

Komposition

Echtzeitsysteme 2 - Vorlesung/Übung

Fabian Scheler
Peter Ulbrich
Wolfgang Schröder-Preikschat

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander Universität Erlangen-Nürnberg

<http://www4.cs.fau.de/~{scheler,ulbrich,wosch}>
{scheler,ulbrich,wosch}@cs.fau.de



Übersicht

- Einordnung
- Grundlagen
- Problematik der Ad-Hoc-Methode
- Komposition: Forschung



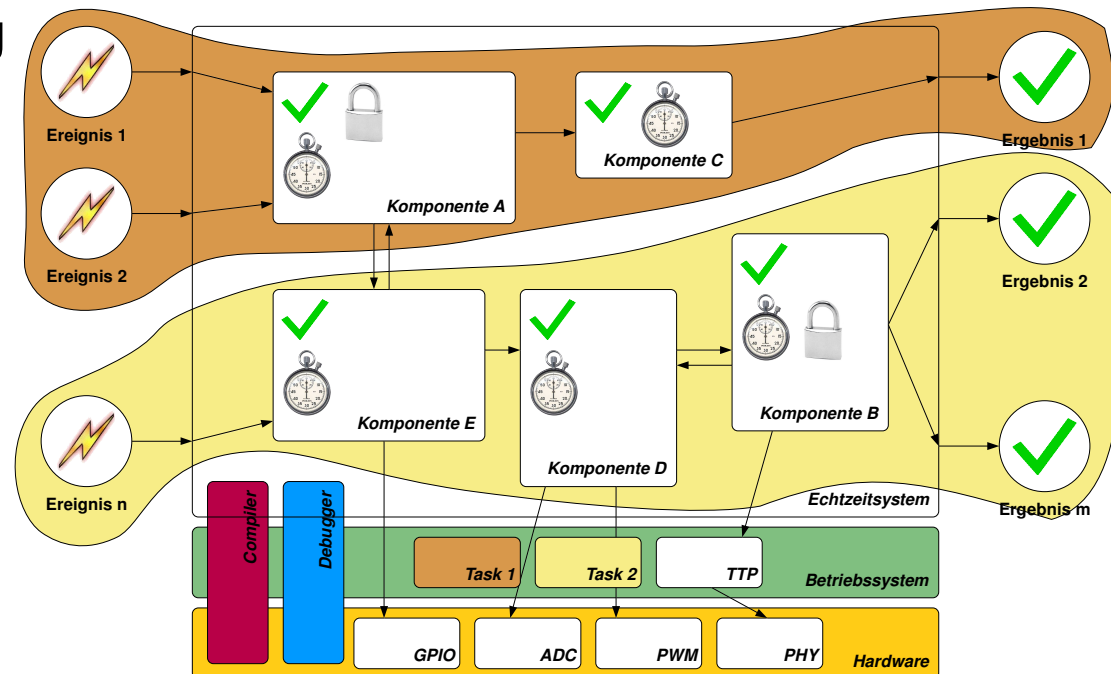
Einordnung

■ Abbildung:

- Komponenten → Ereignisbehandlungen
- Ereignisbehandlungen → Betriebssystem

■ Planung:

- Prioritätenvergabe & statische Ablaufplanung
- Zulässigkeitsprüfung



Grundlagen

■ Teil 1: **Abbildung**

Ereignisbehandlungen → BS-Dienste

- Verknüpfung von Ereignissen mit **Ereignisbehandlungen**
- **Abhängigkeiten** verschiedener Ereignisbehandlungen
- **Synchronisation** kritischer Abschnitte
- Modellierung von **Datenabhängigkeiten**
- ...

■ Teil 2: **Ablaufplanung**

- Berechnung **statischer Ablaufpläne**
- Auswahl eines geeigneten **Algorithmus** (RMA, DMA, EDF, ...)
- **Planbarkeitsanalyse**: Antwortzeitanalyse, CPU-Auslastung, ...
- ...



Komposition: Eingaben

■ Ereignisbehandlungen

- welche **Ereignisse**:
 - zeitliche Parameter: periodisch, aperiodisch, sporadisch
 - Herkunft: physikalische bzw. logische Ereignisse
- welche **Termine**: hart, fest, weich
- **Abhängigkeiten**
 - gerichtete Abhängigkeiten
 - gegenseitiger Ausschluss ↔ kritische Abschnitte

■ Komponenten

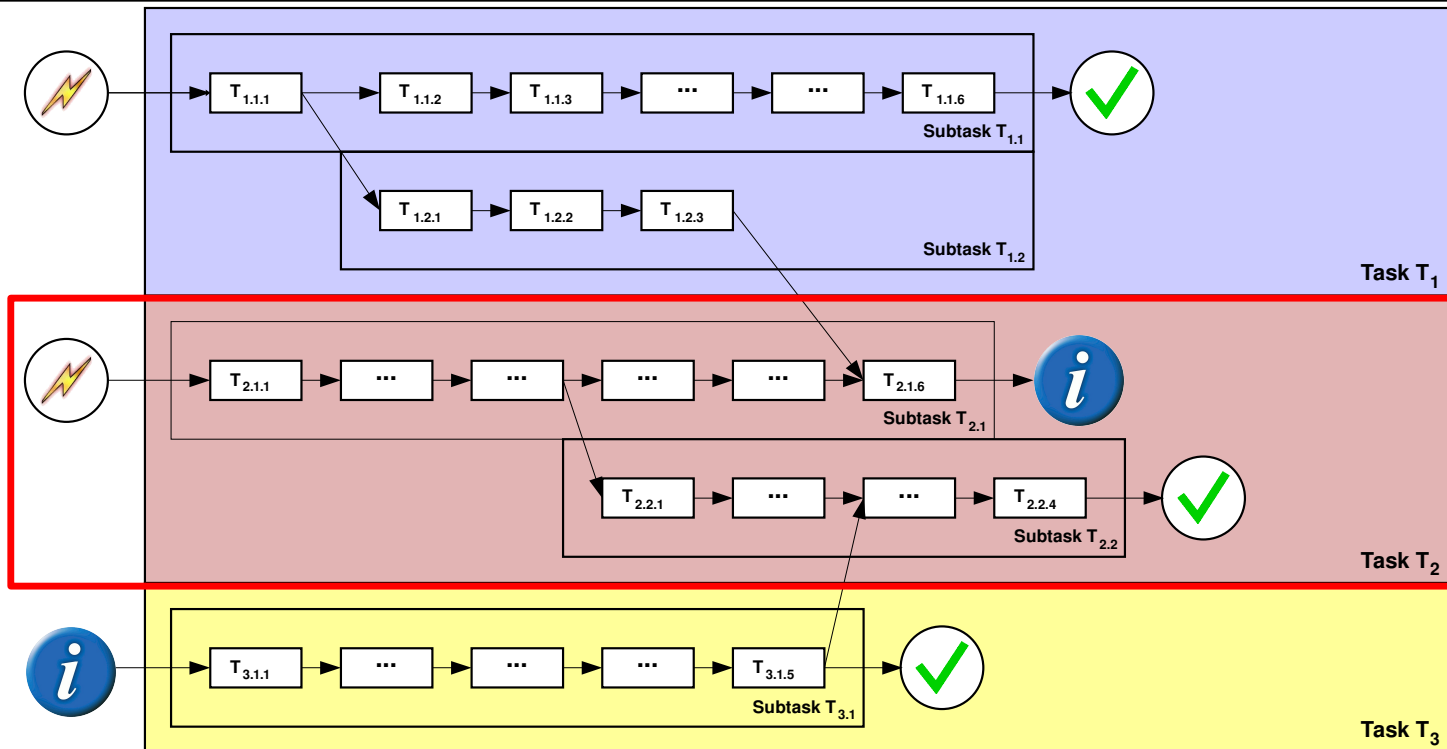
- funktionale und temporale Spezifikation

■ Laufzeitsystem

- funktionale und temporale Spezifikation



Ereignisbehandlungen

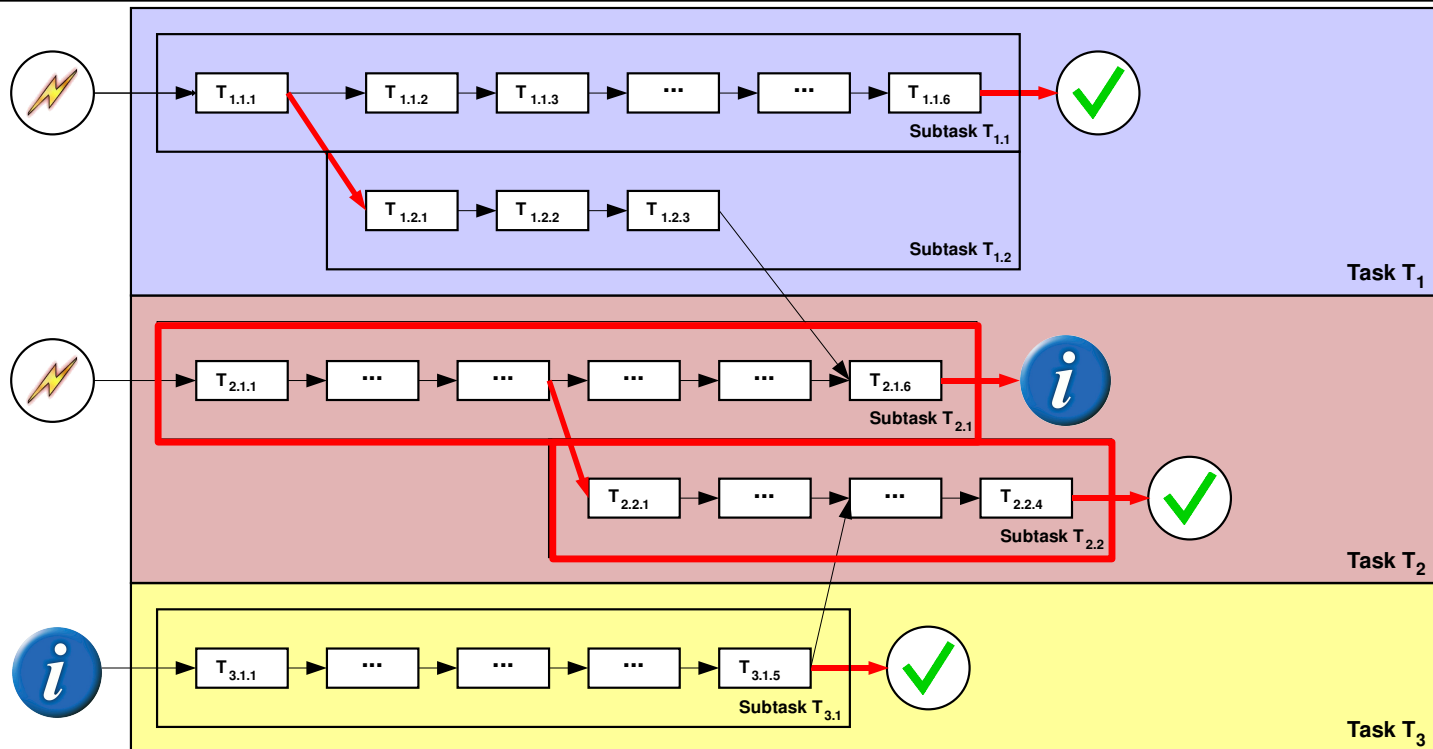


■ Tasks

Reaktionen auf **physikalische** und **logische Ereignisse** sind **Ereignisbehandlungen** bzw. **Tasks**. Tasks und ihre **auslösenden Ereignisse (Auslöser)** sind durch eine **Periode**, eine **Phase** und einen **Jitter** (periodische Ereignisse) oder eine **minimale Zwischenankunftszeit** charakterisiert (nicht-periodische Ereignisse). Häufig werden Tasks und ihre Auslöser synonym verwendet.



