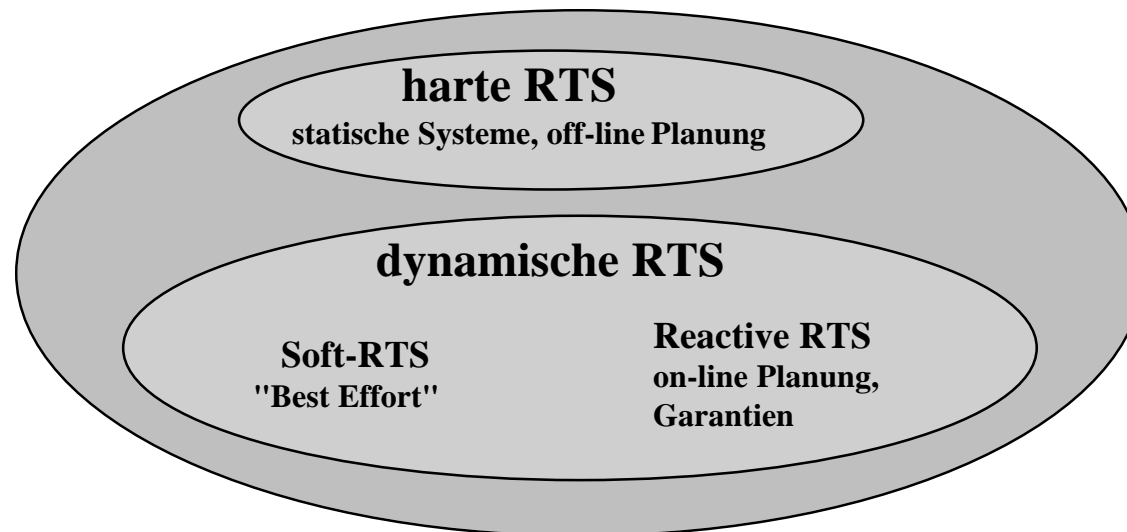


Maruti

Ziel: Bereitstellung einer Umgebung zur Entwicklung und Ausführung kritischer Anwendungen mit unterschiedlichen Anforderungen an die Einhaltung von Echtzeitbedingungen.



Maruti Steckbrief: Projektbeginn : 1988 an der University of Maryland,

Leitende Wissenschaftler: Ashok Agrawala, Satish Tripathi

1. Version: Maruti oberhalb Unix

2. Version: Maruti oberhalb Mach / OSF 1

3. Version: Maruti 3.0 Standalone auf 486/Pentium basierten Systemen, Entwicklungsumgebung NetBSD.

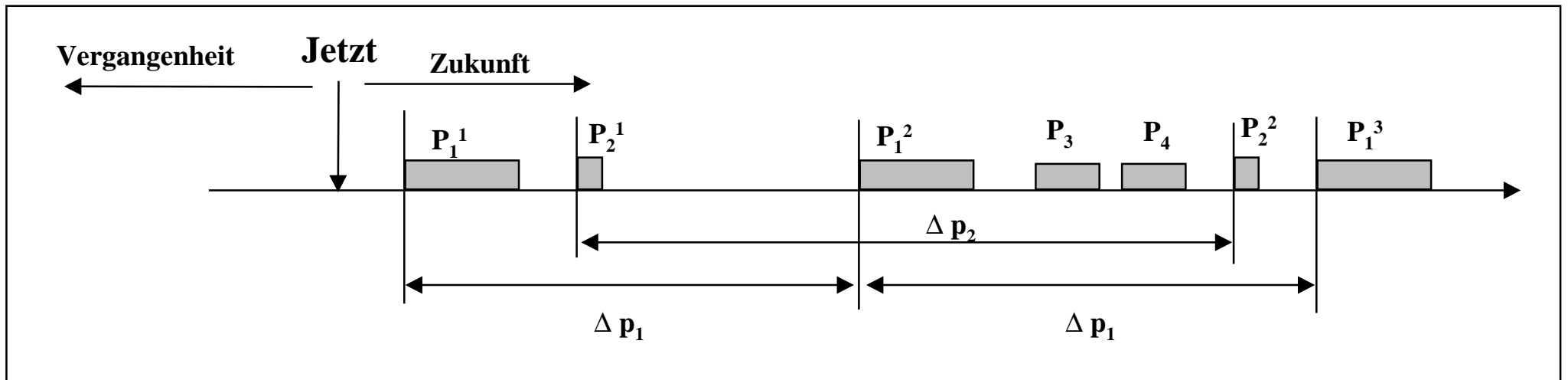
Maruti Entwurfsziele

- **Unterstützung unterschiedlich kritischer Anwendungen**
- **Fehlertoleranz unter Echtzeitbedingungen**
- **Verteiltheit ; Homogene und Heterogene Systeme; kooperierende Komponenten auf der Hard- und Softwareseite**
- **Mehrere Szenarien für Hard-RTS (z.B. denkbar: normaler Betrieb, Alarmbetrieb, Wartungsbetrieb eines RTS oder z.B. beim Space Shuttle : Startphase, Missionsphase, Landephase). Umschalten von einem zum anderen Szenario muß unter RT-Bedingungen sicher möglich sein.**
- **Integration von Anforderungen aus verschiedenen Bereichen, z.B. Fehlertoleranz und RT. Dadurch werden u.U. bestimmte Techniken der FT ausgeschlossen, z.B. Recovery-Blocks oder allgemein: Roll-Back Recovery Verfahren.**

Grundlegende Entwurfsprinzipien in Maruti (1)

