10 Echtzeitsprachen

10.1 Bemerkungen zur Historie

- 1960–1970
  - ⋆ Assemblerprogramme, da Speicher teuer
  - ⋆ frühe Ansätze zur Vereinheitlichung in Kernphysik: CAMAC–Geräteschnittstelle
  - ⋆ in den Progammen wurde "optimiert" ⇒ jedes Bit genutzt
- ab ca. 1966
  - ⋆ erster Einsatz von höheren Sprachen, z.B.
    - ◦ CORAL und RTL/2 (England)
    - ◦ ALGOL 60
    - ◦ FORTRAN IV (meistens)
  - ⋆ Prozeduraufrufe für Betriebssystem-Echtzeit-Dienste
  - ⋆ Probleme
    - ◦ viel Wissen über BS nötig
    - ◦ wenig portabel
    - ◦ keine semantische Prüfung der Parameter durch den Übersetzer, da keine speziellen Datentypen für z.B. task, semaphor, clock
    - ◦ schwierige Fehlersuche

- ab 1970: erste Echtzeitsprachen
  - ⋆ nationale und internationale Normungen
    - ◦ PEARL (Deutschland)
      (Process and Experiment Automation Realtime Language)
    - ◦ HAL/S (USA)
    - ◦ PROCOL (Japan)
    - ◦ RT-FORTRAN
    - ◦ RT-BASIC
  - ⋆ Datentypen z.B.
    - ◦ task
    - ◦ clock
    - ◦ duration
    - ◦ sema
    - ◦ interrupt

    mit den zugehörigen Operationen in die Sprache integriert
  - ⋆ einheitliche Anweisungen für E/A und die Beschreibung von Datenwegen
  - ⋆ Vorteile
    - ◦ benutzerfreundliche Sprachelemente

- Semantik der Paramter bei BS–Aufrufen vom Übersetzer/Binder prüfbar
- weitgehend portabel

⋆ Nachteile

- eigenes umfangreiches BS nötig (Speicher, BS–Overhead, Software–Kosten,)
- vorhandenes universelles BS wird angepaßt → Gefahren: ineffizient, überflüssige Teile im BS, Supersets der Sprache gefährden Portabilität

• ab 1978 universelle Echtzeitsprachen

⋆ universelle sichere hohe Sprachkonzepte für alle Anwendungsbereiche

⋆ Standardisierung, insbesondere DoD (ADA)

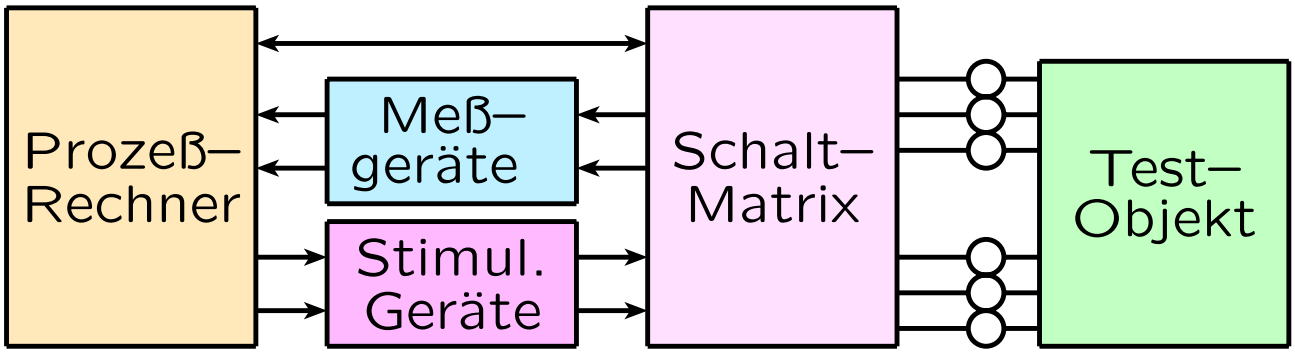
⋆ Datentypen (z.B. task, duration, interrupt) oder systemabhängige Parameter in sprachlich sauberer Weise mittels Moduln / packages eingebunden

⋆ Beispiele

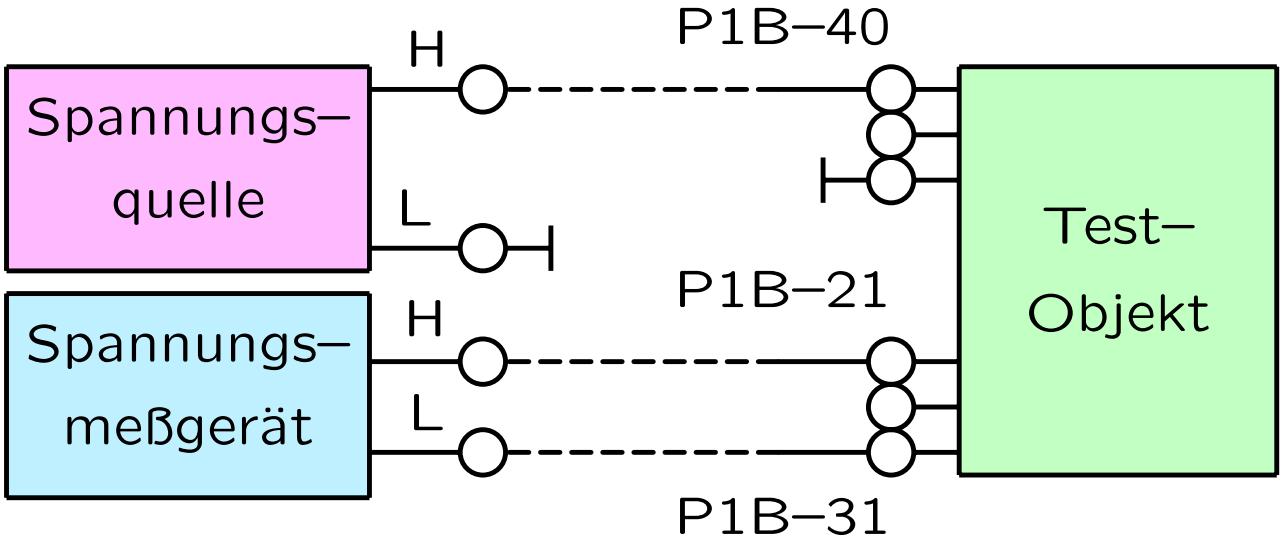
- ADA83, ADA95
- CHILL
- MODULA–2
- PEARL, PEARL 90, Mehrrechner–PEARL

- heute
  - ⋆ eher Tendenz statt spezieller Echtzeitsprache, eine universelle Sprache (z.B. C, C++ oder JAVA) mit Bibliotheksprozeduren für Realzeitdienste anzureichern
  - ⋆ (herstellerspezifische) Speziallösungen für eingeschränkte Anwendungsbereiche, z.B.
    - ◦ Prüfwesen
    - ◦ Standard–Regelungsaufgaben
    - ◦ Fördertechnik, Chargenprozesse
    - ◦ Visualisierung (Leitstand)
    - ◦ Telefonanlagen
  - ⋆ meist als Progammbibliotheken
  - ⋆ Beispiele
    - ◦ SPS-Programmierung
    - ◦ ATLAS (Abbreviated Test Language for All Systems) für Prüfsysteme, insbesondere Flugzeug–Elektronik
    - ◦ ESTEREL und andere synchrone Sprachen (Automaten)
    - ◦ Sprachen zur Spezifikation von einfachen Echtzeitsystemen, meist grafisch

- Anwendungsbeispiel Atlas
  - ⋆ Blockschaltbild Testsystem



  - ⋆ Prüfung Elektronik–Gerät



    - ◦ Verbale Testvorschrift

Lege 28V = an P1B–40

OK, falls 30mV zw. P1B-21 und P1B-31

- ATLAS–Programmausschnitt

```
100 302  APPLY, SOURCE, DC SIGNAL,
         VOLTAGE 28 ERRLMT + - 0.5V,
         CNX HI P1B-40 LO COMMON $
100 304  VERIFY, (VOLTAGE), DC SIGNAL,
         NOM 0.000V LL-0.030V UL 0.030V,
         CNX HI P1B-21 LO P1B-31 $
100 306  GOTO, STEP 105 084, IF NOGO $
         ...
105 084  PRINT, MESSAGE, 'FEHLER' $
```