Ereignisbehandlungen

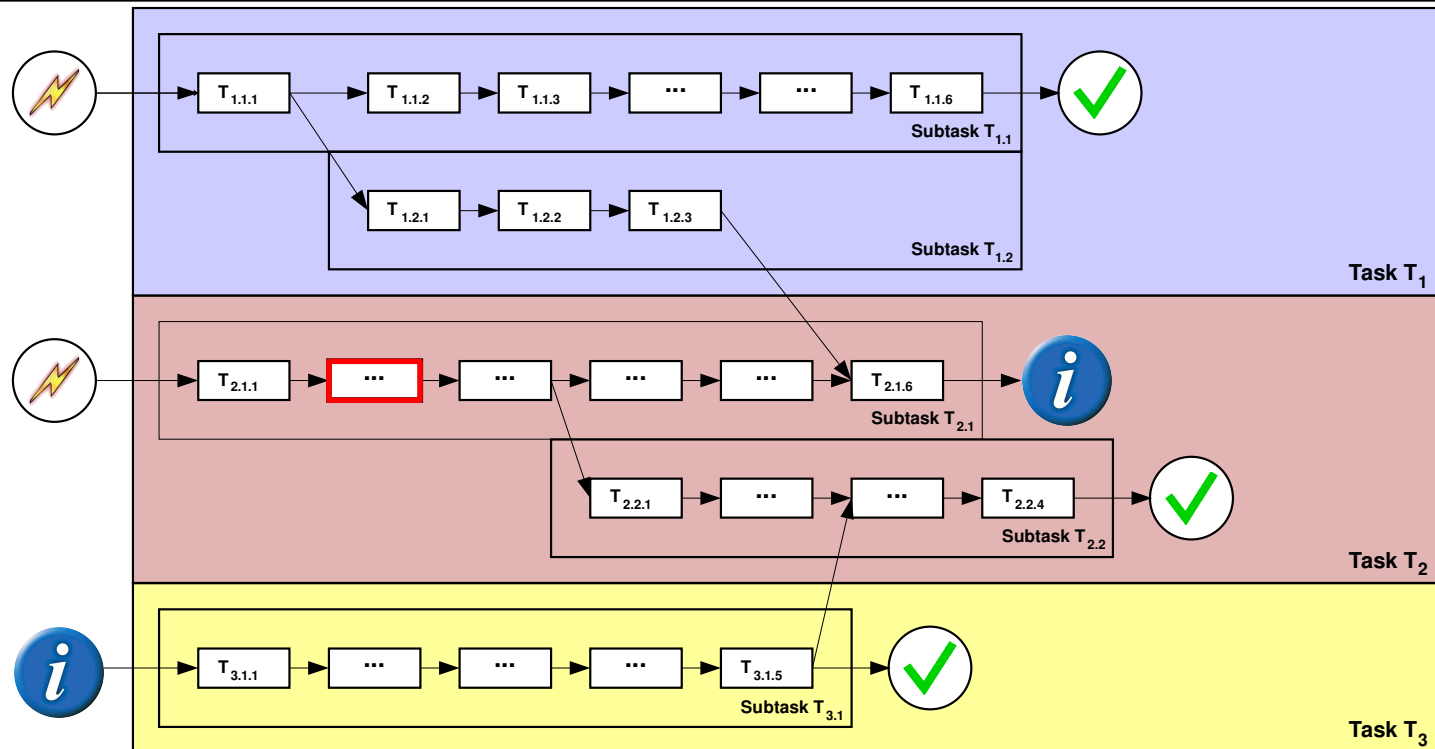


■ Subtasks

Tasks beinhalten einen oder mehrere **Subtasks**, die ein Ereignis auf verschiedene Weisen behandeln. Hierfür können von einem Subtask weitere Subtasks **abgezweigt** werden. Subtasks können dabei ein Ergebnis oder ein logisches Ereignis erzeugen. Darüber hinaus können Subtasks mit **harten, festen** oder **weichen Terminen** versehen sein.



Ereignisbehandlungen

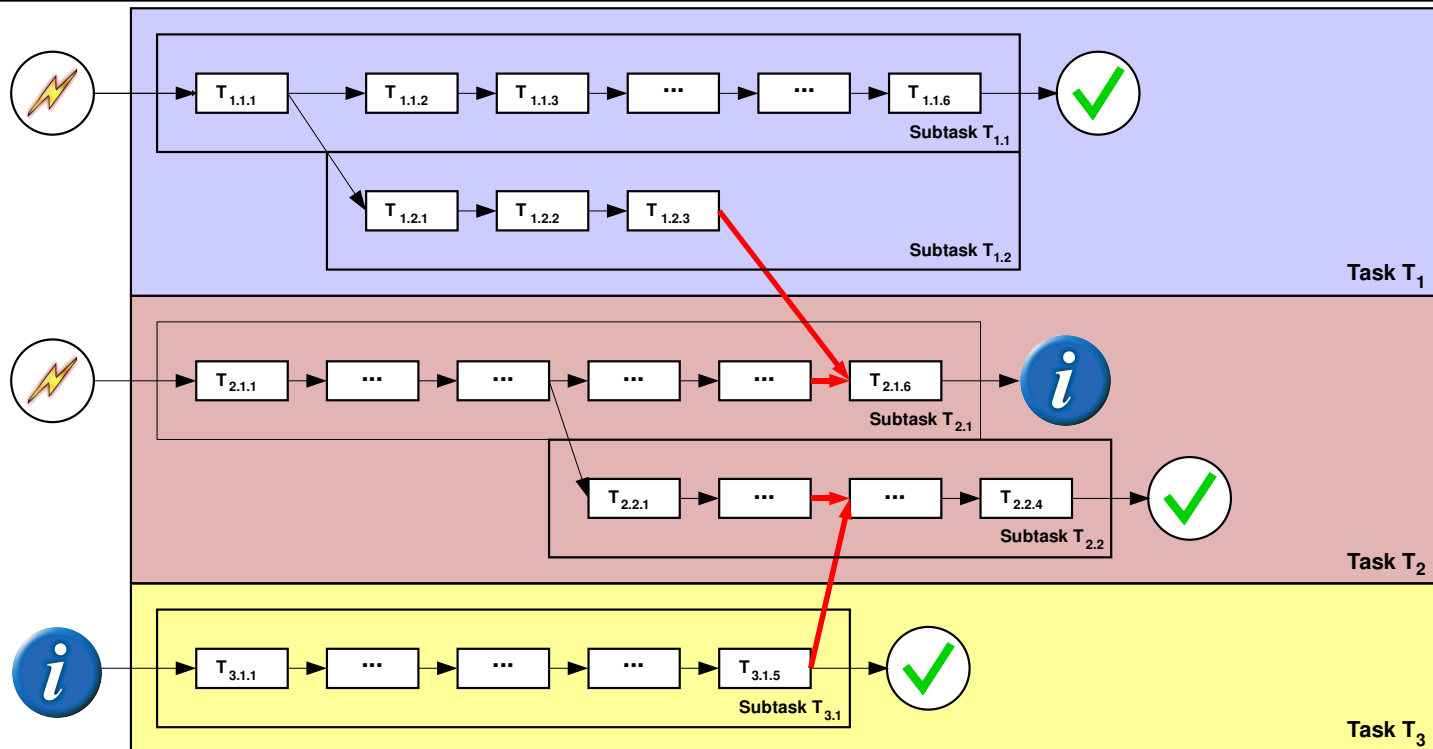


Atomic Basic Blocks (ABBs)

Subtasks bestehen aus einem oder mehreren **ABBs**. Innerhalb eines ABB findet keine Interaktion mit anderen Subtasks statt. Endpunkte von ABBs bilden z.B. die Freigabe eines Mutex, die Aktivierung eines Threads oder das Warten auf ein Signal. Jeder ABB besitzt eine bekannte **maximale Ausführungszeit (WCET)**, zusätzlich kann auch er mit einem **harten, festen** oder **weichen Termin** versehen sein.