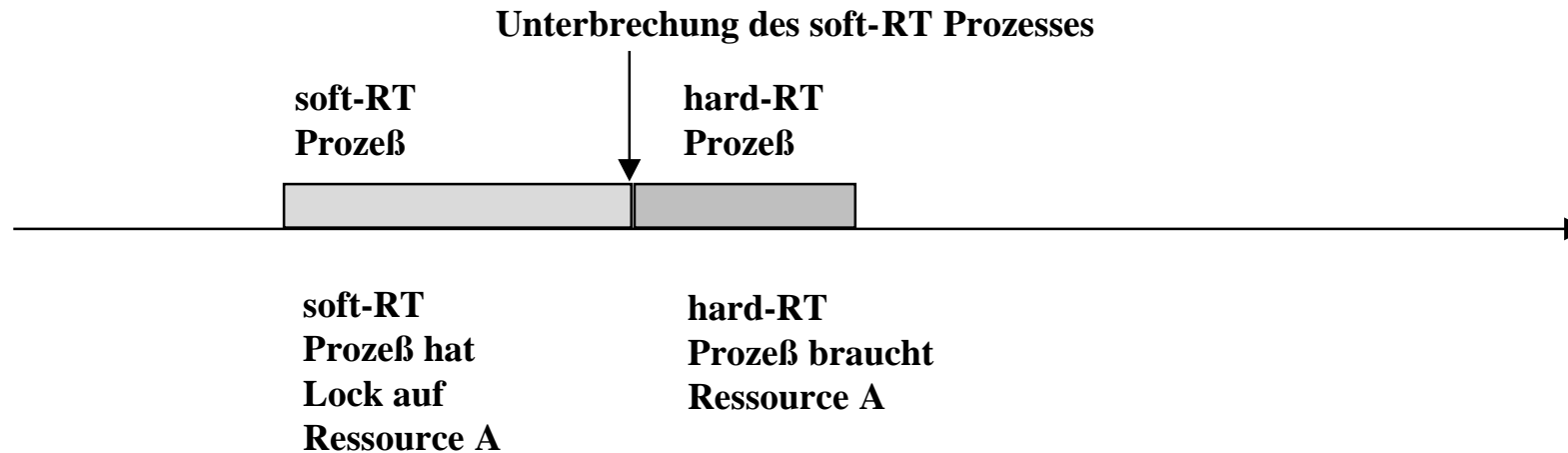
- **Ressourcen-Reservierung für Prozesse mit harten Zeitbedingungen**
→ **Kalender-basiertes Verfahren**

Kalender



Grundlegende Entwurfsprinzipien in Maruti (2)

- **Vorhersagbarkeit durch Verminderung von Ressource-Konflikten**



- **Portabilität und Erweiterbarkeit**

Anwendungsbereich von kleinen "embedded Systems" bis hin zu komplexen "mission critical" Systemen.

- Portabilität gewährleistet durch C, Standardkomponenten und minimale Annahmen bezgl. der Hardwareplattform.
- Erweiterbarkeit durch modulares Design und wohldefinierte Schnittstellen.
→ Objekt-basiertes System

Grundlegende Entwurfsprinzipien in Maruti (3)

- **Integrierte Unterstützung von Fehlertoleranz (FT)**
 - **frühe Fehlererkennung (Anwendungsspezifisch)**
generell trägt die objekt-basierte Systemkonstruktion hierzu bei.
 - **Redundanz (durch Replikation)**
(Zukunftsmusik; wird im Moment nur implizit unterstützt (vgl. getypte Komm. Kanäle, Struktur der "Elementaren Einheiten"))
 - **Möglichkeit, während der Laufzeit den Ausführungsmodus zu wechseln. (z.B. Zurücknahme von Garantien für Hard-RT Prozesse → Graceful Degradation)**
- **Trennung von Policy und Mechanism (Strategie und spezifische Verfahren)**

Beisp. : im Laufzeitsystem gibt es lediglich einen einfachen Dispatcher. Das Schedulingverfahren ist anwendungsspezifisch. Forderung: Die anwendungsspezifischen Strategien müssen leicht in das System integrierbar sein.

Grundlegende Entwurfsprinzipien in Maruti

- **Trennung von funktionalen und nicht-funktionalen Eigenschaften eines RT-Programms.**

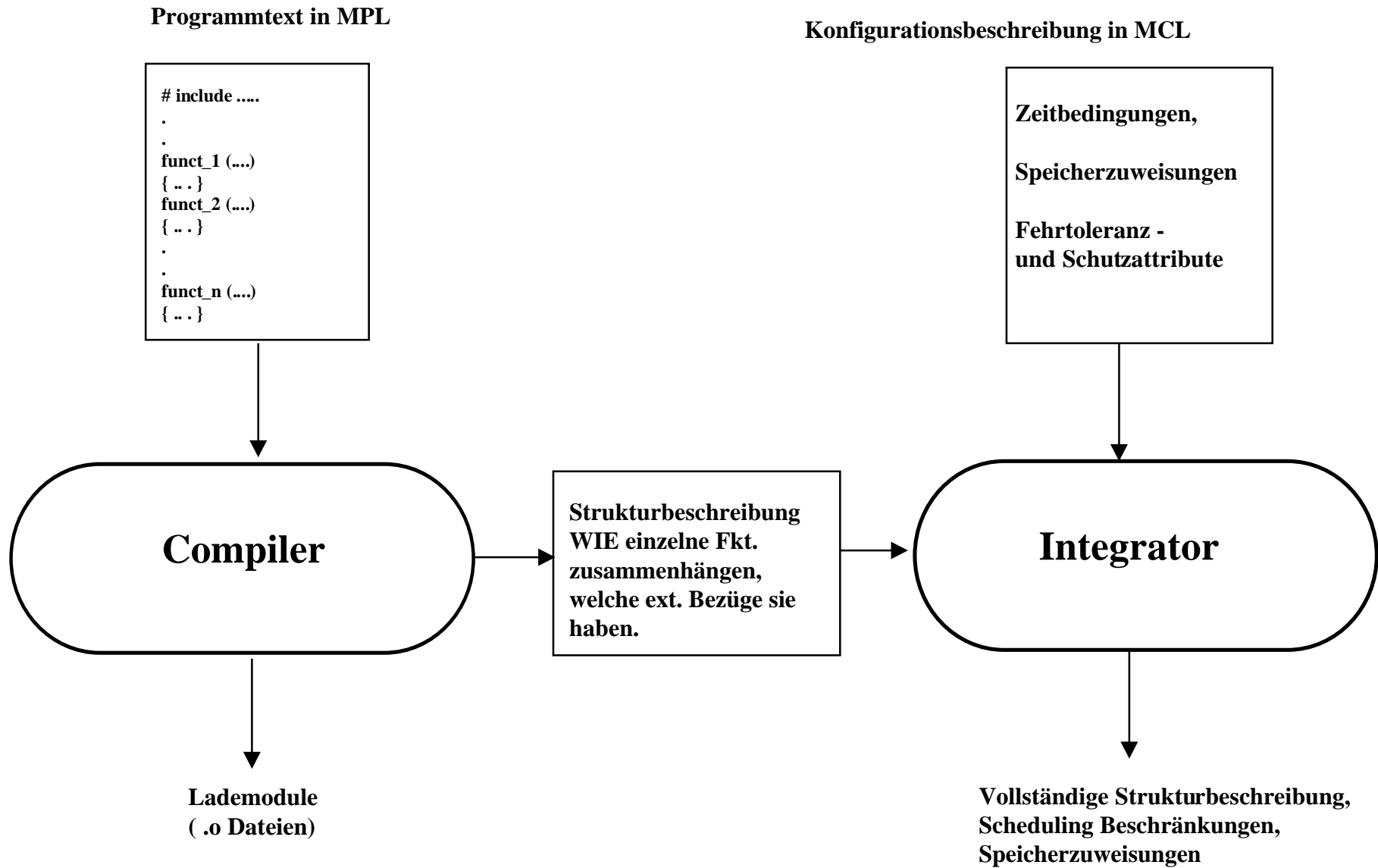
Funktionale Eigenschaften werden in MPL (Maruti Programming Language = ANSI-C + Erweiterungen) spezifiziert.