10.2 Anforderungen an Echtzeitsprachen

- effiziente Erstellung großer komplexer Systeme mit hohen Sicherheitsanforderungen (z.B. in Luft– und Raumfahrt, Verkehr, Medizin)
- unabhängiges Implementieren, Übersetzen, Testen von Moduln durch verschiedene Personen
- Daten- und Methodenkapselung in Moduln
- Lesbarkeit, Übersichtlichkeit, Einfachheit durch wenige Konzepte
- Modularisierung und strenge Typprüfung als Voraussetzungen zur frühen Fehlererkennung durch Übersetzer, Binder und Laufzeitsystem
- überprüfbare Schnittstellen(beschreibungen) der Moduln
- wenig (kein) Overhead durch Laufzeitsystem
- projektierbares Zeitverhalten

- spezielle Datentypen, z.B.
  - ⋆ process (task, thread)
  - ⋆ exception
  - ⋆ semaphore, event
  - ⋆ bit, byte (auf jeden Fall Einzelbits setzen/löschen, z.B. in Statusregistern)
  - ⋆ clock, time, duration
- Ansprechen von Speicheradressen, z.B. für "memory mapped" E/A, Adreßrechnung
- Zeitdienste
- Zeitüberwachung aller Wartezustände
- Aktivierung von Prozessen
  - ⋆ sofort
  - ⋆ zu bestimmten Zeitpunkten
  - ⋆ zyklisch
  - ⋆ in bestimmten Zeitabständen innerhalb eines Zeitintervalls
  - ⋆ bei bestimmten Ereignissen
- Prioritäten
- einfache Dateistrukturen (z.B. fester Speicherbereich)

- Synchronisation (auch in verteilten Systemen)
- schnelle Kommunikation
- vielseitige Protokolle zur Kommunikation (Feldbus-Protokolle, LAN-Protokolle, V24 usw.
- Interrupt– und Ausnahme–Behandlung, insbesondere benutzerdefinierte Unterbrechungs–Antwortprogramme
- Unterstützung der vielseitigen Peripherie und der Coprozessoren
- Definition virtueller Geräteklassen mit einheitlichen Schnittstellen
- einheitliches E/A–Konzept für Standard– und Prozeß–E/A, einschließlich Beschreibung der Datenwege
  - ⋆ Beispiel:
    Endlagenschalter Nr. 53 ist verbunden mit Eingang 5 einer Digitaleingabe auf Steckplatz Nr. 8
    PEARL:
    Datenweg: schalter53 → DIGIN ∗8 ∗ 5
    READ schaltzustand FROM schalter53;
    IF schaltzustand = '1'B THEN ...;

  - ⋆ Übersetzer muß Namen der Datenwege kennen und die korrekten Treiberprogramme mit Parametern einsetzen
  - ⋆ bei Änderung des Datenweges (z.B. Umstecken) nur Wegbeschreibung aufdatieren; Programm nicht ändern, aber neu übersetzen (neue Treiber bzw. geänderte Parameter)
- Evtl. Funktionen für verteilte Systeme, z.B
  - ⋆ Operationen auf Daten auf anderen Rechnern im Netz
  - ⋆ Objektverwaltung im Netz (aktuelle Verzeichnisse der Namen mit Attributen, lokale Teilkopien)
  - ⋆ Konfigurierungs–Funktionen zur Zuordnung der Programm–Moduln zu Rechnern
  - ⋆ (automatische) Rekonfiguration in Fehlersituationen
  - ⋆ Synchronisierung Uhren

- Anmerkung:
  Viele Fragen bei der Sprachdefinition sind immer noch offen, z.B.
  - ⋆ Bei Ausnahmen: Wo wird nach der Behandlung fortgesetzt?
  - ⋆ Soll es untergeordnete Tasks (Hierachien) geben?
  - ⋆ Darf eine Task neu gestartet werden, falls die vorhergehende Inkarnation noch aktiv ist?
  - ⋆ Darf eine Task eine andere löschen und wenn ja, in welchem Zustand ?

  Beispiel für mögliche Fehlersituation:

```
P1:                              P2:
öffneVentil (Nr53);              ...
delay (2 sec);          ←        TERMINATE (P1);
schließeVentil (Nr53);           ...
```

10.3 Modul–Strukturen

- Moduln enthalten:
  - ⋆ Daten
  - ⋆ Typdefinitionen
  - ⋆ lokale und/oder globale Prozeduren
  - ⋆ Tasks
- evtl. Konfigurierungseinheiten in verteilten Systemen
- ermöglichen Datenkapselung (objektorientierte Konzepte mit Vererbung)
- Modularten
  - ⋆ Funktionsmodul (Bibliotheksmodul)
    nur Funktionen und Prozeduren im Modul
  - ⋆ Datenmodul
    nur permanente Objekte oder globale Systemdaten im Modul
  - ⋆ Objektmodul
    (Abstrakte Datenstruktur, Rechenstruktur)
    Daten und zugehörige Operationen im Modul; von außen sichtbare Schnittstelle ist definiert (Geheimnisprinzip)

  - ⋆ Klassenmodul (Modul als Schema)
    (classes, templates, generic packages)
    erst durch Instantiierung (constructors)
    entstehen Exemplare
    objektorientierte Programmierung

- Sprachen trennen die Modulbeschreibung in
  - ⋆ Schnittstellenbeschreibung
    - ◦ Definitionsteil, Kopf, head, interface
    - ◦ Deklaration exportierter Objekte (Typen, Daten, Funktionen)
    - ◦ bei Prozeduren: Spezifikation Typ aller Parameter!
    - ◦ Spezifikation importierter Objekte
    - ◦ "opaque type"–Objekte, private Objekte
      Datentyp im Modul definiert, aber von außen nur der Name sichtbar
  - ⋆ Implementierungsteil
    - ◦ Rumpf, body
    - ◦ Realisierung der exportierten Objekte
    - ◦ Realisierung der internen Objekte
    - ◦ Initialisierung der Objekte
- möglichst Erkennung aller Schnittstellenfehler beim Übersetzen oder spätestens beim Binden

10.4 Programmstruktur und Moduln in Ada

- Freie Ada-Compiler für Windows und Linux unter
  `http://www.adahome.com/Resources/Compilers`
  - ⋆ ObjectAda Compiler von Aonix
  - ⋆ GNAT: The GNU NYU Ada 9X Translator
- Programm strukturiert in
  - ⋆ Blöcke
  - ⋆ Unterprogramme
  - ⋆ Pakete
  - ⋆ Tasks
- beliebige Schachtelung
- Hauptprogramm ist eine Prozedur (main task)
- Übersetzungseinheiten
  - ⋆ Bibliothekseinheiten (Bibl–E)
    - ◦ Unterprogramm–Vereinbarung
    - ◦ Unterprogramm–Rumpf
    - ◦ Paketvereinbarung
    - ◦ Vereinbarung einer generischen Einheit

  - ⋆ Sekundär–Einheiten
    - ◦ Unterprogramm– oder Paketrumpf
- Pakete und Tasks müssen, Unterprogramme können in Spezifikation (Kopf, head) und Rumpf (body) aufgeteilt werden
- im Kopf stehen die sichtbaren (Export–) Objekte
- die Angabe private spezifiziert die opaken Objekte, deren Realisierung verborgen (nicht ausnutzbar) bleibt
- Anweisungsteil des Pakets wird einmalig beim Abarbeiten der Paketdeklaration ausgeführt
- Syntax Paket

```
PACKAGE <name>  IS
   <sichtbare Vereinbarungen> ;
   [PRIVATE <Vereinbarungen> ;]
END <name>

PACKAGE BODY <name>  IS
   <lokale Vereinbarungen> ;
BEGIN
   <Anweisungen>
   [EXCEPTION <Ausnahmebehandlung> ]
END <name> ;
```

- Spezifikation muß vor Rumpf übersetzt werden
  - ⋆ fehlender Prozedurrumpf kann durch Stummelrumpf (stub unit) ersetzt werden. Beispiel:

```
PACKAGE BODY master IS
   ...
   PROCEDURE q (x, y: REAL) IS
      SEPARATE;
   ...
END master;
```

  - ⋆ und getrennt unter Angabe des Pakets, in dem sich der Stummelrumpf befindet, übersetzt werden. Beispiel:

```
SEPARATE (master)
PROCEDURE q (x, y: REAL) IS
   BEGIN
   ...
   END q;
```