Gerichtete Abhängigkeiten



■ Oder-Verknüpfung von Abhängigkeiten

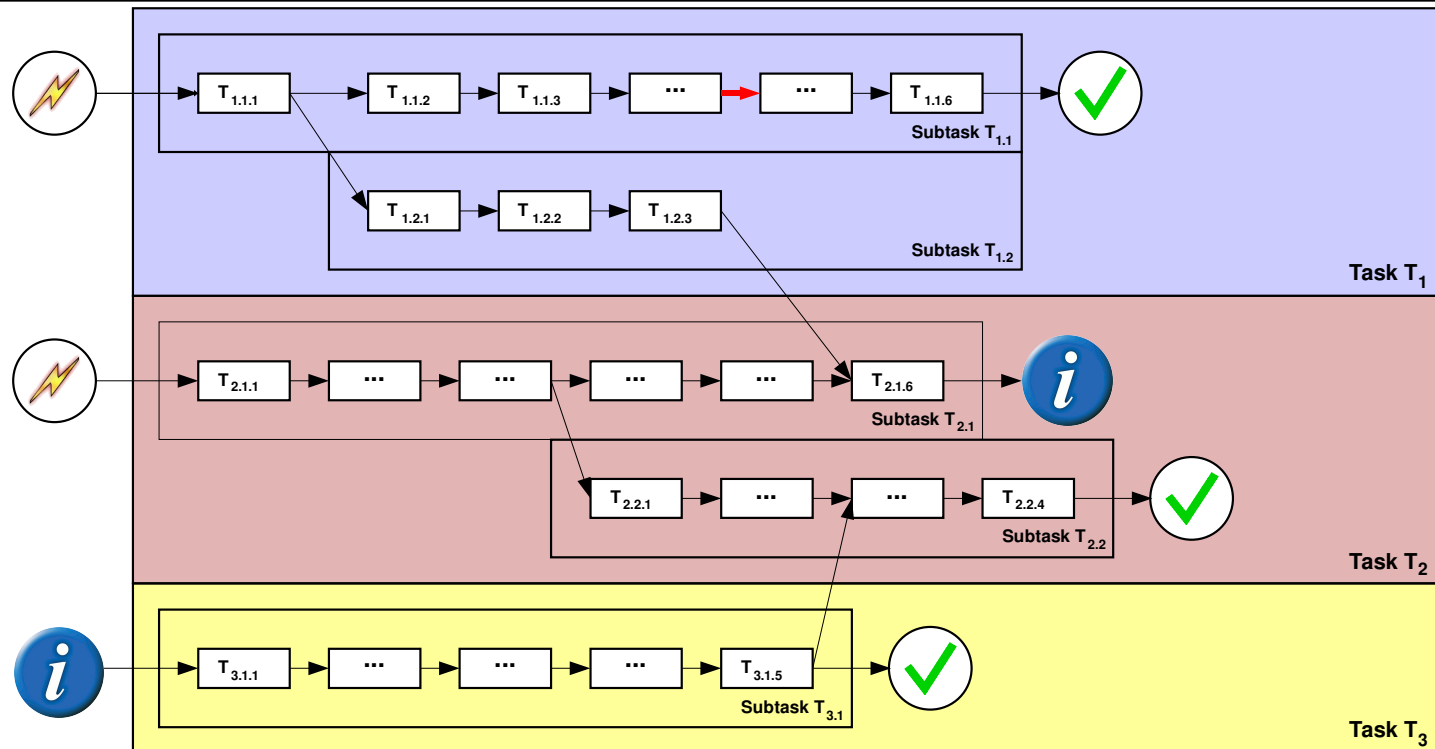
Ausführung **eines** Vorgänger ist abgeschlossen → Ausführung des Nachfolgers

■ Und-Verknüpfung von Abhängigkeiten

Ausführung **aller** Vorgänger ist abgeschlossen → Ausführung des Nachfolgers



Gerichtete Abhängigkeiten

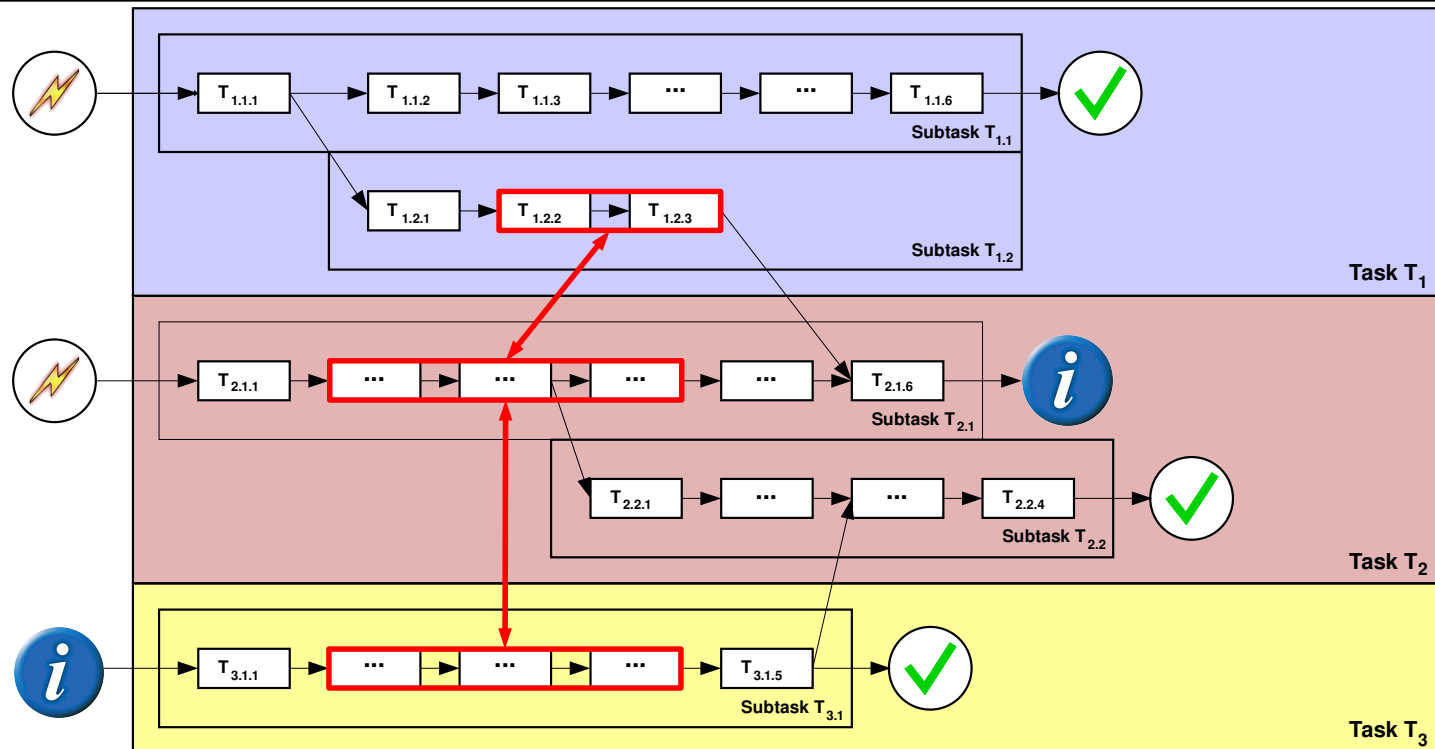


■ zeitliche Abhängigkeiten

Der Nachfolger wird **zeitlich verzögert** ausgeführt. Die Angabe der Verzögerung kann unterschiedlich genau sein: nach Ablauf einer bestimmten Zeitspanne x (z.B. $x = 60\text{ms}$), nach Ablauf einer durch Aufzählung bestimmbarer Zeitspanne ($x = \{30\text{ms}, 60\text{ms}, 120\text{ms}\}$), nicht vor Ablauf einer Zeitspanne y und nicht nach Ablauf einer Zeitspanne z ($30\text{ms} < x \leq 120\text{ms}$).



Ungerichtete Abhängigkeiten



■ Gegenseitiger Ausschluss

Werden Datenstrukturen nebenläufig manipuliert, kann dies zu Inkonsistenzen und somit zu undefiniertem Verhalten führen. Man bezeichnet daher die Abschnitte, in denen diese Manipulationen durchgeführt werden, als **kritische Abschnitte**. Zwischen kritischen Abschnitten muss i.d.R. der **gegenseitige Ausschluss** gewährleistet werden, falls die Datenstrukturen nicht anderweitig synchronisiert werden können.



Abbildung

- Elemente werden durch ein Laufzeitsystem implementiert
- Folgende Elemente müssen unterstützt werden:
 - **Oder-Verknüpfung** von Abhängigkeiten
 - **Und-Verknüpfung** von Abhängigkeiten
 - **zeitliche Abhängigkeiten**
 - **gegenseitiger Ausschluss**
 - **Abzweigen** von Subtasks
- Implementierung auf verschiedene Art und Weise möglich
 - **explizit**
 - **implizit**

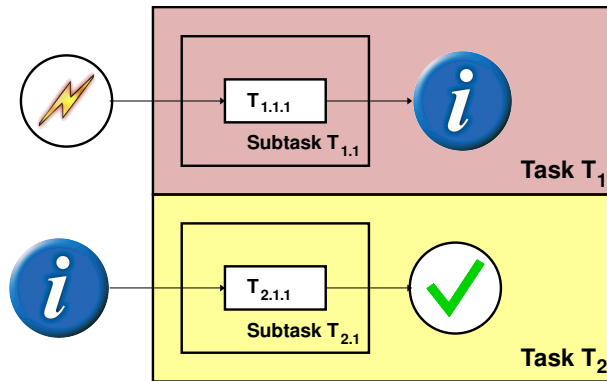


Tasks/Subtasks – explizit und implizit

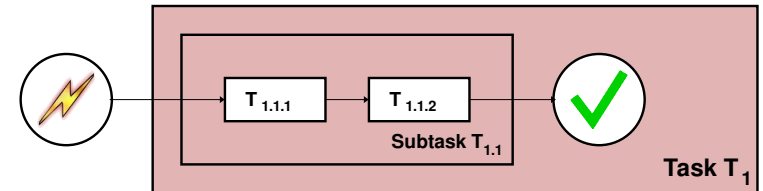
■ Szenario

- Annahme von Daten an einer Kommunikationsschnittstelle
- angenommene Daten bilden Pakete
 - Pakete sind i.d.R. fragmentiert
 - Pro Paket sind mehrere Sende-/Empfangsvorgänge notwendig

■ explizite Implementierung ■ implizite Implementierung



- Unterbrechung → Task T_1
- Paketempfang → Task T_2



- Unterbrechung & Paketempfang → Task T_1



Implementierung

■ explizit

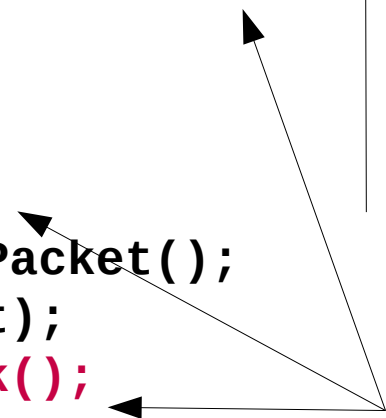
```
ISR(serial) {  
    getSerialData();  
    bufferSerialData();  
  
    if(completePacket()) {  
        ActivateTask(packet);  
    }  
}
```

```
Task(packet) {  
    packet = getPacket();  
    buffer(packet);  
    TerminateTask();  
}
```

■ implizit

```
ISR(serial) {  
    getSerialData();  
    bufferSerialData();  
  
    if(completePacket()) {  
        packet = getPacket();  
        buffer(packet);  
    }  
}
```

explizite Implementierung



Implikationen

■ Annahmen

- Periode: $p_1 < p_2$
 - Pakete sind fragmentiert und bestehen aus mehreren Datenelementen
- WCET: $e_{1.1.1} < e_{2.1.1}$
 - Behandlung eines Paketes i.d.R. algorithmisch aufwendiger

■ Overhead

- WCET: $e_{1.1.1} + e_{2.1.1} \geq e_{1.1.1} + e_{1.1.2}$
 - explizite Implementierung erfordert z.B. die Aktivierung eines Fadens

■ Systemauslastung (normalerweise)