Einschränkungen: rekursive Aufrufe, unbeschränkte Schleifen

Nicht-funktionale Eigenschaften (z.B. Zeitbedingungen, Fehlerannahmen, Zugriffsschutz) werden in MCL (Maruti Configuration Language) spezifiziert.

MCL ist eine interpretierte C-ähnliche Spezifikationsprache.

Zusammenspiel zwischen Funktionaler und Konfigurations- Beschreibung



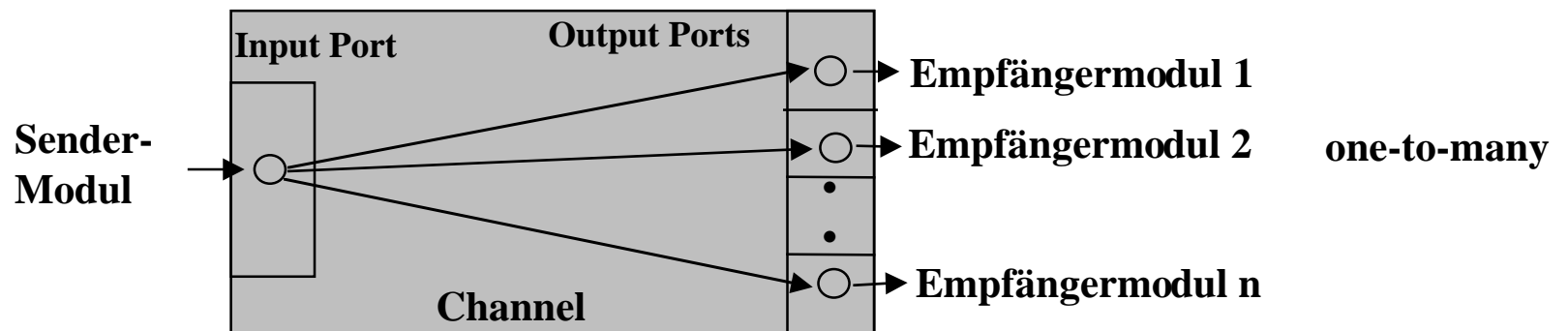
Besonderheiten der Maruti Programmiersprache MPL

Module: unabhängig compilierbare Einheit, die aus einer Menge von Prozeduren, Funktionen und lokalen Datenstrukturen besteht. Optional hat jedes Modul eine Initialisierungsfunktion, die aufgerufen wird, wenn das Modul in den Speicher geladen wird. Die Initialisierungsfunktion kann mit eigenen Parametern aufgerufen werden. Module werden mit anderen Modulen zu vollständigen Anwendungen verbunden.

Channels: Channels spezifizieren die Kommunikationsverbindungen..

Channels sind Punkt-zu-Punkt-Verbindungen. Sie haben eine Richtung (one-way-Channels). Channels sind typisiert. Es gibt mehrere Möglichkeiten, die Charakteristika eines Channels zu spezifizieren, z.B. Puffergröße, was nach einem Pufferüberlauf geschehen soll oder was passiert, wenn keine Nachricht im Channel ist. Darüberhinaus kann synchrone und asynchrone Nachrichtenübertragung spezifiziert werden.

Die Verbindung der Endpunkte eines Channels wird in der MCL-Spezifikation für die jeweilige Anwendung durchgeführt. Dabei wird eine Typprüfung durchgeführt, die gewährleistet, daß Endpunkte vom selben Typ sind und zur Laufzeit richtig verbunden werden.



Besonderheiten der Maruti Programmiersprache MPL (cont.)

Entry points: Periodische Funktionen spezifizieren sogenannte "Entry Points" in einer Anwendung, an denen die Ausführung der entsprechenden Funktion implizit (durch entsprechende Zeitwerte) gestartet wird. MCL spezifiziert, WANN die Ausführung gestartet wird, d.h. die Periode.

Services: Funktionen, die explizit durch Nachrichten aufgerufen werden. Services werden ausgeführt, wenn die entsprechende "request" Nachricht empfangen wurde.