- Private Typen
  - ⋆ Realisierung privater Typen kann zwar gesehen (z.B. wegen Speicherbedarf), aber nicht geändert oder benutzt werden
  - ⋆ Typ im Rumpf muß mit Spezifikation im Kopf übereinstimmen
  - ⋆ Änderung der Spezifikation erfordert Neuübersetzung aller abhängigen Bibliothekseinheiten bzw. benutzenden Programme
  - ⋆ in Spezifikation eines Pakets
    TYPE <name> IS [LIMITED] PRIVATE
  - ⋆ im Rumpf dann
    PRIVATE <Spezifikation von name>
  - ⋆ ohne LIMITED: außerhalb muß Zuweisung und Vergleich (= oder ≠) für diesen Typ existieren
  - ⋆ mit LIMITED: Zuweisung und Vergleich durch eigene Paketfunktionen

- Beispiel

```
PACKAGE complex_numbers IS
   TYPE compl IS PRIVATE;
   i: CONSTANT compl;
   FUNCTION + (x, y: compl) RETURN compl;
   FUNCTION – (x, y: compl) RETURN compl;
   PRIVATE
      TYPE compl IS
         RECORD
         re, im: FLOAT;
         END RECORD;
      i: CONSTANT compl := (0.0, 1.0);
END complex_numbers;

PACKAGE BODY complex_numbers IS
   ...
   FUNCTION + (x, y: compl) RETURN compl IS
      BEGIN ... END +
   FUNCTION – (x, y: compl) RETURN compl IS
      BEGIN ... END –
END complex_numbers;
```

Aufruf in Block, z.B.:

```
DECLARE
   WITH complex_numbers;
   USE complex_numbers;
   u,v: compl;
BEGIN
   u:= (2.5, 1.0);
   v:= i;
   u:= u + v;
END
```

- Anmerkungen
  - ⋆ WITH <Bibliothekseinheit>
    - ◦ Benutzung Bibliothekseinheit bei Übersetzung
    - ◦ Bibliothekseinheit muß bei der Übersetzung schon bekannt sein (mindestens mit ihrem Definitionsteil)
    - ◦ zyklische WITH–Relationen daher nicht möglich
  - ⋆ USE <Bibliothekseinheit>
    - ◦ im Rumpf kann der qualifizierende Paketname entfallen

- Generische Einheiten
  - ⋆ Unterprogramme oder Pakete
  - ⋆ dienen als Programmschablonen zur Mehrfachverwendung von Algorithmen
  - ⋆ freie Parameter (generic parameters) nach dem Wortsymbol GENERIC
  - ⋆ Parameter sind Objekte und Objekttypen
  - ⋆ freie Parameter bei Übersetzung der Ausprägung im Rumpf textuell durch die aktuellen Parameter ersetzt (entspricht templates in C++)
  - ⋆ Ausprägung (Instantiierung) wird durch NEW und Belegung der freien Parameter deklariert
  - ⋆ Beispiel: Generisches Paket zur Erzeugung von Ringpuffern verschiedener Länge und mit Elementen verschiedener Art

```
GENERIC
   TYPE sometype IS PRIVATE;
PACKAGE queue_handling IS
   TYPE queue (maxlength: NATURAL)
      IS PRIVATE;
   PROCEDURE enqueue (q: IN OUT queue;
       elem: IN sometype);
      ...
   PROCEDURE dequeue ...
   FUNCTION nonempty ...
   FUNCTION nonfull ...
   PRIVATE
      SUBTYPE index
         IS CARDINAL RANGE 0..1000;
      TYPE queue (maxlength: NATURAL) IS
         RECORD
         head: index := 0;
         tail: index := 0;
         q: ARRAY (0..maxlength-1) OF sometype;
         END RECORD;
END queue_handling;
```

```
PACKAGE BODY queue_handling IS
   ...
END queue_handling;
```

- Im Rumpf befinden sich die Prozeduren enqueue, dequeue, nonfull, nonempty. Es gibt keinen ausführbaren Teil im Rumpf.
- Ausprägung, z.B. eines Blocks

```
DECLARE
   PACKAGE int_queue
      IS NEW queue_handling (INTEGER);
   USE int_queue;
   longqueue: queue (200);
   shortqueue: queue (10);
BEGIN
   IF nonfull(shortqueue) THEN ...;
   ...
END int_queue;
```

10.5 Objektorientierung in ADA95

- ADA83 hat strenges Typkonzept mit enger Datenkapselung
- zur besseren Wiederverwendbarkeit von Paketen durch Erweiterung der Funktionen wird in ADA95 Objektorientierung mit Vererbung hinzugefügt
- Objektorientierung basiert in ADA 95 auf der Typerweiterung von RECORD (TAGGED), ähnlich STRUCT in C++
- Beispiel:

```
TYPE messwert IS TAGGED
   RECORD
      m1, m2 : INTEGER;
   END RECORD;
```

Erweiterung um Zeitstempel in anderem Paket z.B.

```
TYPE zeitmesswert
      IS NEW messwert WITH
   RECORD
      ankunft: time;
   END RECORD;
```

- abgeleitete Typen erben die Operationen des Vorgängers, können diese umdefinieren und neue hinzufügen (Typhierarchie mit Einfachvererbung)
- in ADA95 trägt die gesamte Ableitungshierarchie (Grundtyp mit Ableitungen) die Bezeichnung "Klasse"
- Zum Grundtyp T existiert der Klassentyp T'CLASS (polymorpher Typ), der als formaler Parameter beim Überladen von Unterprogramm-Namen dienen kann
- beim Aufruf mit einem aktuellem Parameter aus der Ableitungshierarchie wird dynamisch die passende Operation verwendet
- dieses Vorgehen heißt in ADA95 dispatching (spätes Binden)

- Abstrakte Typen und Unterprogramme
  - ⋆ auf oberstem Niveau können abstrakte Typen (dh. ohne Angabe des Inhalts) und abstrakte Unterprogramme (ohne Angabe eines Rumpfes) definiert werden
  - ⋆ in der Ableitungshierarchie kann dieser Grundstock von allgemeinen Typen und Operationen auf verschiedene Weise realisiert werden
  - ⋆ der Anwender sucht sich unter den Varianten die für seine Aufgabe geeignete Implementierung aus
  - ⋆ Beispiel
    - ◦ abstrakter Typ: menge
    - ◦ abstrakte Operationen: einfügen. löschen, istenthalten, Vereinigung, Durchschnitt
    - ◦ als Implementierung kann der abgeleitete Typ menge in verschiedener Weise angeboten werden, z.B. als
      - ▷ BitSET (bei kleiner Grundmenge)
      - ▷ Feld
      - ▷ als geordnete einfach verkettete Liste

- Hierarchische Bibliotheken
  - ⋆ bei Wiederverwendung vorhandener Pakete sollen Erweiterungen nicht zur Neuübersetzung aller bisherigen Benutzer des Grundpakets führen, da sie ja die neuen Operationen nicht verwenden
  - ⋆ Erweiterung darf also nicht in Grundpaket eingehen
  - ⋆ ADA95 führt aus diesem Grund das Konzept von Kind–Paketen (child packages) ein und erlaubt die Definition eines Baums von Paket–Spezifikationen (hierarchical libraries)
  - ⋆ ein Kind–Paket erbt vom Eltern–Paket alle sichtbaren Operationen und darf im Rumpf alle privaten Spezifikationen des Elternpakets verwenden
  - ⋆ durch Weitervererbung wird eine Paket–Hierarchie aufgebaut
  - ⋆ Kind–Paket enthält bei Spezifikation den (qualifizierenden) Namen des Elternpakets

⋆ Beispiel: Paket Für Komplexe Zahlen wird um Darstellung in Polarkoordinaten erweitert

```
PACKAGE complex_numbers IS
   TYPE compl IS PRIVATE;
   FUNCTION + (x, y: compl) RETURN compl;
   PRIVATE
      TYPE compl IS
         RECORD
         re, im: FLOAT;
         END RECORD;
END complex_numbers;
PACKAGE complex_numbers.polar IS
   TYPE polar IS PRIVATE;
   FUNCTION abs (x: compl) RETURN FLOAT;
   FUNCTION arc (x: compl) RETURN FLOAT;
   FUNCTION polar2compl (r, theta: FLOAT)
         RETURN compl;
   PRIVATE
      TYPE polar IS
         RECORD
         r, theta: FLOAT;
         END RECORD;
END complex_numbers.polar;
```