- $U_{expl} = \frac{e_{1.1.1}}{p_1} + \frac{e_{2.1.1}}{p_2} < U_{impl} = \frac{e_{1.1.1} + e_{1.1.2}}{p_1}$
 - Schritt von impliziter zu expliziter Implementierung ist **nicht erkennbar**
 - Overhead übersteigt den Gewinn der expl. Implementierung i.d.R. nicht
- **das muss aber nicht so sein!**



Gerichtete Abhängigkeiten

■ Szenario

- verschiedene Sensoren sind über einen Bus angebunden
- verschiedene Tasks greifen auf diese Sensoren zu

→ Oder-Verknüpfung

- in jedem Fall wird Bus-Kommunikation notwendig



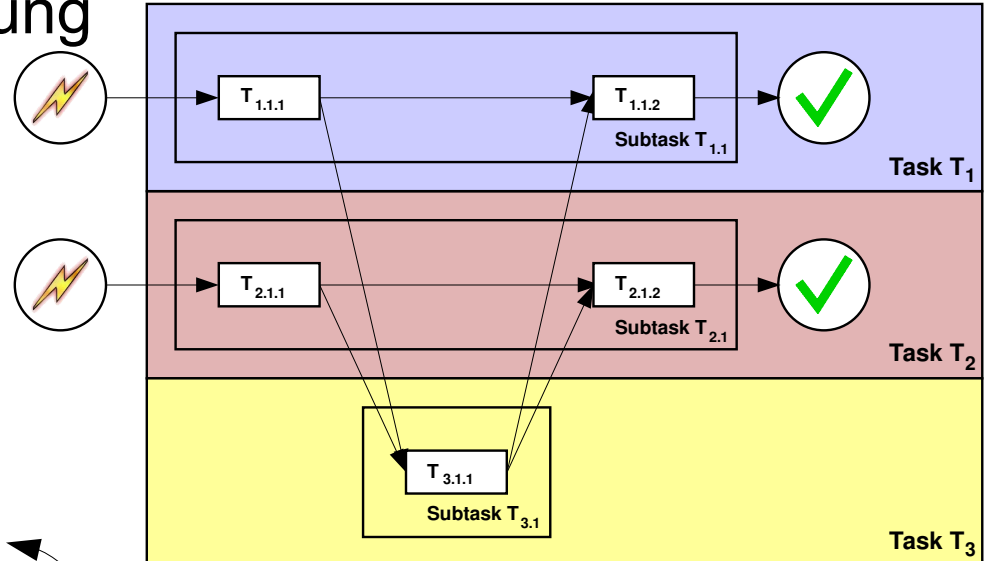
Explizite Implementierung

■ explizite Implementierung

- Gyroskop → Task T_1
- Kompass → Task T_2
- Bus → Task T_3

```
Task(Task1) {
    Message m;
    m.sender    = Task1;
    m.event     = Event1;
    m.data      = getData();
    m.receiver  = receiver;

    putMessage(m);
    ActivateTask(Task3);
    WaitEvent(Event1);
    ...
    TerminateTask();
}
```



```
Task(Task3) {
    Message m = getMessage();
    process(m);
    SetEvent(m.sender, m.event);
}
```

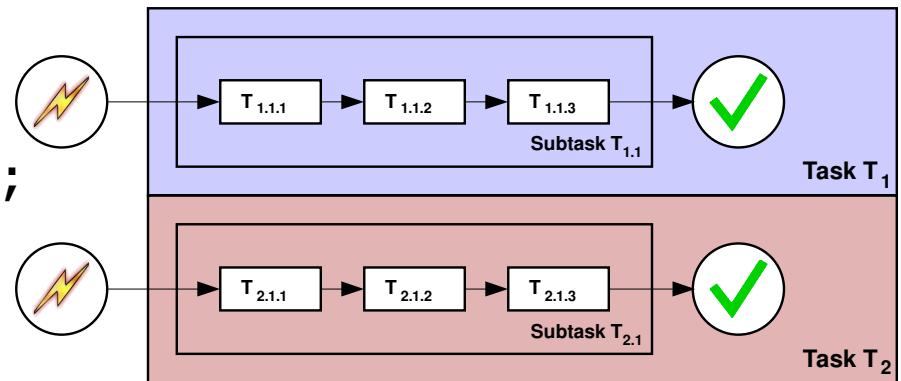
explizite Implementierung



Implizite Implementierung

- implizite Implementierung
 - Gyroskop + Bus $\rightarrow$ Task T_1
 - Kompass + Bus $\rightarrow$ Task T_2

```
Task(Task1) {  
    Message m;  
    m.data      = getData();  
    m.receiver  = receiver;  
  
    process(m);  
    ...  
    TerminateTask();  
}
```



Implikationen

- **Overhead** der expliziten Implementierung
 - pro Ausführung von Task1/Task2 → **2 Kontextwechsel**
- **Synchronisation**
 - **explizit**: Task1/Task2 und Task3 → **gemeinsamer Puffer**
 - Zugriff auf den Puffer muss synchronisiert werden
 - **implizit**: Task1 und Task2 greifen auf den Bus zu
 - der **exklusive Zugriff auf den Bus** muss sichergestellt werden
- **Ablaufplanung** in der expliziten Implementierung
 - Task1 und Task2 können unterschiedliche Priorität besitzen
 - **Prioritätsverletzungen** sind die Folge
 - bei der Verwaltung der Nachrichten im Puffer
 - durch die Ausführung von Task3

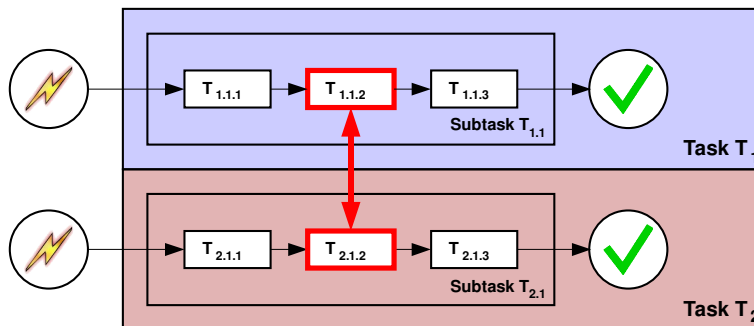


Ungerichtete Abhängigkeiten

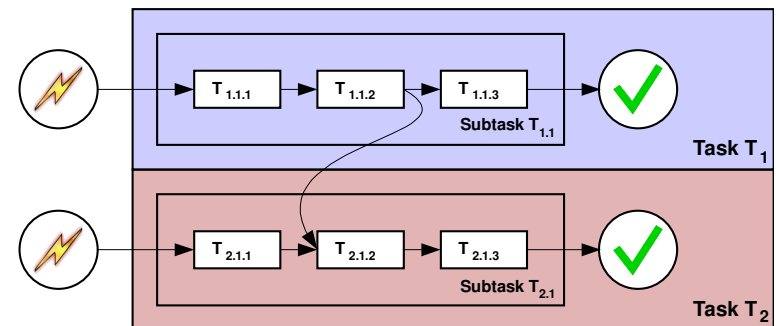
■ Szenario: Kommunikationssystem

- Daten kommen von einer physikalische Schnittstelle
 - Bestandteile von fragmentierten Paketen
- Kommunikationssystem formt fertige Pakete
- **Pufferung** der Fragmente ist notwendig
 - Puffer ist **gemeinsam genutztes Betriebsmittel**
 - Zugriff auf den Puffer **muss synchronisiert** werden

■ explizit



■ Implizit



- phys. Schnittstelle → Task T₁

- Komm.system → Task T₂



Implikationen

■ Mittel für eine **explizite Implementierung**

■ **Schlossvariable**

- implementiert durch Sperren von Unterbrechungen, PCP, PIP, ...

→ **Blockierung** von höher-prioren Aktivitätsträgern

- Ausmaß (Dauer und Umfang) hängt von der Implementierung ab

→ gegenseitiger Ausschluss durch **Sequentialisierung zur Laufzeit**

- unklar welcher Aktivitätsträger die Schlossvariable erfolgreich belegt
- signifikante **Schwankungen der Antwortzeit** können die Folge sein

■ implizite Implementierung

■ **statische Sequentialisierung** zur Übersetzungszeit

- häufig: **zeitliche Parameter** sind **nicht vollständig bekannt**
- diese muss nicht optimal sein

■ auch hier kann die Antwortzeit Schwankungen unterworfen sein

- abhängig von den **Schwankungen der Ausführungszeiten** von T_1 und T_2

→ **phasenverschobene Ausführung** minimiert diese Abhängigkeit



Ablaufplanung

- Tasks und Subtasks beschreiben eine **partielle Ordnung**
- Ziel: Überführung in eine **totale Ordnung**
 - unter Einhaltung aller Abhängigkeiten
- Implementierung der Abhängigkeiten
 - **Implizit**
 - in **zeitgesteuerte Systemen** mit **statischer Ablaufplanung**
 - **Explizit** und **Implizit**
 - in **ereignisgesteuerten Systemen** mit **dynamische Ablaufplanung**



Statische Ablaufplanung

- **Constraint Programming** (*Schild & Würz*)

Das Planungsproblem wird als Problem der mathematischen, linearen Ganzzahlprogrammierung aufgefasst, das es unter Einhaltung diverser Nebenbedingungen zu lösen gilt.

- **Cyclic Executive Model** (Shaw & Baker)

Der Ablaufplan besteht aus Major- und Minor-Cycles, Konstruktion des Ablaufplans durch Festlegung/Berechnung gewisser Eigenschaften von Major- und Minor-Cycles.

- **Suchen** (*Fohler*)

Das Absuchen des kompletten Suchraums aller möglichen Ablaufpläne ist wegen der NP-Komplexität des Planungsproblems nicht möglich, daher wird auf heuristische Algorithmen wie A* und IDA* zurückgegriffen.

- **genetische Algorithmen** (*Hou*)

Verfahren für die Ablaufplanung in Multiprozessorsystemen. Die Suchknoten basieren auf der Reihenfolge der Tasks, die auf einem Knoten abgearbeitet werden, der genetische Operator auf den Abhängigkeiten zwischen den verschiedenen Tasks.