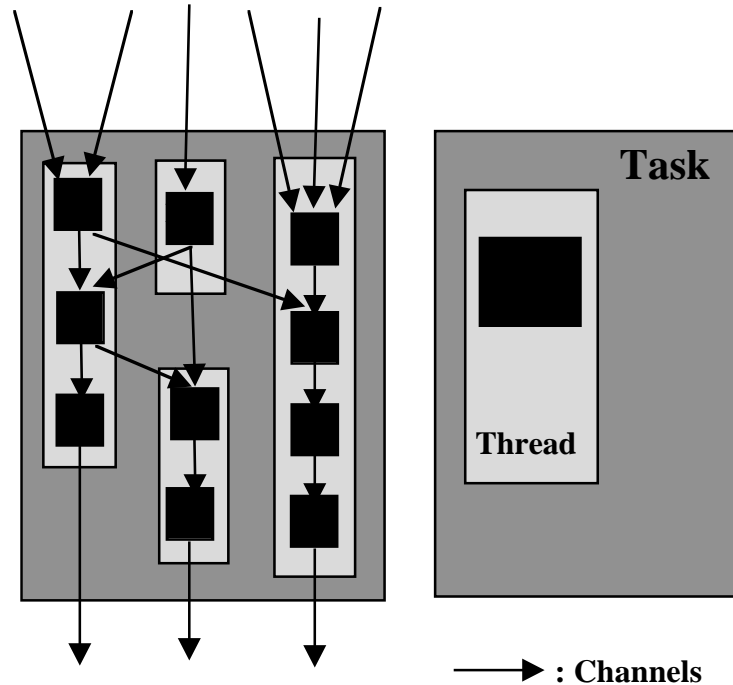
Shared memory blocks: werden innerhalb von Moduln deklariert. Sie bieten die Möglichkeit, daß Moduln, die nicht in einem Adreßraum ausgeführt werden, Daten gemeinsam nutzen können. Wie und von welchen Moduln shared memory blocks zugreifbar sind, wird durch die MCL festgelegt.

Actions: Definieren eine sequentielle Codefolge, die eine extern beobachtbare Aktion des Moduls beschreiben. Actions werden benötigt, um Zeitbeschränkungen in MCL zu spezifizieren.

Critical regions: werden benutzt, um sicher auf Daten zugreifen zu können ohne daß die Konsistenz der Daten gefährdet wird. Maruit garantiert, daß zwei Einheiten nie so geplant werden, daß sie sich gleichzeitig in ihren kritischen Regionen befinden.

MCL

MCL beschreibt die dynamische Ausführungsumgebung, d.h. die Konfiguration der funktionalen Module.



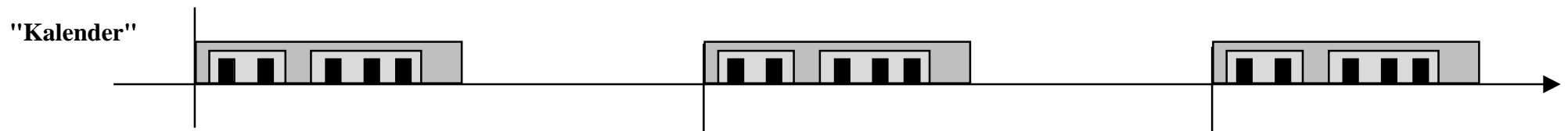
EU: Elementareinheit (Elemental Unit)
Grundlegender Baustein einer Berechnung in Maruti.

Task: Adreßraum, in dem mehrere Threads nebenläufig ausgeführt werden.

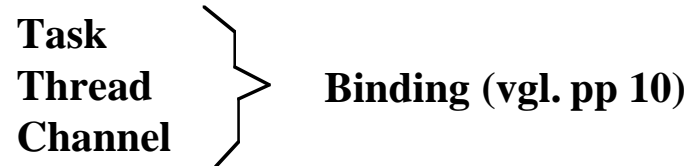
Thread: sequentielle Komposition von EUs. Ein Thread hat einen sequentiellen Kontrollfluß, der durch eine Nachricht an die erste EU im Thread getriggert wird. Der Kontrollfluß wird mit der letzten EU im Thread beendet. Alle EUs eines Threads haben gemeinsam einen Execution Stack und einen Prozessor Status.

Job: Eine Kollektion von Threads, die zur Erbringung einer Aufgabe miteinander kooperieren. Alle Threads eines Jobs unterliegen der globalen Zeitbeschränkung eines Jobs.

Beisp.: Globale und "lokale" Zeitbeschränkungen



MCL



Resources : Alle globalen Ressourcen, d.h. Ressourcen, die außerhalb eines Moduls sichtbar sind, werden in der Konfigurationsdatei spezifiziert, zusammen mit den Zugriffsbeschränkungen. MCL erlaubt die Bindung von Ressourcen in einem Modul an globale Ressourcen. Jede Ressource, die nicht auf eine globale Ressource abgebildet wird, ist Modul-lokal.

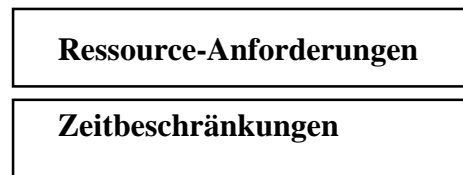
Zeitanforderungen: Eine Anwendung besteht aus einer Menge kooperierender Jobs. Ein Job ist eine Menge von und
Zeitbeschränkungen Entries - d.h. eng kooperierenden Services (statisch) , die über die Entries aufgerufen werden und (dynamisch) einer Menge von Tasks und Threads, die die Services ausführen. Mit dem Job wird seine Aufrufcharakteristik beschrieben, d.h. periodisch oder aperiodisch. Es werden die Periode und optional die Bereitzeit und Frist innerhalb der Periode angegeben. Diese Beschränkungen gelten für alle Komponenten des Jobs, d.h. für seine Threads.