- ⋆ Kindpakete können PRIVATE spezifiziert werden
- ⋆ ADA95 gibt Regeln, die die Beziehungen von Eltern–Kind–Paketen und zwischen Geschwister–Paketen beschreiben, um zu verhindern, daß in untergeordneten Paketen durch Umdefinition PRIVAT–Spezifikationen sichtbar gemacht werden können
- ⋆ Beispiele zu den Regeln
  - ◦ Kontext–Umgebung des Elternpakets gilt auch für Nachkommen
  - ◦ WITH–Benutzung eines Kindpaket umfaßt auch WITH für alle Vorfahren
  - ◦ Kind benötigt keine WITH–Beziehung zu seinen Eltern, jedoch zu den Geschwistern
  - ◦ sichtbares Kind kann kein PRIVATE–Geschwisterpaket verwenden
  - ◦ PRIVATE–Teil und Rumpf jedes Kindes kann PRIVAT–Teil seiner Eltern (und aller Vorfahren) benutzen

10.6 Tasks in ADA

- für Tasks (Prozesse) Datentyp TASK
- Deklaration in Prozeduren oder Paketen
- auch Komponenten von Feldern oder Records
- auch aktuelle Parameter
- Tasks werden automatisch beim Abarbeiten der Deklaration aktiv, aber erst am Ende dieses Deklarationssteils gestartet
- Tasks terminieren automatisch beim Erreichen des Blockendes, falls sie nicht auf des Ende von untergeordneten Tasks warten müssen
- Block wird erst verlassen, wenn alle in ihm vereinbarten Tasks beendet sind
- keine Parameter beim Start einer Task
- bei Deklaration ist Trennung in Spezifikationsteil und Rumpf obligatorisch
- Spezifikationsteil darf nur Eingänge (ENTRY) für Aufrufe von Rendezvous–Anweisungen aus anderen Tasks deklarieren mit Angabe der formalen Parameter der zugehörigen ACCEPT–Anweisung im Rumpf

- Syntax

```
TASK [TYPE] name IS
   ENTRY ename (<formale Parameter> );
   ENTRY ...
END name;

TASK BODY name IS
   <deklarationen>
BEGIN
   <Anweisungen>
   [EXCEPTION [<exception_handler> ]]
END name;
```

bei Anweisungen u.a.:

```
ACCEPT ename (<formale Parameter> ) DO
   ...
END ename;
```

- es gibt nur die Operation ABORT zum Datentyp TASK (gewaltsames Beenden)
- durch die Blockstruktur können Taskaufrufe geschachtelt auftreten. Die umfassende Task wird durch implizite Synchronisation des Betriebssystems erst beendet, wenn alle in ihr deklarierten und damit gestarteten Tasks (subtasks) beendet sind (Ausnahme: Tasks die durch Zeiger–Kreierung aktiviert wurden).

10.7 Partitionen in ADA 95

- ADA 95–Programm kann aus mehreren Partitionen (partitions) bestehen
- entsprechen einem ADA 83–Programm
- haben einen eigenen Adreßraum
- können Tasks enthalten
- Programm kann aus Partitionen auf verschiedenen Rechnern zusammengesetzt sein
- active partitions enthalten tasks und main ()
- passive partitions enthalten nur Daten und/oder Unterprogramme
- Partition wird erst beendet, wenn alle ihre Tasks beendet
- werden von außen oder durch eine sog. Environment–Task angestoßen, bei deren Abarbeitung die in ihr enthaltene main–Prozedur aufgerufen wird
- Partition kann mit einer anderen mittels RPC (remote procedure call) oder über gemeinsame Daten einer dritten Partition kommunizieren

10.8 Synchronisation

- Prozesse, die an gemeinsamen Aufgaben kooperieren, müssen sich koordinieren (Synchronisation) und über Prozeßgrenzen (Adreßraum, Rechnernetz) Daten austauschen (Kommunikation)
- Synchronisation erforderlich wegen
  - ⋆ Konkurrenz und Konsistenz bei Benutzung exklusiver Betriebsmittel
  - ⋆ Einhaltung von Reihenfolgen (Produzent vor Konsument)
  - ⋆ Auftraggeber/Auftragnehmer–Diensten (client server)
- Synchronisations–Operationen legen nicht unbedingt die Reihenfolge der Zuteilung fest (faire oder unfaire (prioritätengebundene) Zuteilung)
- logische Synchronisation:
  es wird feste zeitliche Reihenfolge von Aktionen in verschiedenen Prozessen erzwungen, z.B. Prozeß P_i erst fortsetzen, wenn P_j beendet

- betriebsmittelorientierte Synchronisation: Operationen auf gemeinsamen Objekten (Variablen, Betriebsmittel) dürfen von verschiedenen Prozessen aus nur mit Einschränkungen erfolgen, z.B. nicht gleichzeitig (wechselseitiger Ausschluß, mutual exclusion)
- Lösungen (vgl. Grundvorlesungen) u.a. durch Semaphore (Dijkstra 68) mit den unteilbare Operationen P (sema) und V (sema)
- Spezielle Echtzeitforderungen
  - ⋆ Eine V–Operation wählt (unfair) den Prozeß höchster Priorität aus der Warteschlange zu diesem Semaphor
  - ⋆ Zeitüberwachung des Wartezustandes
  - ⋆ sichere implizit arbeitende Sprachkonstrukte
  - ⋆ bei Fehlerabruch im kritischen Bereich keine Blockade wartender Prozesse
  - ⋆ Prioritätsvererbung

- Rendezvous in ADA83
  - ⋆ Synchrone Kommunikation
  - ⋆ Definition eines Eingangs in einer Task
  - ⋆ ACCEPT-Anweisung zu den ENTRYs in dieser Task
  - ⋆ Aufruf dieses Eingangs aus einer andern Task wie ein Prozduraufruf mit Parametern
  - ⋆ sowohl aufrufende als auch aufgerufene Task warten, bis Anweisungen im ACCEPT ausgeführt sind
  - ⋆ alternatives Warten durch SELECT mit guards (WHEN)
  - ⋆ watchdog möglich

```
GENERIC
   TYPE item IS PRIVATE
PACKAGE readerwriter IS
   PROCEDURE read(x: OUT item);
   PROCEDURE write(x: IN item);
END;

PACKAGE BODY readerwriter IS
   V: item;
```

```
TASK control IS
   ENTRY start;
   ENTRY stop;
   ENTRY write(x:item);
END;

TASK BODY control IS
   readers: integer :=0;
BEGIN
   ACCEPT write(x:IN item) DO
      v := x;
   END;
   LOOP
      SELECT
         WHEN write'count=0 =>
         ACCEPT start;
         readers:=readers+1;
      OR
         ACCEPT stop;
         readers:=readers-1;
      OR
         WHEN readers=0 =>
         ACCEPT write(x:IN item) DO
            v := x;
         END;
```

```
        END SELECT;
      END LOOP;
    END control;

    PROCEDURE read(x:OUT item) IS
    BEGIN
      control.start; x:=v; control.stop;
    END read;

    PROCEDURE write(x:IN item) IS
    BEGIN
      control.write(x);
    END write;

END readerwriter
```

Beispiel für Zeitbedingungen im rendezvous:

```
SELECT
  ACCEPT this(···) DO
    ···
  END;
OR
  ACCEPT that(···) DO
    ···
  END;
OR
  DELAY 10.0;
    ··· – time out statements
END SELECT;
```

- Protected type in ADA 95
  - ⋆ ADA 83 ohne direkte Operationen für exklusive Programmteile. Semaphore durch eigenen Prozeß darzustellen, wobei durch synchrone Kommunikation die Exklusivität gesichert wird. Dieses Vorgehen über Tasks ist ineffizient
  - ⋆ daher in ADA 95 als Typ und Deklaration PROTECTED [TYPE]
  - ⋆ Objekte dieses Typs beinhalten (zusammengesetzte) Typen und Daten sowie die benötigten Operationen (Funktionen, Prozeduren, ENTRY)
  - ⋆ Laufzeitsystem sichert, daß PROTECTED PROCEDURES nur exklusiv ausgeführt werden
  - ⋆ auf PROTECTED FUNCTION kann gleichzeitig mehrfach lesend zugegriffen werden (nur IN–Parameter), da read–only–Eigenschaft ohne Nebeneffekte
  - ⋆ Vorgehen ähnelt Monitoren
  - ⋆ auch ENTRY-Aufruf mit Bedingung möglich