- **von Hand** (*ad-hoc Methoden*)



Planbarkeitsanalyse

■ Bestimmung der **maximalen CPU-Auslastung**

- Perioden p_i , Termine d_i und WCET e_i aller n Tasks t_i sind bekannt

- CPU Auslastung:
$$U = \sum_{i=1}^n \frac{e_i}{\min(p_i, d_i)}$$

- EDF:
$$U \leq 1$$

- RMA & DMA:
$$U(n) = n(2^{1/n} - 1); d_i = p_i \forall 1 \leq i \leq n$$

■ **Antwortzeitanalyse**

- Perioden p_i , Termine d_i und WCET e_i aller n Tasks t_i sind bekannt

- $\text{Priorität}(t_i) > \text{Priorität}(t_{i+1})$

- Zeitbedarf:
$$w_i(t) = e_i + \sum_{k=1}^{i-1} \text{ceiling}\left(\frac{t}{p_k}\right) e_k$$

- prüfe:
$$w_i(t) \leq t; t = jp_k; k = 1, 2, \dots, i; j = 1, 2, \dots, \text{floor}(\min(d_i, p_i)/p_k)$$



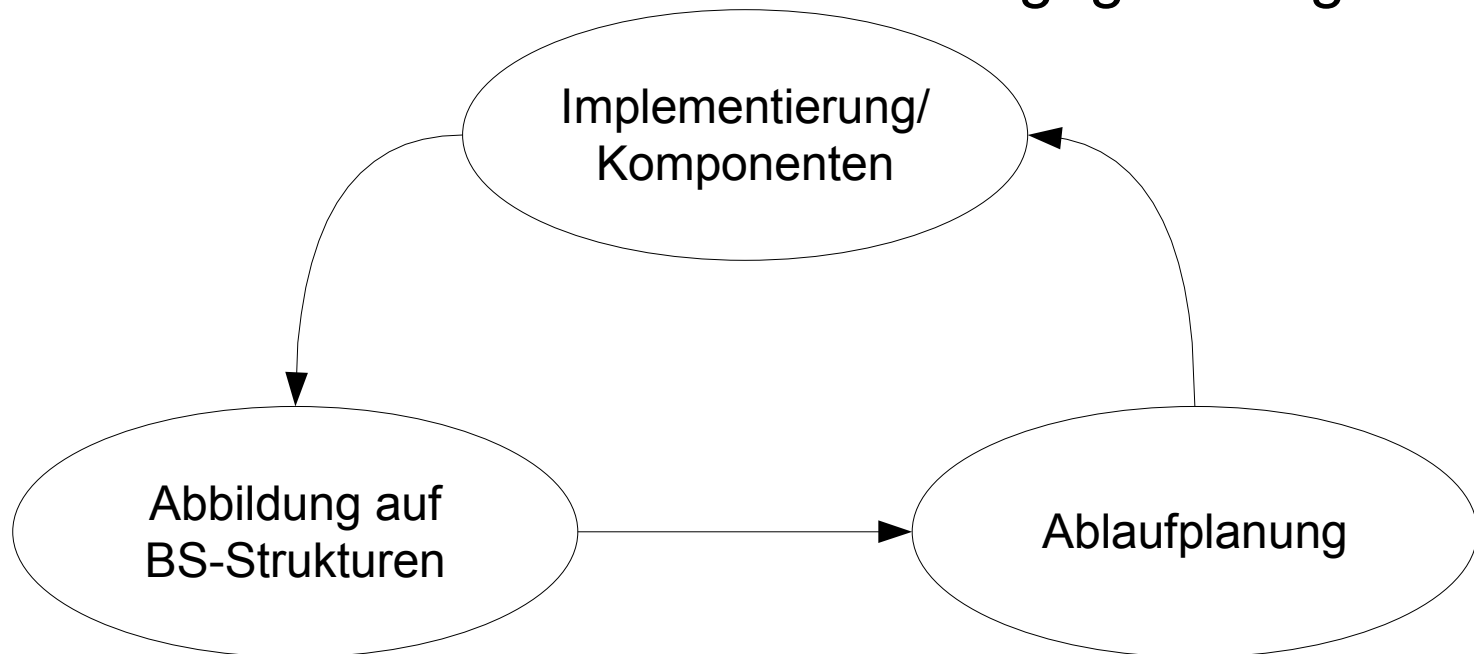
Komposition: traditionell

- oft erfolgen einige Schritte simultan:
 - Entwicklung der Komponenten
 - Abbildung auf BS-Strukturen
 - Berechnung statischer Ablaufpläne / Planbarkeitsanalyse
- das ist **problematisch**