Weitere Möglichkeiten der Zeitspezifikation: Für jede beobachtbare Aktion können, relativ zu den Zeiten des Jobs, Bereitzeiten und Fristen spezifiziert werden. Außerdem können relative Beschränkungen spezifiziert werden, die sich auf das Intervall zwischen zwei Ereignissen beziehen, z.B. die minimale oder maximale Zeitdifferenz. (Ereignisbasierte Modellierung)

Fehlertoleranz : MCL untertützt anwendungsspezifische FT durch Vereinfachung der Replikation und Allokation replizierter Komponenten.

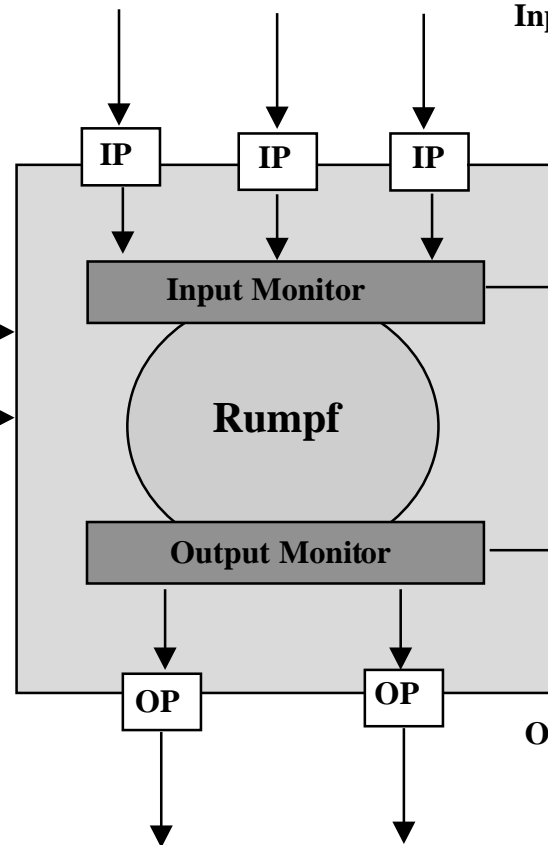
Elementareinheiten (EU)

Input Monitor: Der I-Monitor nimmt Daten von den IPs, führt Filter - und Eingangsprüfungen durch und stellt die Daten dem Rumpf zur Verfügung. Außerdem können durch den I-Monitor Trigger-Bedingungen für die EU überprüft werden.



Rumpf: Der Rumpf einer EU definiert ihre Funktionalität. Er akzeptiert Eingabedaten vom I-Monitor und versorgt den O-Monitor mit Ausgabedaten.

Output-Monitor: Analog zum I-Monitor stellt der O-Monitor einen Filter für die Ausgabedaten dar. Der O-Monitor kann zur Fehlererkennung und zur Durchsetzung von Zeitbedingungen genutzt werden.



Input Ports : Jeder I-Port spezifiziert einen Teil der Eingangs-Schnittstelle einer EU. I-Ports sind mit dem Output-Port eines Channels verbunden, d.h. Endpunkte von Channels.

Status / Error

(werden mit entsprechenden EUs zur Behandlung verbunden, z.B. Exception-Handlers)

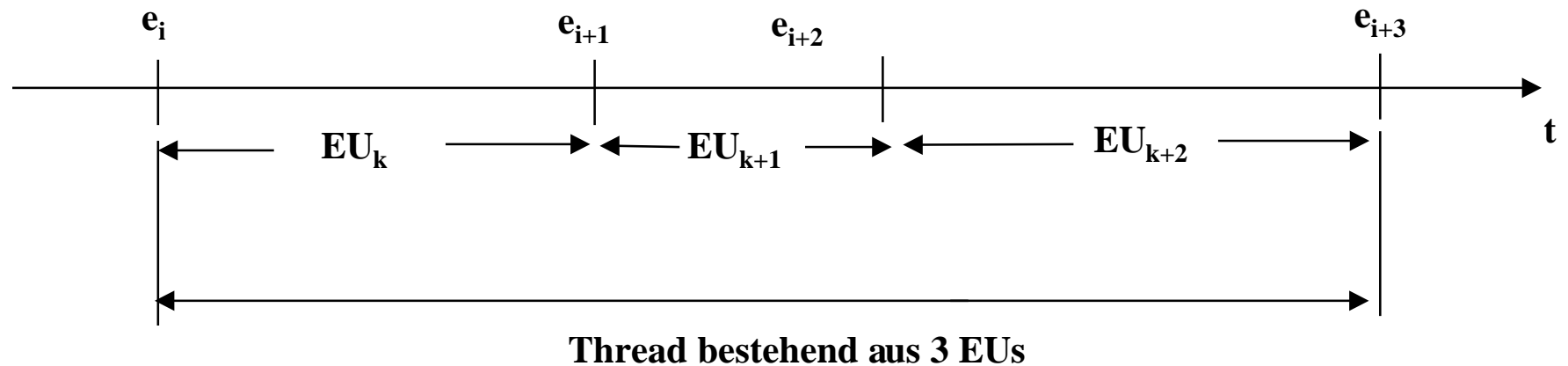
Status / Error

Output Ports: Jeder O-Port spezifiziert einen Teil der Ausgangs-Schnittstelle der EU. O-Ports sind mit Input-Ports von Channels verbunden, über die die Verbindung zu anderen EUs hergestellt wird.

Das Verhalten einer EU ist atomar im Hinblick auf ihre Interaktion mit der Umgebung:

- Alle benötigten Ressourcen, die eine EU benötigt, stehen ihr während der gesamten Ausführungszeit zur Verfügung
- Die Interaktion mit der Umgebung einer EU erfolgt nur vor ihrem Start oder nachdem ihre Ausführung beendet ist.

Aufgrund der Atomaritätseigenschaft lassen sich EUs zeitlich mit einem Ereignisbasierten Modell beschreiben.



Programmanalyse

Ziel: Automatische Identifizierung von EUs aus dem Programmcode
Erstellen eines Kontrollflußgraphen
Erstellen eine partiellen EU-Graphen

Annahmen: Alle Input-Nachrichten werden *logisch* beim Start einer EU empfangen.
Alle Output-Nachrichten werden *logisch* am Ende einer EU gesendet.
(Atomaritätseigenschaften legen dies nahe !)