- ⋆ Priority Inheritance hier möglich: Angabe von PRAGMA Locking_Policy (Ceiling_Locking)
- ⋆ beim Auftreten einer Ausnahme im kritischen Bereich wird der Block verlassen und die Belegung automatisch aufgehoben, sodaß keine Blockade anderer erfolgen kan
- ⋆ PROTECTED PROCEDURES in ADA 95 als Interrupt–Antwortprogramme
- ⋆ Definition wie Pakete mit sichtbarem und privatem Teil
- ⋆ Beispiele

```
TYPE index IS MOD N;
TYPE itemarray IS ARRAY (index) OF item;

PROTECTED TYPE datensatz IS
   PROCEDURE schreibe (n: IN index);
                                  wert: item);
   FUNCTION lese (n: index) RETURN item;
PRIVATE
   tabelle: itemarray;
END datensatz;

PROTECTED BODY datensatz IS
   PROCEDURE schreibe (n: IN index;
                                  wert: item) IS
      BEGIN tabelle(n) := wert; END schreibe;
   FUNCTION lese (n: index) RETURN item IS
      BEGIN RETURN tabelle(n); END lese;
END datensatz;

...

meindatensatz : datensatz;
...
v := meindatensatz.lese(m);
```

```
PROTECTED TYPE sema
                          (init_s: INTEGER := 1) IS
   ENTRY P;
   PROCEDURE V;
PRIVATE
   count: INTEGER := init_s;
END sema;

PROTECTED BODY sema IS
   ENTRY P WHEN count > 0 IS
   BEGIN
      count := count - 1;
   END P;
   PROCEDURE V IS
   BEGIN
      count := count + 1;
   END V;
END sema;

--Aufrufbeispiel

s : sema;
...
s.P;
... --Exklusive Anweisungen
s.V;
```

10.9 Ausnahmen (exceptions)

- Ausnahmen sind Unterbrechungssignale aus dem aktuellen Programmablauf
- Beispiele
  - ⋆ arithmetische Alarme
  - ⋆ Speicherschutzalarme
  - ⋆ Fehlermeldungen vom Laufzeitsystem wie z.B. Indexgrenzenüberschreitung
- Echtzeitsysteme erfordern zumindest eine saubere, abschließende Behandlung und im Notfall den Übergang in einen "fail–safe–Zustand" des technischen Prozesses
- Sprachmittel für spezifische Behandlung auf Benutzerebene notwendig
- Benutzer kann in der Sprache Ausnahmen selbst (logisch) definieren und auslösen; nützlich, um bei bestimmten Ereignissen, die das Programm entdeckt, gezielt den sequentiellen Programmablauf zu durchbrechen

- problematisch ist Festlegung der Fortsetzungsstelle nach Ausführung (abhängig von der Block– und Modulhierarchie)
  - ⋆ Variante Terminierung
    - ◦ verursachende Programmstelle wird verlassen
    - ◦ nach Setzen von Zustandsinformation wird in die nächsthöhere Instanz zurückgekehrt (z.B. zur Aufrufstelle der verursachenden Prozedur)
    - ◦ die Ausnahme–Behandlung kann mehrfach nach oben weitergereicht werden bis zu einer abschließenden Beantwortung
  - ⋆ Variante Reparatur und Wiederholung
    - ◦ nach Reparatur wird an einer definierten Stelle fortgesetzt
    - ◦ bei einer Wiederholung kann ein anderer Algorithmus gleicher Funktion vorgesehen werden (partielle Diversität)

- PEARL
  - ⋆ SIGNAL
  - ⋆ alle SIGNAL–Namen vordefiniert und im SYSTEM–Teil beschrieben
  - ⋆ neben EOF, OVERFLOW u.a. sind implementierungsabhängige Namen für Laufzeitfehler vorgesehen, z.B. geplanter Startzeitpunkt schon verstrichen, Operationen an nicht (mehr) existierenden Task
  - ⋆ ON–Anweisung zur Beantwortung
  - ⋆ ON signalname DO <anweisungen>
  - ⋆ ON signalname DO SYS
    für Default–Behandlung
  - ⋆ beim Verlassen des definierenden Blocks wird die übergeordnete ON–Anweisung aktiv bzw. beim Fehlen die SYS–Reaktion

- ADA
  - ⋆ EXCEPTION
  - ⋆ Ausnahmen können in Anweisungen, bei Deklarationen und im Rendezvous auftreten
  - ⋆ Benutzer kann Ausnahmen selbst definieren

    ausn3: EXCEPTION;

  - ⋆ vor- oder benutzerdefinierte Ausnahmen durch RAISE auszulösen

    RAISE ausn3;

  - ⋆ Anweisungen zur Behandlung sind am Ende eines Rahmens (Block, UP–Rumpf, TASK, PACKAGE) anzugeben
  - ⋆ nach dem Eintreten werden die Anweisungen abgearbeitet und der Rahmen verlassen
  - ⋆ falls keine Behandlung angegeben, wird die Ausnahme an den umgebenden Rahmen weitergereicht (exception propagation), bis eine Behandlung oder letzlich ein Programmabbruch erfolgt

⋆ Syntax

```
EXCEPTION
   WHEN exceptionname =>
        <Anweisungsfolge> ;
   ...
   WHEN OTHERS =>
        <Anweisungsfolge> ;
- - Ende des Rahmens
```

⋆ an Stelle des Namens der Ausnahme auch mit | getrennte Liste von Namen möglich

⋆ OTHER am Ende der WHEN–Folge

⋆ es gibt viele vordefinierte Ausnahmen wie

- constraint_error
- numeric_error
- programm_error
- storage_error
- tasking_error

10.10 Unterbrechungs–Antwortprogramme

- Unterbrechung (interrupt) kommt von CPU, Peripherie oder Prozessen
- Unterbrechung durchbricht sequentiellen Programmablauf
- Formulierung von Unterbrechungs–Antwortprogrammen auf Sprachebene ist wichtige, effizient zu realisierende Forderung der Echtzeitprogrammierung
- selbst wenn vorgefertigte Gerätetreiber viele Interrupts vom Benutzer fernhalten, bleiben Interrupts zur Prozeßführung, die nur spezifisch behandelbar
- Antwortprogramme fest (statisch) oder kurzzeitig (dynamisch) zugeordnet
- default–Behandlung durch Betriebssystem

- ADA 83
  - ⋆ Antwortprogramm wird als Rendezvous (Kommunikation) mittels ACCEPT–Anweisung in einer Task ausgeführt
  - ⋆ zugehörige ENTRY–Spezifikation im Task–Kopf enthält Interrupt–Identifikator
  - ⋆ Interrupt wird vom Laufzeitsystem in ENTRY–Aufruf umgesetzt
  - ⋆ Nachteile: ineffizient und verzögert durch Taskwechsel

⋆ Beispiel

```
TASK intantwort IS
   ENTRY alarm;
   FOR alarm USE AT 8#220#;
      - - Interrupt 144
   PRAGMA PRIORITY (120)
END intantwort;

TASK BODY intantwort IS
   ...
      - - Deklarationen
BEGIN
   LOOP
      ACCEPT alarm DO
         ...
      END;
   END LOOP;
END intantwort;
```

- ADA 95
  - ⋆ Nachteile des Konzepts von ADA 83 durch Verwendung von PROTECTED PROCEDURES als Antwortprogramme weitgehend vermieden
  - ⋆ mit priority_ceiling_locking werden derartige Prozeduren beim Auftreten des Interrupts exklusiv mit genügend hoher Priorität ausgeführt
  - ⋆ Gesamtprioriätsbereich hängt vom Betriebssystem ab
  - ⋆ Prioritäten für Tasks mindestens 30 Werte
  - ⋆ höhere Prioritätsklasse für Interrupts
  - ⋆ Zuordnung zu Signalen statisch oder dynamisch
  - ⋆ Identifikatoren der Unterbrechungen vom Hersteller im Paket ADA.Interrupt.Names

⋆ Beispiel statische Zuordnung

```
PROTECTED PROCEDURE alarm IS
   PROCEDURE response;
      - - muß parameterlos sein
   PRAGMA
      attach_handler (response, Alarm_Id);
END alarm;
PROTECTED PROCEDURE BODY alarm IS
   PROCEDURE response IS
      ...
   END response;
END alarm;
```