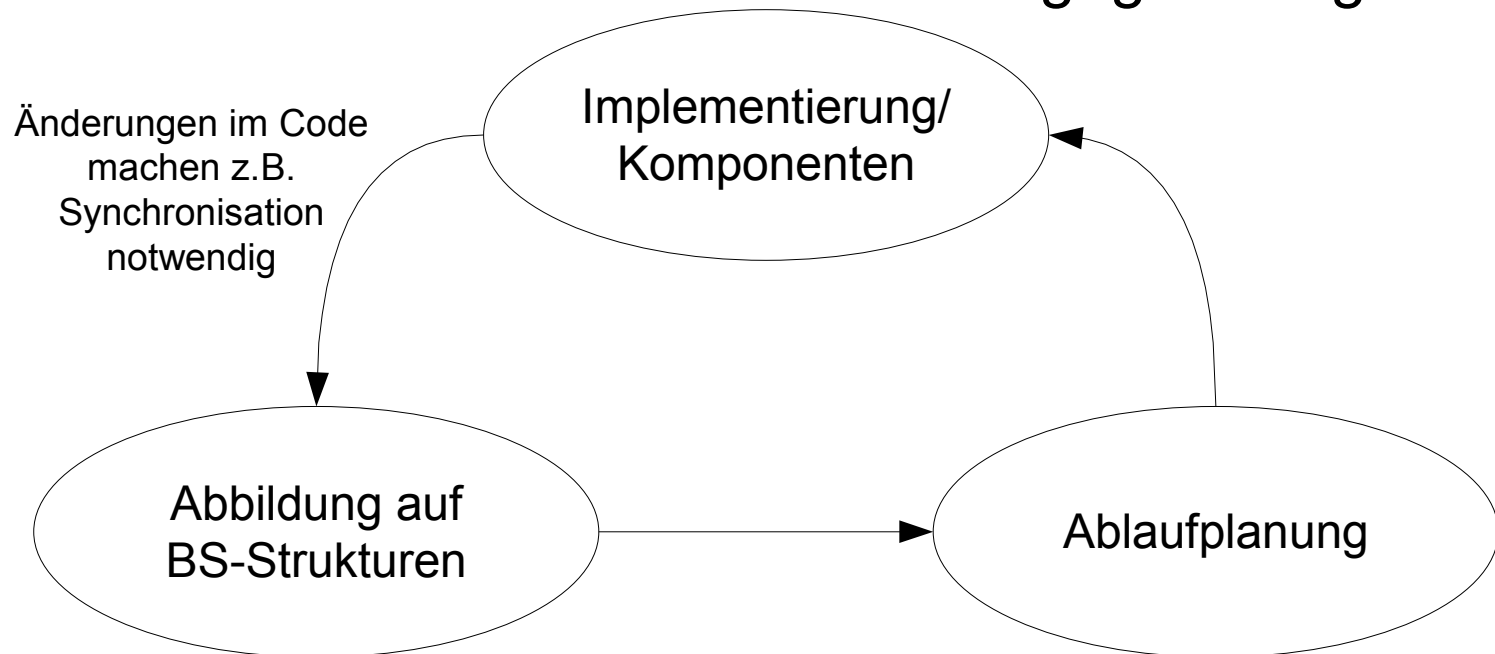
Probleme & Folgen

- Problem: Schritte beeinflussen sich gegenseitig



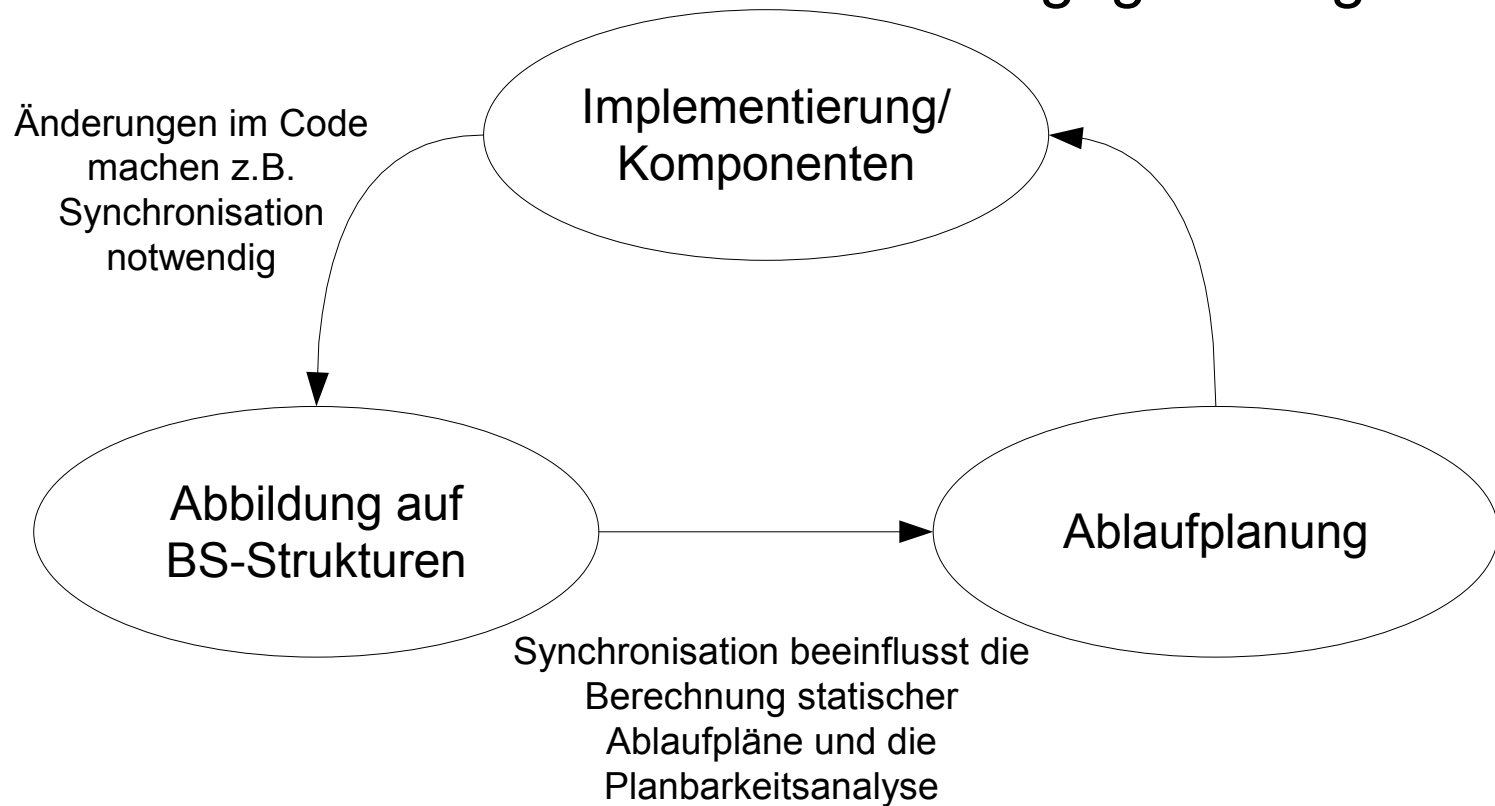
Probleme & Folgen

- Problem: Schritte beeinflussen sich gegenseitig



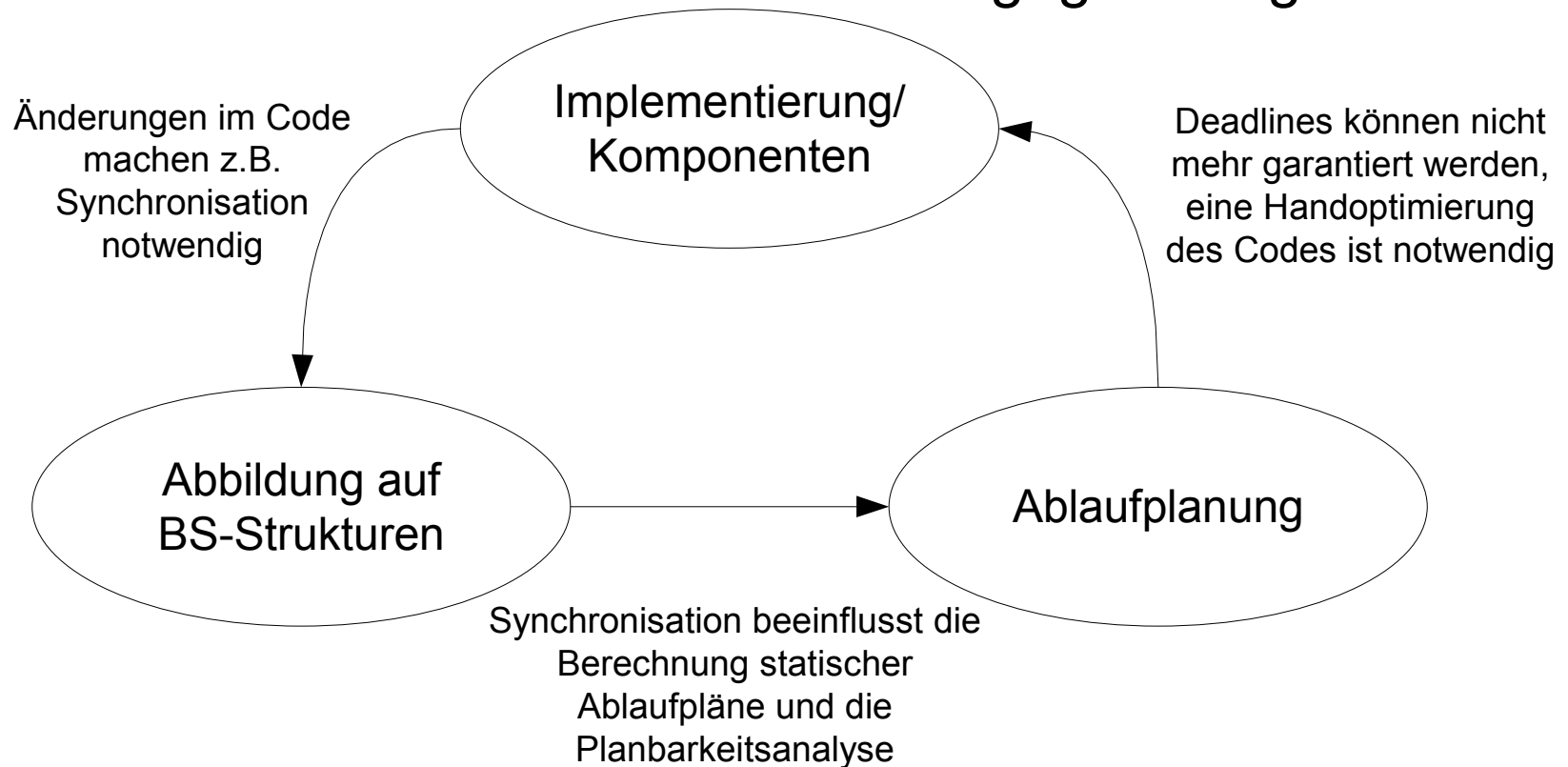
Probleme & Folgen

- Problem: Schritte beeinflussen sich gegenseitig



Probleme & Folgen

- Problem: Schritte beeinflussen sich gegenseitig



- Folge
 - langwieriger Entwicklungsprozess
 - schlecht wartbarer Code



Probleme & Folgen

- **Problem:** Abbildung auf BS-Dienste erfolgt zu früh
Modularisierung findet auf der Ebene von BS-Strukturen statt, z.B. LED-Task, Lichtschranken-ISR ...

- **Folgen:**
 - System ist **nicht wiederverwendbar**
 - bestimmte BS-Dienste existieren nicht unbedingt in allen BS
 - Portierungsaufwand
 - Architekturunterschiede: TT vs. ET
 - **Overhead**
 - Einsatz von BS-Diensten ist **immer teurer!**
 - Komponenten / Module auf der Ebene von BS-Diensten ziehen ausgiebigen Gebrauch von BS-Diensten nach sich!



What you need ...

*„For operating-system technology, the cat is out of the bag. Fundamentally, **every RTOS is the same as every RTOS. What you need is a way to divide your problem into tasks** and have sufficient computing power. Then you want to have a priority-based preemptive kernel. And they're all the same, whether you get your OS out of a book or with free source code included, or you get something else free. ...“ [1]*

Michael Barr
(former editor-in-chief Embedded Systems Design)



Komposition: Forschung

■ Ziele:

- Entkopplung der einzelnen Schritte
- Automatisierung der einzelnen Schritte
- Einhaltung gewisser Randbedingungen:
 - Rechtzeitigkeit
 - Security / Safety
 - Zuverlässigkeit
 - ...



Forschung: Aufteilung

■ Grundlagenforschung

- stellt Mechanismen zur Verfügung,
 - um Ereignisbehandlungen zu implementieren
 - die es erlauben, Aussagen über gewisse Eigenschaften zu treffen
- Forschungsgebiete
 - Ablaufplanung
 - WCET-Analyse
 - Echtzeitbetriebssysteme
 - ...

■ Werkzeugunterstützung

- basiert auf den Ergebnissen der Grundlagenforschung
- (halb)automatische Anwendung der entwickelten Mechanismen



Forschung: Beispiele

- Grundlagenforschung:
 - Time-Triggered Architecture (TTA)
- Werkzeugunterstützung
 - Cluster Compiler



TTA & Cluster Compiler

- Entwicklungsziel:
sicherheitskritische, verteilte Echtzeitsysteme
- TTA: Grundlage/Architektur [2]
- Cluster Compiler: Entwicklungswerkzeug [3]
- Entwickler: TU Wien – Kopetz et. al
- Einsatzgebiete
 - Luftfahrt
 - Automobilbau
 - Industrieanlagen
 - Antriebssysteme
 - Beispiel: Kabinendrucksteuerung des A380



TTA: Architekturkonzepte

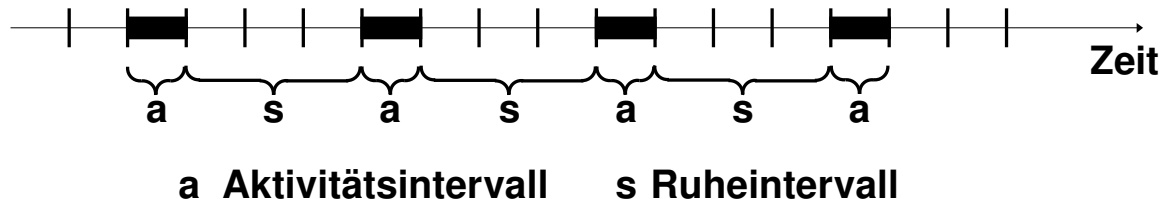
- Modell der Zeit
- Zeit & Zustand
- Echtzeitentitäten und Echtzeitabbilder
- Zustandsinformation & Ereignisinformation
- Struktur
 - einzelner Knoten
 - TTA-Cluster
- Topologien



TTA: Architekturkonzepte

■ Zeit

- basiert auf physikalischer Zeit
- Zeitpunkte auf der Zeitachse sind Ereignisse
 - Beobachtung eines Zustands ist ein Ereignis
- Ereignisse tragen Zeitstempel
- global synchronisierte Zeitbasis
- perfekte Synchronisation nicht möglich
 - oft keine Aussagen über die Folge von Ereignissen möglich
- Lösung: **sparse time base**
 - Ruhe- und Aktivitätsintervalle
 - Ruheintervalle werden durch TTA überwacht
 - Ereignisse lassen sich zeitlich nach ihren Zeitstempeln ordnen



TTA: Architekturkonzepte

■ Zeit & Zustand

- der Zustand entkoppelt Zukunft und Vergangenheit
- Zukunft hängt nur von Zustand und zukünftigen Eingaben ab
- **sparse time base** erlaubt globale Trennung zwischen
 - Zukunft und
 - Vergangenheit

■ Echtzeitentität und Echtzeitabbild

- EZ-Entität: relevante Variablen, die den Zustand kapseln
- EZ-Abbild: zeitlich akkurates Abbild einer Echtzeitentität