EUs werden so *geschnitten*, daß keine Zyklen entstehen. Z.B. ein *send* dem ein *receive* folgt, kann zu einer zyklischen Vorrangrelation führen und muß deshalb verhindert werden.

Vorgehensweise: EU-Grenzen werden erzeugt bei:

- einem *receive*-Statement,
- dem Anfang oder Ende eines *Resource-Blocks*,
- dem Anfang oder Ende einer *beobachtbaren Aktion*.

Datenstruktur : Für jede EU wird folgende Information erzeugt und gespeichert:

- ein symbolischer Name mit dem sie identifiziert wird,
- die Vorgänger-EU einer EU,
- die Nachfolger-EU einer EU,
- die Zeilennummern im Source-Code des entsprechenden Moduls,
- Platzhalter für fehlende Informationen

!! Zeitanalysen werden in dieser Phase NICHT durchgeführt !!

Kommunikationsmodell

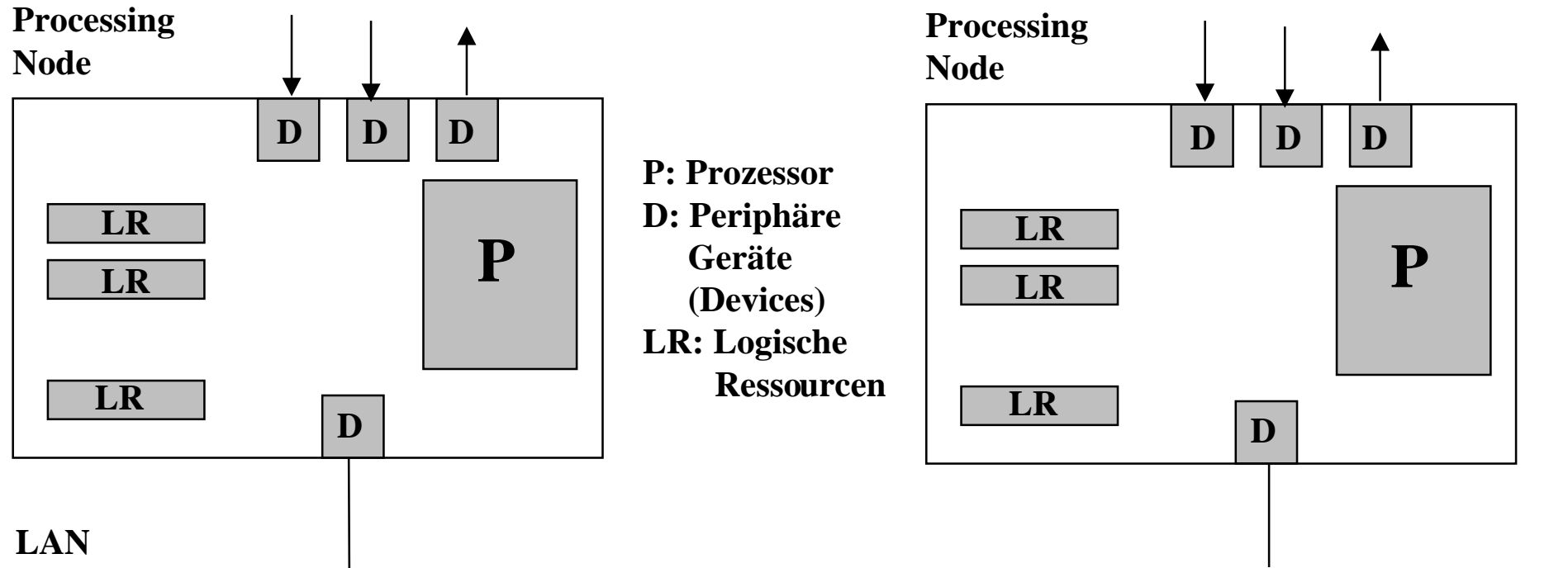
- **Punkt-zu-Punkt Nachrichtenübertragung zwischen zwei EUs.
Abstraktion: Channel - Ein Channel wird durch die Deklaration eines Output-Ports beim Sender und eines Input-Ports beim Empfänger eingerichtet. Der Sender blockiert nicht nach einem Sendeereignis (asynchron).**
- **Multicast Nachrichten können durch die Angabe mehrerer Empfänger realisiert werden.**
- **Synchrone Kommunikation wird zwischen eng gekoppelten EUs eines Jobs benutzt. Für jeden Aufruf des Senders wird auch der Empfänger aufgerufen. Der Sender ist blockiert (de-scheduled), bis die Rückantwort eingetroffen ist. Nachrichten zwischen Sender und Empfänger werden in FIFO - Ordnung ausgeliefert.**
- **Asynchrone Kommunikation wird zwischen EUs unterschiedlicher Jobs oder zwischen RT und nicht-RT Jobs angewendet. Da Sender und Empfänger asynchron arbeiten, kann es sein, daß der reservierte Puffer überläuft, so daß Nachrichten verloren gehen können.
Auf der Empfängerseite muß spezifiziert werden, welche Nachricht beim Pufferüberlauf ignoriert werden soll:
inFirst: die älteren (ersten) Nachrichten im Puffer werden erhalten, die neueste Nachricht wird überschrieben
inLast: die neuesten Nachrichten im Puffer werden erhalten, die älteste Nachricht wird überschrieben (Ringpuffer)**

Kommunikationsmodell (cont.)