⋆ Beispiel dynamische Zuordnung

```
PROTECTED PROCEDURE alarm IS
   PROCEDURE response;
      - - muß parameterlos sein
   PRAGMA interrupt_handler (response);
END alarm;

PROTECTED PROCEDURE BODY alarm IS
   - - wie oben

 - - späterer Prozeduraufruf:
attach_handler (alarm.response, Alarm_Id);
```

10.11 Real-Time JAVA

- Real-Time Specification for Java (RTSJ)
- ausgearbeitet durch Real-Time for Java Expert Group (RTJEG)
- first release update Juni 2000
- JSR-000001 Real-time Specification for Java Final Release Dezember 2001
- Wesentliche Konzepte:
  - ⋆ präemptiver Scheduler mit mindestens 28 Prioritäten
  - ⋆ Prioritätsvererbung
  - ⋆ no heap realtime thread
  - ⋆ abstract class: memory area mit physical, immortal and scoped memory
  - ⋆ Monitor, no wait queues, event, timer
  - ⋆ asynchronous interrupt exception
- PDF-Dokument mit Syntax und Semantik unter `http://www.rtj.org/`

10.12 Multi-Agent Distributed Real-Time System

- Eigenentwicklung am Lehrstuhl
- Echtzeit Programmierung
  mit verteilten Agenten
  - ⋆ Top Down Zerlegung der Anwendung in autonome Agenten
  - ⋆ zustandsbasiertes regelorientiertes Programmiermodell
  - ⋆ Spezifikation der Aufträge (contracts), die Agent annimmt
  - ⋆ Möglichkeit der Verhandlung (contract net protocol)
  - ⋆ Verteilung Agenten auf heterogene Rechner
  - ⋆ Entwurf unabhängig von Hardware
  - ⋆ Transparenz der Kommunikation
  - ⋆ Isolation der Ein/Ausgabe in Subagenten
  - ⋆ bestimmbares Zeitverhalten der Agenten
  - ⋆ zeitkritische Agenten in eigene Hardware

- Sprachdefinition
  - ⋆ globale Deklaration der verwendeten Zielrechner (**targetdecl**)
  - ⋆ globale Deklaration der verwendeten Kommunikationsverbindungen (**conndecl**)
  - ⋆ Vollständige Deklaration eines Agenten
    - ○ Zielrechner (**target**)
    - ○ Instantiierung der Subagenten (**decls**)
    - ○ Deklaration der aufrufbaren Aufträge (**contracts**) mit
      - ▷ Name und Parameter
      - ▷ Kostenfunktion für Verhandlung (**cost**)
      - ▷ Ausführungsteil (**do**)
    - ○ Aktionsteil (**states**)

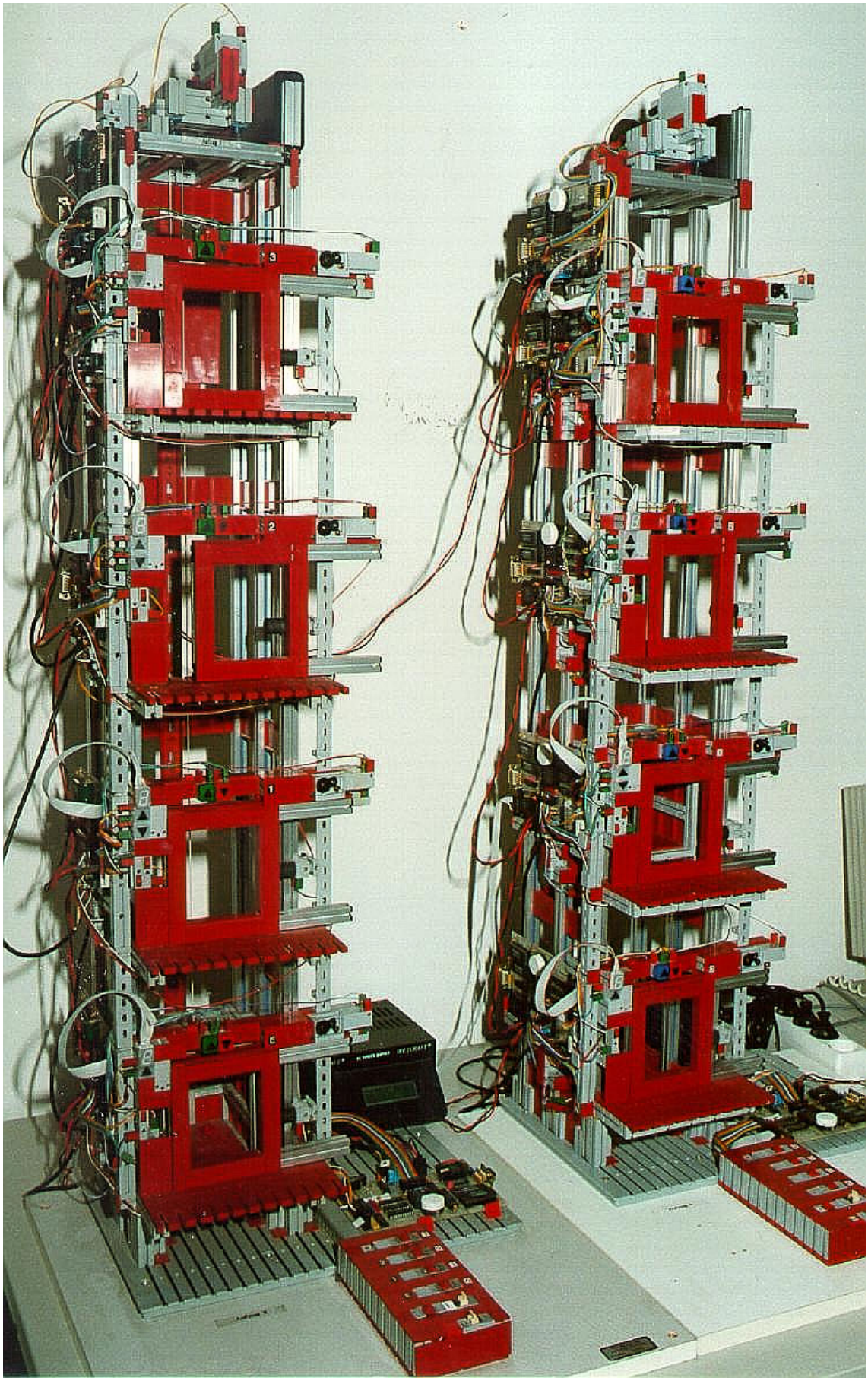
Beispiele für Subagenten:

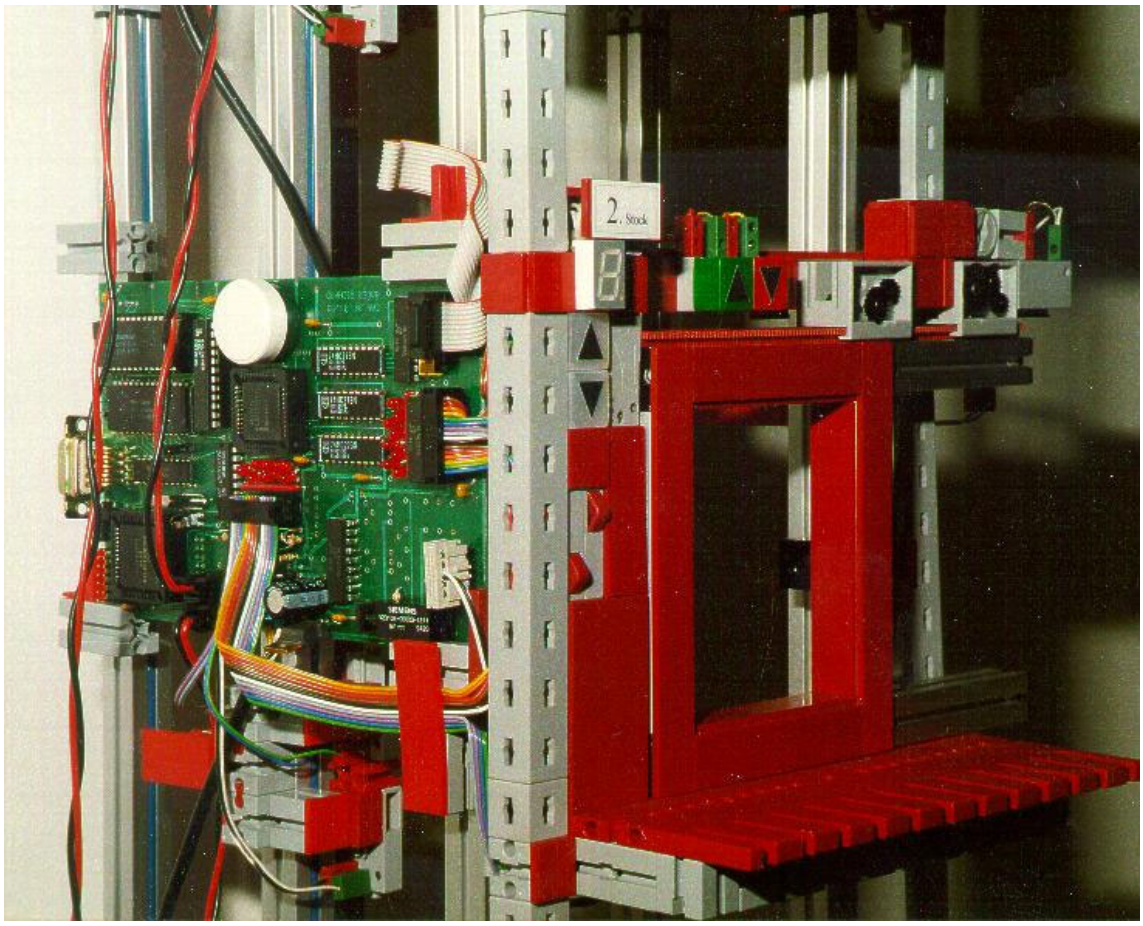
	Definition	**contracts**
VARINT	<Name>	.set, .put
ARRAYINT	<Name>	
	(<Dimension>)	.set, .put
DIGIN	<Name>	
	(<Kanal> ,	.on, .off
	<id-on> ,<id-off>)	.<id-on> , .<id-off>
DIGOUT	<Name>	.on, .off
	(<Kanal> ,	.<id-on> , .<id-off>
	<id-on> ,<id-off>)	.toggle(t_on,t_off)
ANAIN	<Name>	
	(<Kanal>)	.get
ANAOUT	<Name>	
	(<Kanal>)	.put(<value>)
TIMER	<Name>	.restart, .tout(<delay>)
CFUNC	<Bib-Name>	.<C-subroutine-id> (. . .)
PID	<Name>	
	(<ad-in> ,<da-out> ,	.setpoint(<value>)
	<kp> ,<ki> ,<kd>)	

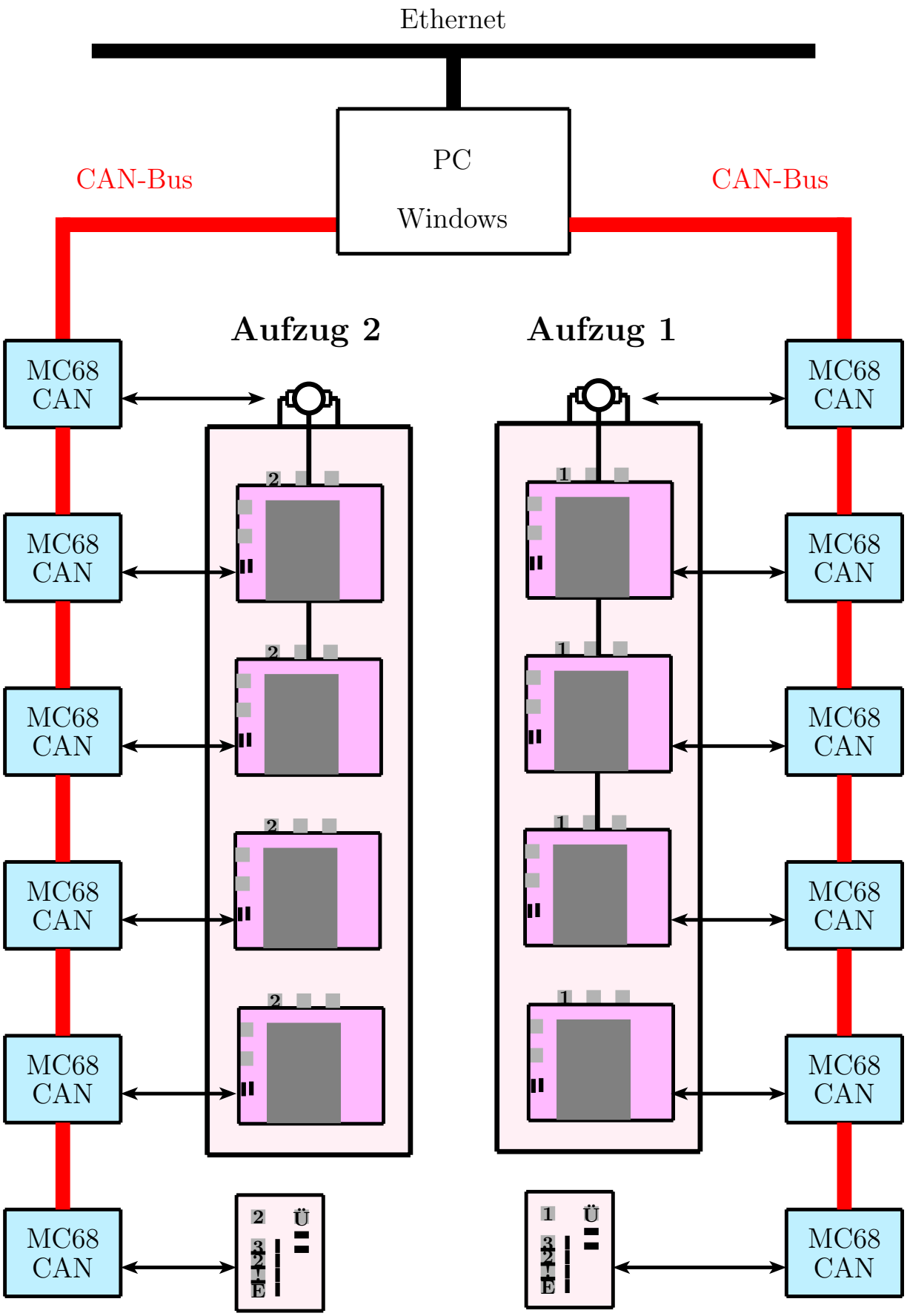
Aktionsteil eines Agenten

- ⋆ endlicher Automat mit Zuständen
- ⋆ zyklische Bearbeitung der Aktionen des aktiven Zustands
- ⋆ jede Aktion durch Bedingung gebunden (guard $\Rightarrow$)
- ⋆ Aktion wird ausgeführt, wenn Bedingung erfüllt
- ⋆ Anweisungen einer Aktion:
  - ◦ Aufruf eines Auftrags eines Agenten oder Subagenten
  - ◦ Zustandsübergang (**newstate**)
  - ◦ arithmetische und logische Ausdrücke
- ⋆ Spezielle Bedingung: **once**
- ⋆ Spezielle Zustände: **initial** , **shutdown**

• Prototypische Anwendung: Doppelaufzug




2. Stock


Ethernet
PC
Windows
CAN-Bus
CAN-Bus
Aufzug 2
Aufzug 1
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
MC68
CAN
2
Ü
1
E
2
Ü
1
E

- Beispiel Türagent

```
targetdecl mc68hc11 mc12;
conndecl mc12 CAN to mc_elev CAN;

agent door2 target mc12;
decls
   DigOut motor(c0, on, off);
   DigOut direction(c2, close, open);
   DigIn open_key(d1, off, on);
   DigIn closed_key(d4, off, on);
   DigIn light_barrier(d0, interrupted, ok);
   Timer delay;
contracts
   open do newstate opening;
states
   closed/shutdown:
      once ⇒ motor.off;
   opening:
      once ⇒ {direction.open; motor.on;}
      open_key.on ⇒ newstate waiting;
   waiting:
      once ⇒ {motor.off; delay.restart(5000);}
      delay.tout ⇒ newstate closing;
   closing/initial:
      once ⇒ {direction.close; motor.on;}
      closed_key.on
            ⇒ {elevator1.start; newstate closed;}
      light_barrier.interrupted
            ⇒ {motor.off; newstate opening;}
endagent;
```