■ Zustands- und Ereignisinformation

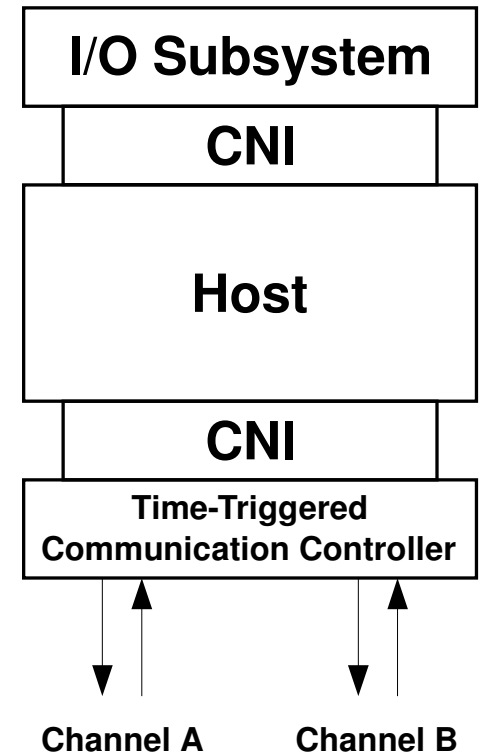
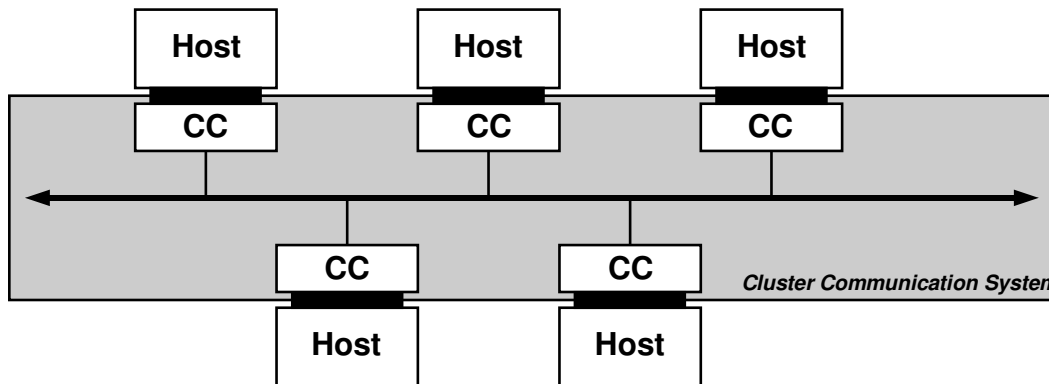
- Zustandsinformation: EZ-Abbild mit Zeitstempel
- Ereignisinformation: Änderung des Zustands
- TTA verwendet ausschließlich Zustandsinformation



TTA: Architekturkonzepte

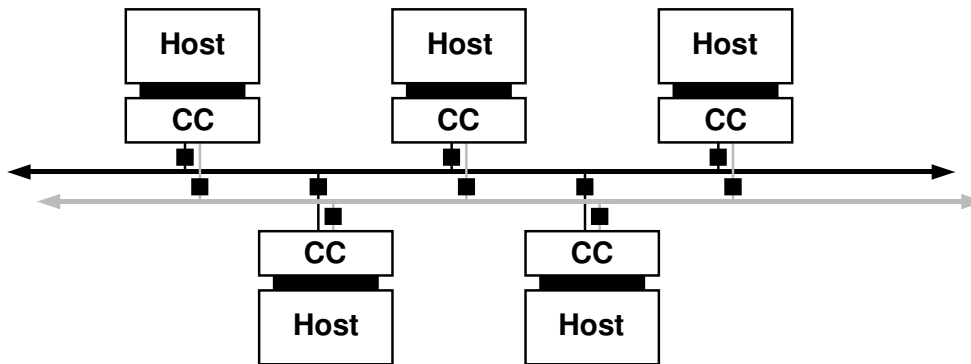
■ **Struktur:** ein TTA-System besteht aus

- Knoten
 - Prozessor/Speicher
 - Communication Network Interface (CNI)
 - zeitgesteuerter Kommunikationscontroller
 - I/O Subsystem
 - BS + Anwendung (Host)
- die zu einem TTA-Cluster gekoppelt sind
 - Cluster Kommunikationssystem – TTP
 - replizierter Broadcast-Kanal (zuverl. Broadcast)

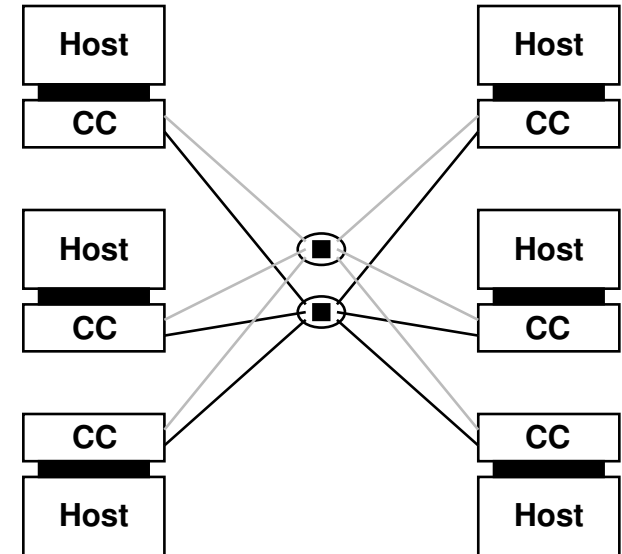


TTA: Architekturkonzepte

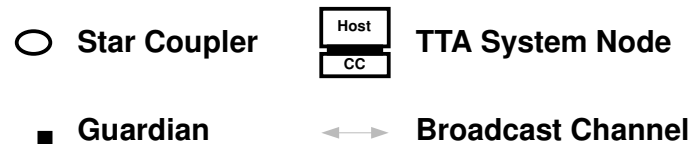
■ Topologie



TTA-Bus



TTA-Stern



TTA: Designkonzepte

- zuverlässige, verteilte Rechnerplattform
- Temporal Firewalls
- Zusammensetzbarkeit
- Skalierbarkeit
- transparente Fehlertoleranz



TTA: Designkonzepte

■ zuverlässige, verteilte Rechnerplattform

- kurze Fehlererkennungslatenz durch zeitgesteuerte Kommunikation
 - Fehlererkennung auf Protokollebene
 - Membership-Service
 - korrektes Verhalten oder Stille (**fail-silent nodes**)
- Mitglieder verhalten sich korrekt

■ Temporal Firewalls

- CNI ist die wichtigste Schnittstelle innerhalb eines TTA-Systems
 - **zwei unidirektionale Kanäle** (Senden & Empfangen)
 - elementare Schnittstelle: **unidirektionaler Kontrollfluß**
 - **Sender übergibt Daten** an das CNI
 - **Empfänger holt Daten** vom CNI ab
 - **Kontrollfehlerausbreitung** aufgrund des Designs **unmöglich**
- Schnittstelle mit diesen Eigenschaften: **Temporal Firewall**



TTA: Designkonzepte

■ Zusammensetzbarkeit

- unabhängige Entwicklung der einzelnen Knoten
Dienste, die ein Knoten zur Verfügung stellt müssen funktional und zeitlich auf Architekturebene spezifiziert werden können.
- Stabilität existierender Dienste
Die Integrität bestehender Dienste wird nicht durch die Integration neuer Knoten kompromittiert.
- konstruktive Integration neuer Knoten
Die Integrität existierender Knoten, darf nicht durch die Integration neuer Knoten beeinträchtigt werden.
- Replikdeterminismus
Replizierte Knoten müssen zur selben Zeit denselben von außen sichtbaren Zustand einnehmen und dieselben Nachrichten versenden.



TTA: Designkonzepte

■ Skalierbarkeit

- Komplexität einzelner Funktionen (Knoten) darf nicht mit der Komplexität des Gesamtsystems wachsen
- horizontale & vertikale Schichten (Abstraktion & Partitionierung)
- CNIs bieten exakt diese Abstraktionen

■ transparente Fehlertoleranz

- in EZ-Systemen basierend auf aktiver Redundanz & Mehrheitsentscheid
 - Replikation
 - Mehrheitsentscheid
 - (Re)integration ausgefallener Knoten
- in der Anwendung: hohe Komplexität nur durch Fehlertoleranz
- separate Fehlertoleranzschicht in TTA

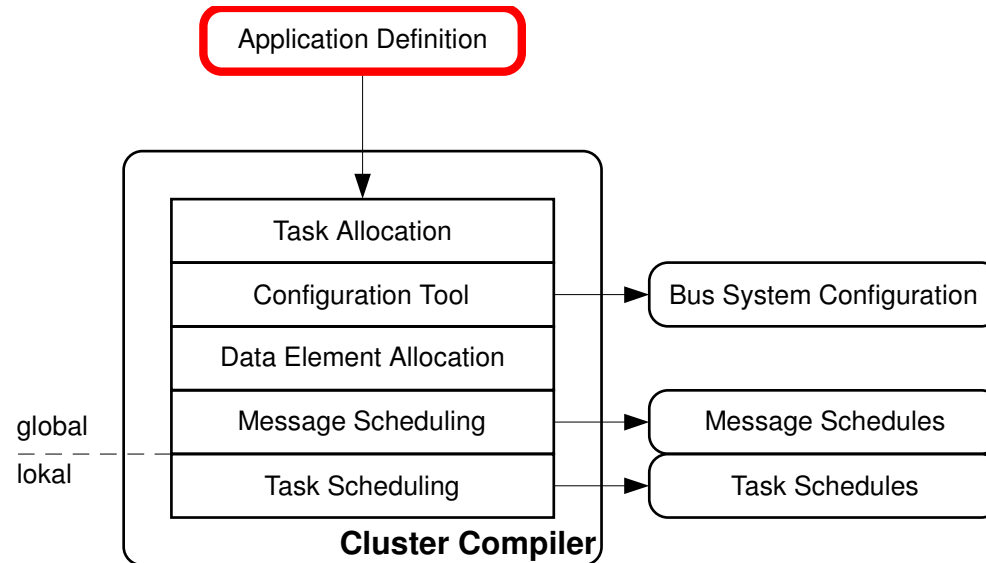


Cluster Compiler

- Entwicklungswerkzeug
 - zeitgesteuerte, verteilte Echtzeitsysteme
- statische Ablaufplanung für
 - Tasks und
 - Nachrichten
- verwendet Heuristiken
 - A^{*}-, IDA^{*}-Search
 - Tabu-Search
 - genetische Algorithmen
 - Simulated Annealing
- Grundprinzip: strikte Trennung von
 - globalen Aspekten und
 - lokalen Aspekten eines verteilten Echtzeitsystems



Cluster Compiler: Struktur

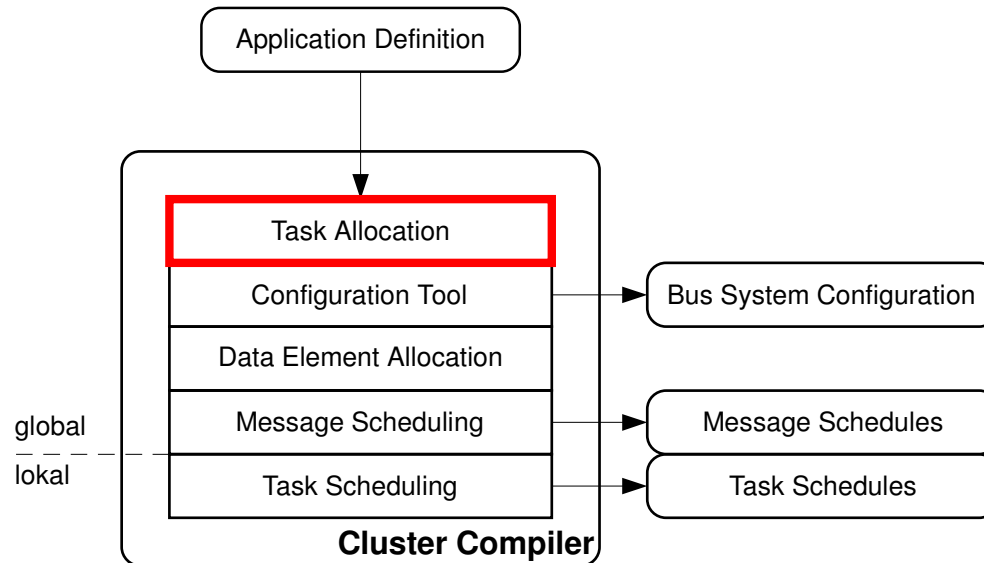


■ Eingabe

- Periode / WCET für alle Tasks
- Daten Elemente
 - Gültigkeitsintervalle, sender/empfangender Task
- Abhängigkeitsgraph
 - gegenseitiger Ausschluss, Datenabhängigkeiten, ...