Shared Memory

- **EUs existieren in demselben Adreßraum (task)**
- **Shared Memory Buffer: Partition des Speichers, auf den mehrere EUs zugreifen dürfen. Schutz ist benutzerspezifisch. Zugriffsbeschränkungen durch Definition der Partition als "logische Ressource" (Shared memory blocks)**

Maruti Ressource Modell



Einschränkungen bei der Nutzung der Ressourcen:

- **Nicht unterbrechbar (non-preemptable):** Viele Geräte benötigen diese Einschränkung zum konsistenten Betrieb. Das Programmstück, das die entsprechende Ressource steuert, muß vollständig ausgeführt werden, d.h. CPU darf während der Ausführung nicht unterbrochen werden.
- **Exklusiv (exclusive):** Die Ressource ist unterbrechbar, darf aber während der Unterbrechung nicht anderweitig benutzt werden. Eine kritische Region ist ein Beispiel für diesen Ressource-Typ.
- **Seriell wiederbenutzbar (serially reusable):** keine besonderen Einschränkungen.
- **Shared:** Eine "shared resource" darf simultan von mehreren Einheiten benutzt werden. In einem 1-Prozessor-System ist kein Unterschied zwischen "seriell wiederbenutzbar" und "shared".

Maruti Ressource Modell

Ebenen, auf denen die Ressource-Anforderungen spezifiziert werden können:

- **EU-Ebene**
Wegen der **Atomaritätseigenschaft** der EU impliziert die Spezifikation einer Ressource auf dieser Ebene, daß sie innerhalb der EU angefordert und freigegeben wird. Für die Planung wird angenommen, daß die Ressource während der gesamten Ausführungsdauer der EU benötigt wird.
- **Thread-Ebene**
Ressourcespezifikation für verschiedene EUs eines Threads. Eine kritische Region kann z.B. von einer EU angefordert und von einer anderen EU freigegeben werden.
- **Job-Ebene**
Ressourcen werden für den gesamten Job spezifiziert. Sie werden nicht bei jedem Aufruf eines periodischen oder sporadischen Jobs, sondern bei der Initialisierung und der Beendigung angefordert und freigegeben.



Operationale Beschränkungen

- **Synchronisation**
Diese Beschränkungen beziehen sich auf Daten- und Kontrollabhängigkeiten.
Beisp.: Vorrang und wechselseitiger Ausschluß.
- **Timing**
Job-Ebene: sporadisch, periodisch, aperiodisch, Bereitzeit, Frist
Thread-Ebene: Bereitzeit und Frist relativ zu Job
EU-Ebene: Bereitzeit, Frist (wie bei Thread), Intervalle zwischen zwei EUs
- **Speicherzuteilung: Tasks werden Rechnerknoten (Processing Nodes) zugeteilt.**
(Allocation) Beschränkungen der Speicherzuteilung werden genutzt, um diese Zuteilung zu beeinflussen.
Beisp.:
 - Eine Task benutzt ein peripheres Gerät, das nur an einem bestimmten Rechnerknoten verfügbar ist.
 - Bei Replikation sollen Replikas auf unterschiedlichen Rechnerknoten liegen.
 - Bei Memory Sharing sollen Tasks auf demselben Rechnerknoten liegen.

Scheduling und Speicherzuteilung

- **Zeitbasiertes Scheduling mit Kalender**
- **Statisches Scheduling und Speicherzuteilung**
- **Planung der Speicherzuteilung erfolgt auf der Ebenen von Tasks**
- **Scheduling erfolgt auf der Ebene der EUs**
- **Betrachtete Länge des Kalenders: Kleinste gemeinsame Vielfache (lcm) aller Task-Perioden.**
- **Schedulingmethoden: Zeitanalyse von Ausführungen, Simulated Annealing**

Maruti Laufzeitumgebung

Maruti Kern:

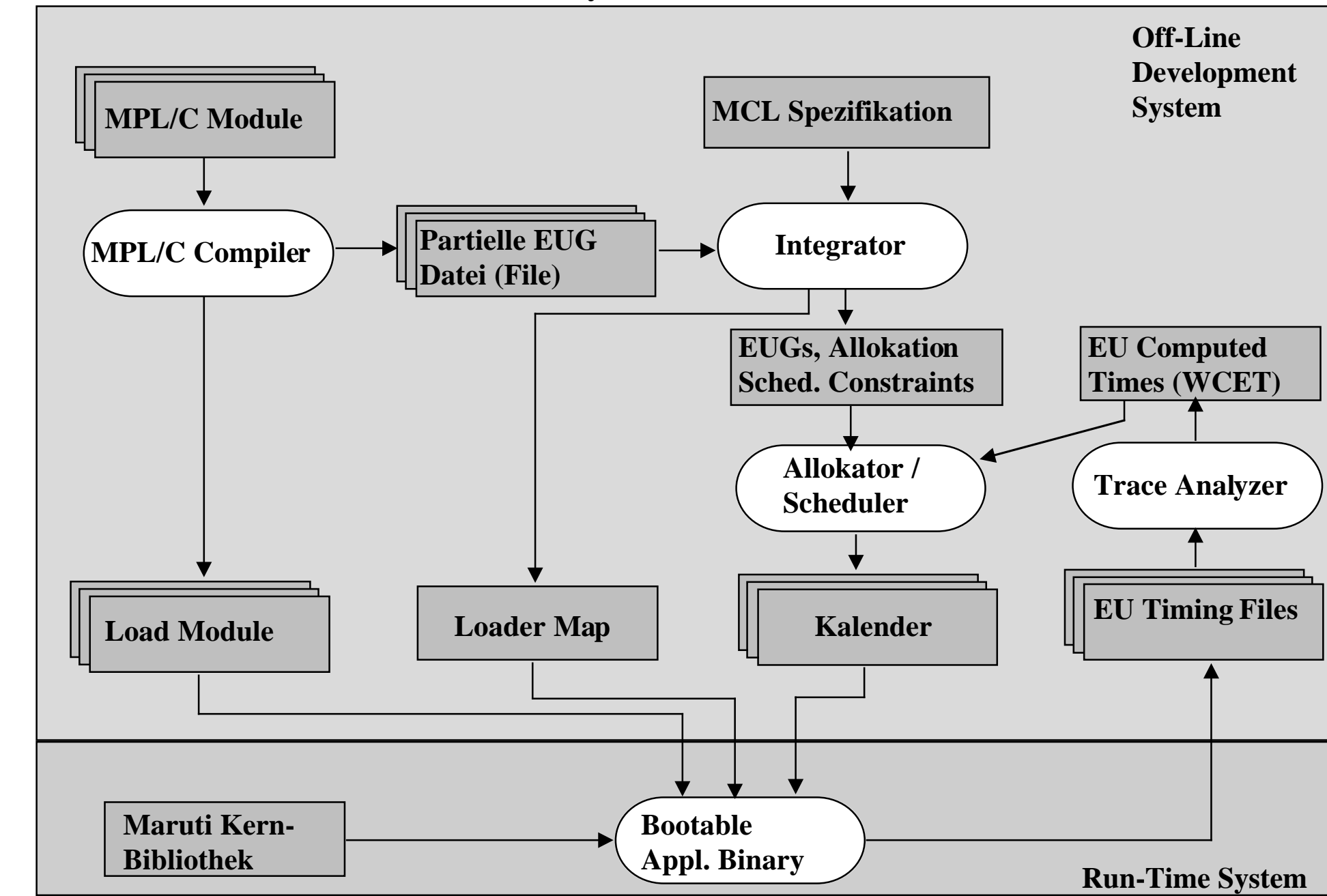
Aufgaben: Prozessorzuteilung (Scheduling), Nachrichtenübertragung, Thread-Kontrolle, "low level" Kontrolle der Hardware.

