Forschungsgebiet.
- Neben den funktionalen Anforderungen werden auch Modelle der nichtfunktionalen Anforderungen und der Hardware erstellt.



Embedded Container Architectures

- Neues Forschungsgebiet: Übertragung der erfolgreichen Konzepte von Container-Architekturen auf eingebettete Systeme.
- Grundsätzliche Idee
  - Entwickler entwerfen und implementieren funktionale Komponenten mit möglichst wenig technischen Aspekten.
  - Nichtfunktionale Anforderungen werden mit Hilfe des Deployment-Deskriptors spezifiziert.
  - Der Container verwaltet dann die Komponenten so, dass sie automatisch korrekt ins Scheduling eingebunden werden etc.
  - Erwartete Vorteile sind Konfigurierbarkeit, Unabhängigkeit von der technischen Infrastruktur und Offenheit für neue Anwendungen.
- Erste Ansätze basieren auf Generierung eines zugeschnittenen Containers für die spezielle Menge von Anwendungen und benötigten Diensten– insgesamt damit sehr ähnlich zu MDA-Ansatz.

Inhalt

- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklung und Deployment
- Zusammenfassung

Entwicklung und Deployment

- Die Entwicklung und das Deployment von eingebetteten Systemen sind im Allgemeinen wesentlich komplexer als bei betrieblichen Informationssystemen. Die Gründe sind vor allem:
  - Das Programmiermodell ist inhärent parallel und verteilt.
  - Hardwaregeräte sind wesentlicher Bestandteil des Systems.
  - Die Basissysteme sind sehr divers – oft werden im gleichen System unterschiedliche Sprachen und Basiskomponenten verwendet.
  - Zielsystem und Entwicklungssystem sind meist sehr unterschiedlich.
  - Das Deployment kann sehr langwierig sein (Sicherstellung aller Zeitschranken für gegebene Ressourcen).
- Meist kommen zusätzliche Entwicklungswerkzeuge zum Einsatz:
  - Cross-Development-Tools erlauben die Entwicklung für ein unterschiedliches Zielsystem.
  - Simulatoren erlauben den Test des Systems auf dem Entwicklungssystem.
  - Protocol-Analyzer erlauben die Aufzeichnung von Abläufen, ohne das zeitliche Verhalten des Systems zu beeinflussen.

Inhalt

- Eingebettete Systeme
- Fachliche Architektur
- Technische Architektur
  - Basismechanismen und Infrastruktur
  - Architektur- und Entwurfsmuster
  - Umsetzung der Fachlichkeit
- Entwicklungs- und Deployment-Architektur
- Zusammenfassung

Zusammenfassung

- Im Bereich der eingebetteten Systeme gibt es eine Fülle von Modellierungs- und Realisierungstechniken sowie technischen Infrastrukturen und Basiskomponenten.
- Die Standardisierung ist allerdings nicht so fortgeschritten wie bei betrieblichen Informationssystemen.
- In der Praxis gibt es meist einen Methodenbruch zwischen fachlicher Architektur und technischer Architektur (oder schlimmer: eine fachliche Architektur wird überhaupt nicht erstellt).
- Um ein System realisieren zu können, muss der Entwickler heutzutage in fast allen Fällen über ein umfassendes und detailliertes Verständnis der technischen Infrastruktur (inklusive der Hardware) verfügen.
- Neue Ansätze basierend auf Generierung und Container-Architekturen sind größtenteils erst noch in der Forschung.

Literaturhinweise

[AF03] AutoFocus Home Page, http://auto-focus.in.tum.de, 2003.

[BP03] Bruce Powell: Real-Time Design Patterns, Real-Time Design Patterns, Addison-Wesley 2003.

[BP99] Bruce Powell: Real-Time UML 2nd Edition – Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1999.

[D+02] Douglas Schmidt, Michael Stahl, Hans Rohnert, Frank Buschmann: Pattern-orientierte Software-Architektur II, dpunkt.verlag 2002.

[DT02] K.V.K.K. Prasad, Vikas Gupta, Avnish Dass, Ankur Vema: Programming for Embedded Systems, Wiley 2002.

[DS01] Hassan Gamaa: Designing Real-Time and Embedded Systems with the COMET/UML Method, Dedicated Systems Magazine, Q1 2001.

[HNS99] Christine Hofmeister, Robert Nord, Dilip Soni: Applied Software Architecture, Addison Wesley – Object Technology Series, 1999.

[Lee02] E. A. Lee. Embedded Software. Advances in Computers (Marvin V. Zelkowitz, ed.), Vol. 56, Academic Press, London, 2002

[RJ01] RT-Java Expert Group: The Real-Time Specification for Java, Addison-Wesley, 2000.

[SW01] Bran Selic, Ben Watson: The Real-Time UML Standard – Theory and Application, unter http://www.rational.com, 2001.