Cluster Compiler: Struktur

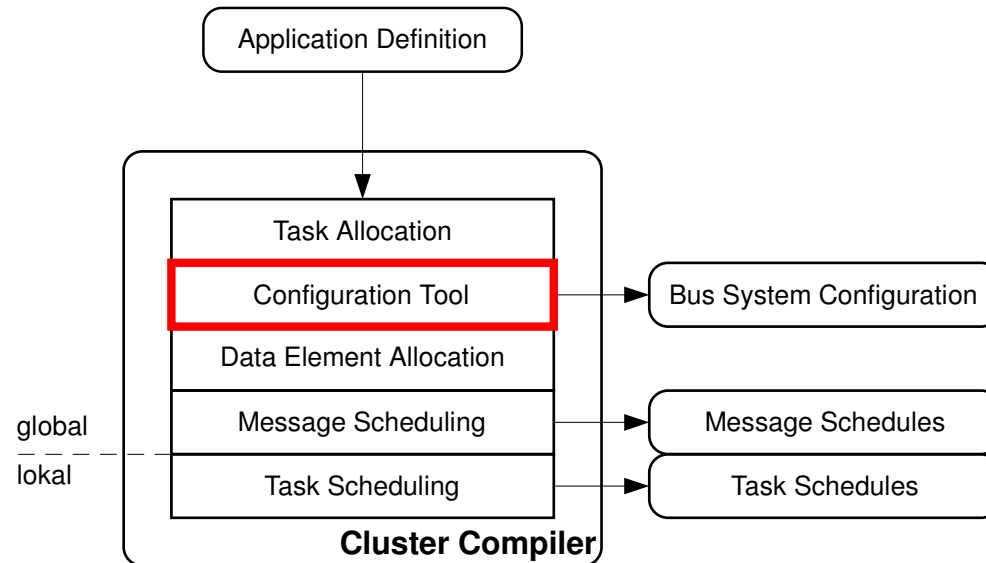


■ Zuordnung: Tasks → Knoten

- Optimierung
 - Lastausgleich
 - Minimierung der Kommunikation zwischen Knoten
- Einschränkungen
 - feste Zuordnung von Tasks zu bestimmten Knoten
 - Hardwareaffinität: Zugriff auf Peripherie



Cluster Compiler: Struktur

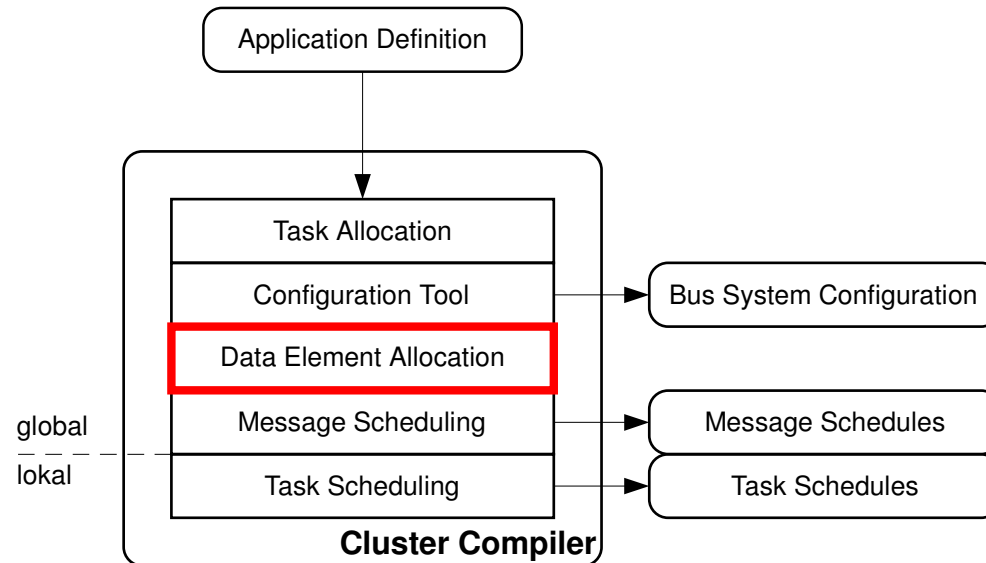


■ Konfiguration des Bussystems

- Optimierung
 - Übertragungslatenz
 - Anzahl Busse/Gateways
- Einschränkungen
 - *Locality Constraints*: vom Entwickler vorgegebene Zuordnung von Knoten zu bestimmten Bussen



Cluster Compiler: Struktur

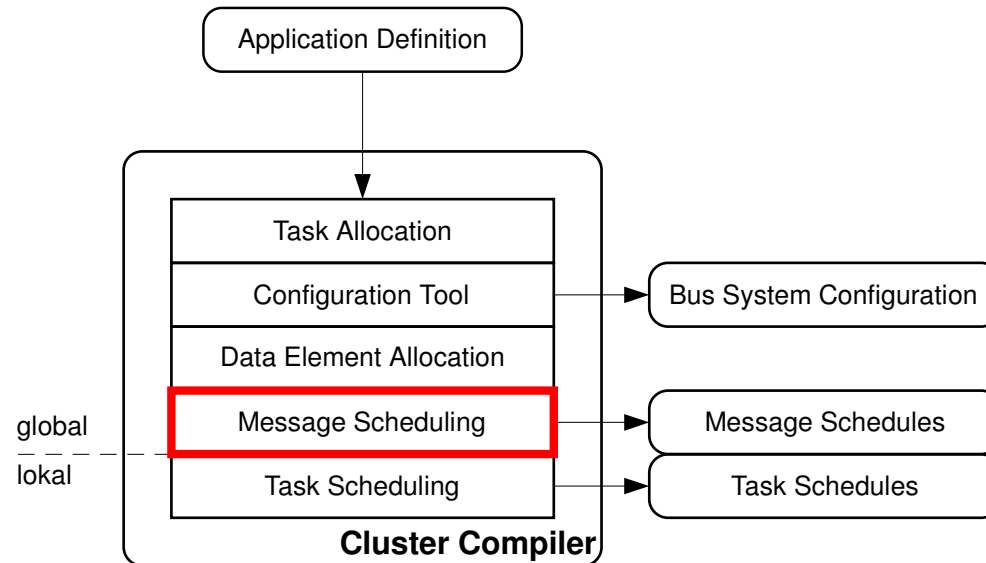


■ Abbildung von Datenelementen → TTP-Nachrichten

- Berechnung der Periode und der Länge der Nachrichten
- Einschränkungen
 - gleiche Reihenfolge von Sendern/Empfänger pro TDMA-Runde
 - Bandbreite des Mediums
 - Gültigkeitsintervalle der Datenelemente



Cluster Compiler: Struktur

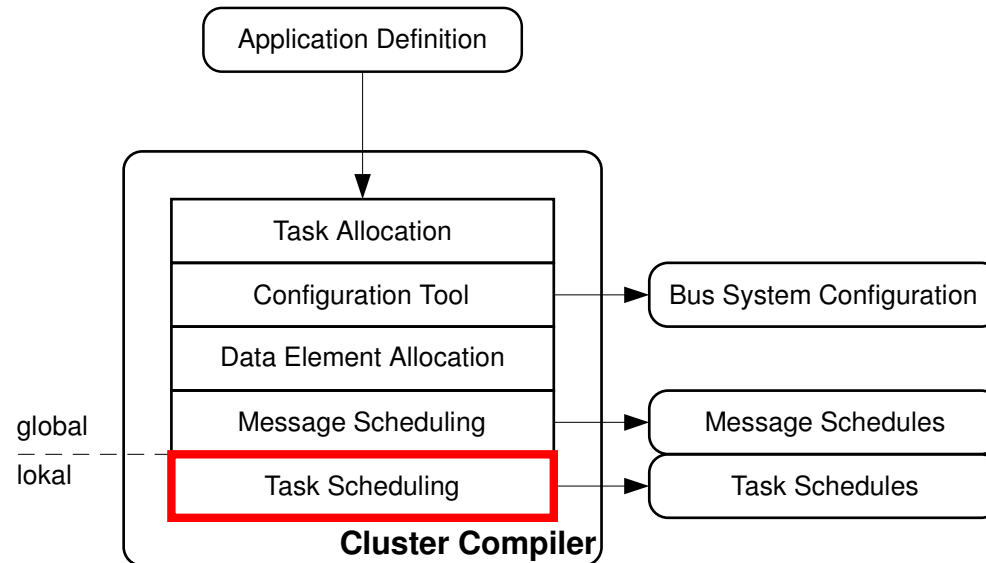


■ Sende- und Empfangstabellen

- fasse Nachrichten zu Cluster-Zyklen zusammen
- erzeuge Kontrollinformation zur Protokollausführung
- Optimierung
 - Übertragungslatenz
 - benötigte Bandbreite / Auslastung der Bandbreite



Cluster Compiler: Struktur



■ Ablauftabellen für lokale Knoten

■ Einschränkung

- Sende- und Empfangstabellen der Nachrichten
- Gültigkeitsintervalle der Datenelemente



Zusammenfassung

- Grundlagen
 - Ereignisbehandlungen
 - Abbildung auf Betriebssystemdienste
 - Berechnungen statischer Ablaufpläne
 - Planbarkeitsanalyse
- traditionelle Komposition
 - Probleme
- Komposition in der Forschung
 - Entkopplung und Automation der Einzelschritte
 - Beispiele: TTA & Cluster Compiler



Literatur

1. *Richard Nass.*
Annual study uncovers the embedded market.
Embedded Systems Design, 2007.
2. Hermann Kopetz and Günther Bauer.
The Time-Triggered Architecture.
Proceedings of the IEEE, 91(1):112-126, 2003.
3. Hermann Kopetz and Roman Nossal.
**The Cluster Compiler –
A tool for the design of time-triggered real-time systems.**
ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, pages 108-116, New York, NY, USA, 1995. ACM Press.