Dispatcher:

Aufgaben:

- **Ressource Management** - Der Dispatcher lädt Anwendungen und generiert dabei die notwendigen Task -und Thread-Strukturen, reserviert Speicher und lädt den Code der Anwendung. Ist eine Anwendung erfolgreich geladen, sind alle notwendigen Ressourcen reserviert.
- **Kalender Management** - Der Dispatcher erzeugt und lädt die Kalender für eine Anwendung. Wird die Anwendung gestartet, aktiviert der Dispatcher den entspr. Kalender.
- **Verbindungs-Management** - Zur Laufzeit baut der Dispatcher die Verbindungen zwischen Tasks auf, die über Channels miteinander kommunizieren. Dabei werden für lokale (auf demselben Rechnerknoten) Verbindungen "Shared Buffers" benutzt, für Verbindungen über Knoten hinweg werden Kommunikationsagenten + Shared Buffers benutzt.
- **Ausnahmebehandlung** - verpaßte Fristen, Stack-Überlauf, arithmetische Exceptions, Zugriff auf nicht reservierten Speicher, etc.
Ausnahmebehandlung ist benutzerdefiniert und reicht von Abbruch der gesamten Anwendung, Abbruch des fehlerhaften Threads bis zum einfachen Aufruf einer Ausnahmebehandlungsroutine.

Maruti Systemarchitektur



Maruti Tools

- **MPL/C Compiler**
modifizierter gcc erzeugt Lademodule (*modulx.o* files) und eine *modulx.eul* Datei, die den partiellen EUG des Moduls enthält.
- **MCL Integrator**
liest die Konfigurationsdatei der Anwendung *app-name.cfg* und alle dazugehörigen *modulx.eul* Dateien. Der Integrator erzeugt und überprüft alle Jobs, Tasks, Threads, und alle Verbindungen der Anwendung. Seine Ausgabedateien sind:
 - eine “loader map” Datei *app-name.ldf*, die beim Laden der Anwendung ausgewertet wird.
 - einen vollständigen EUG, der mit Speicherzuweisungs- und Schedulingbeschränkungen annotiert ist *app-name.sch*.
- **Allocator/Scheduler**
Unter Berücksichtigung aller Beschränkungen und einiger Systemparameter versucht der Allocator/Scheduler:
 1. eine gültige Speicherzuweisung für die Tasks über alle Rechnerknoten eines verteilten Systems zu finden.
 2. einen gültigen Schedule für jeden Rechnerknoten

Der Allocator/Scheduler beendet seine Aufgabe, wenn entweder ein gültiger Plan gefunden ist, oder er die Unmöglichkeit für einen Plan feststellt.

Maruti Tools (cont.)

Der Allocator/Scheduler erzeugt eine Datei *app-name.alloc*, die Speicherzuweisungsinformation enthält und eine Datei *app-name.cal*, die Planungsinformation für Kalender enthält.

Die Anzahl der Rechnerknoten und die Länge der TDMA-Slots können dem Allocator/Scheduler als Planungsparameter angegeben werden.

- **Binder (mbind)**
Der Binder erzeugt die (statischen) Datenstrukturen, die zur Laufzeit benötigt werden. Dazu wertet er aus:
 - Die loader-map-Datei: *app-name.ldf*
 - Die Speicherzuweisungsdatei: *app-name.alloc*
 - Die Kalender-Datei: *app-name.cal*Daraus erzeugt er die Datei: *app-name.-globals.c* und einen “makefile” *app-name.-bind.mk*.
- **Timing Monitor**
Der Timing Monitor läuft auf einem von der Anwendung getrennten Rechnerknoten. Über eine besondere Schnittstelle (im Moment serielle Ltg.) erhält er Timing Daten von der Anwendung, die er zur nachträglichen (off-line) Analyse durch den Timing Trace Analyzer aufbereitet.
- **Timing Trace Analyzer (timestat)**
Analysiert die vom Timing Monitor erstellten Dateien und erzeugt eine Datei *app-name.-wcet*, die die worst-case-execution-times für den Allocator enthält.

Kalender

calendar.h

```
typedef struct { int s, us; } utime_t;
```

```
typedef struct {  
    int thread;  
    utime_t start, end;  
} uentry_t;
```

```
typedef struct {  
    utime_t lcm;  
    int nentries;  
    uentry_t *entries;  
} ucalendar_t;
```

einzelner Eintrag

thread #	Startzeit	Abschlußzeit
----------	-----------	--------------

Kalender

Planungs- intervall	Anzahl der Einträge	Pointer auf Entry
------------------------	------------------------	----------------------