- Beipiel für Verhandlung

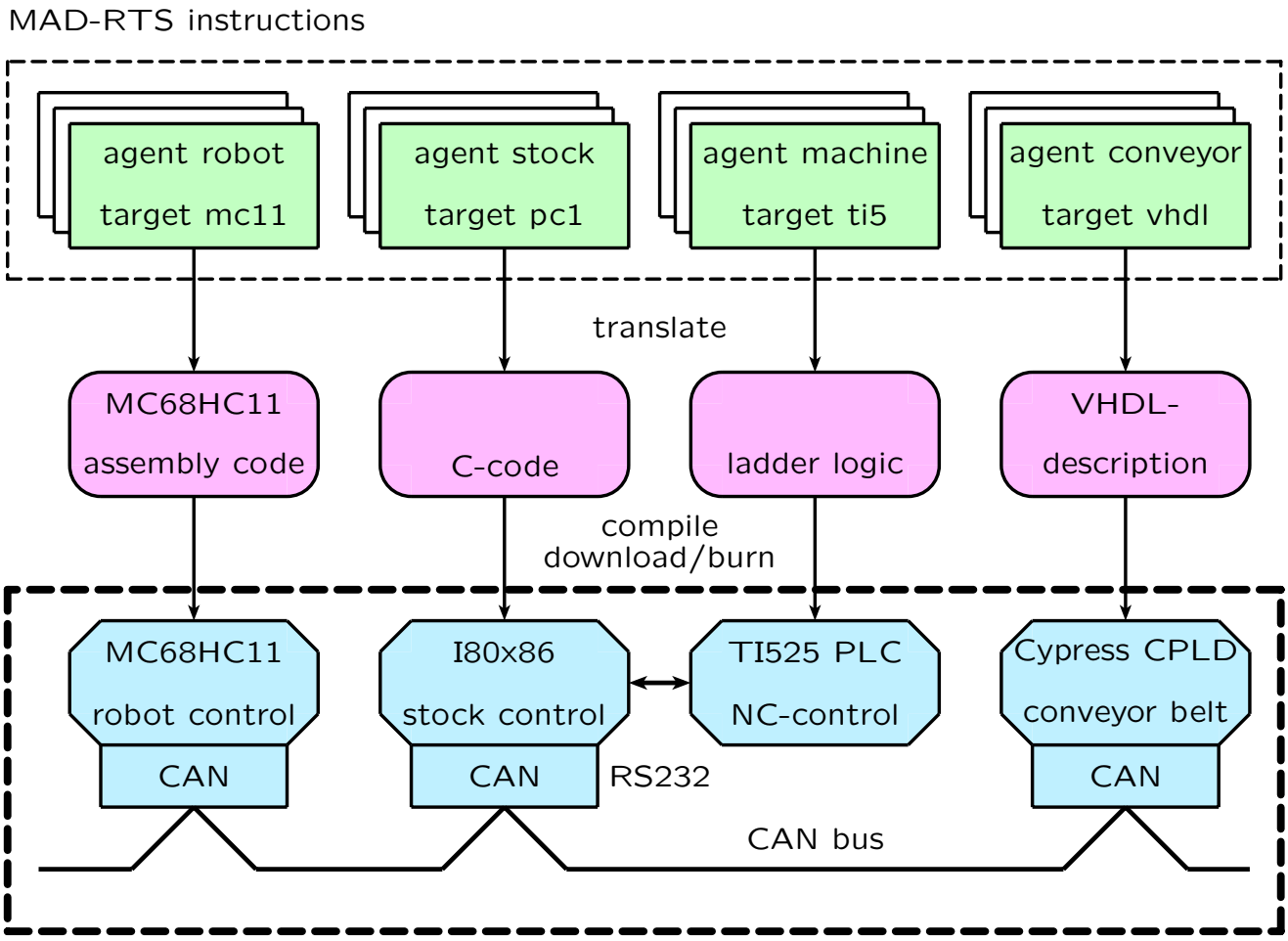
a) Aufzugsagent

```
targetdecl mc68hc11 mc_elev;
agent elevator1 target mc1;
decls
   varinteger aktfloor;
   arraybool(4) stop;
contracts
   get (integer floor)
      cost ((instate busy)*100+
            abs(floor-aktfloor)*20);
      do stop(floor).set(true);
states
      ...
      busy:
         ...
endagent;
```

b) Auftrag, um einen der beiden Aufzüge nach Stockwerk 2 zu holen

```
elevator1 | elevator2.get(2)
```

- Implementierung
  - ⋆ Übersetzer
    - ◦ Syntaxbeschreibung in YACC
    - ◦ Lexikalische Analyse in FLEX
    - ◦ Syntaktische Analyse in BISON
    - ◦ Erzeugung einer Zwischensprache
    - ◦ Implementiert in C++
    - ◦ auf PC unter MSDOS/Windows
  - ⋆ Code Generatoren und Laufzeitsystem für
    - ◦ Intel 80x86
    - ◦ Microcontroller MC68HC11
    - ◦ Speicherprogrammierbare Steuerung (SPS) TI500 Serie
    - ◦ Programmable Logic Device (PLD) über VHDL
  - ⋆ Kommunikation
    - ◦ Direkter Aufruf der Aufträge von Agenten auf dem gleichen Targetrechner
    - ◦ Automatische Übermittlung der Aufträge über CAN-Bus bzw. serielle Schnittstelle



Multi-agent distributed system

- Vorteile von MAD-RTS
  - ⋆ problemorientierte Spezifikation
  - ⋆ TopDown Entwurf unabhängig von der Verteilung auf Rechner
  - ⋆ transparente Kommunikation
  - ⋆ deterministische Antwortzeiten
  - ⋆ Separierung von zeitkritischen Modulen in Agenten auf dediziertem Prozessor
  - ⋆ verzögerungsfreie Reaktion durch Hardware-Agenten (PLD)
  - ⋆ nicht zeitkritische Agenten mit Verhandlungsprotokoll
  - ⋆ tragfähiges Konzept von Signalebene bis Verhandlungsebene
  - ⋆ selbstdokumentierende strukturierte Programme

10.13 Synchrone Programmiersprachen: ESTEREL

- G. Berry, Frankreich, ab 1984 entwickelt
- andere Sprachen
  - ⋆ Lustre
  - ⋆ Signal
  - ⋆ Statecharts
  - ⋆ SE
- ESTEREL-Modell für reaktive Systeme
  - ⋆ Schnittstellenmodul zur Umgebung
  - ⋆ Kernmodul mit logischer Ablaufsteuerung
  - ⋆ Datenmodul (data handling layer bei Berry)
- ESTEREL ist Programm-Generator zur Erzeugung Kernmodul
- Schnittstellen- und Datenmoduln in konventionellen Programmiersprachen geschrieben
- Ziel: deterministische (beweisbare) Abläufe

- Arten der Synchronität
  - ⋆ asynchrone Systeme
  - ⋆ zeit-synchron
    - ◦ gemeinsamer Zeittakt der Komponenten
    - ◦ alle Komp. machen gleichzeitig einen Schritt (im Takt)
  - ⋆ EA-synchron
    - ◦ Ankunft des Eingangssignals und Senden des Ausgangssignals als Ergebnis im gleichen Takt $\Rightarrow$ als gleichzeitig zu betrachten
  - ⋆ nachrichten-synchron
    - ◦ wie Ada
    - ◦ Nachrichtenübertragung erst, wenn Sender sendebereit und Empfänger empfangsbereit
    - ◦ der Sender oder der Empfänger wartet bis der andere bereit
- ESTEREL ist
  - ⋆ zeit-synchron
  - ⋆ EA-synchron

- synchrone Systeme in ESTEREL
  - ⋆ alle Reaktionen der Prozesse erfolgen sofort ohne Zeitverzug
  - ⋆ alle Aktionen haben die Bearbeitungszeit 0 z.B. Befehle, Nachrichtenaustausch, Abläufe in Prozessen
  - ⋆ $\Rightarrow$ alle Prozesse haben dasselbe Bild über den Zustand des Systems
  - ⋆ $\Rightarrow$ der Zustand der Umgebung ändert sich während einer Aktion nicht
  - ⋆ Befehle benötigen nur Zeit, wenn dies explizit angegeben wird, z.B.
    await 30 MILLISECOND
    (wartezustand dauert exakt 30 ms)
- die Anweisung

  every 1000 MILLISECOND do
  emit SECOND
  end

  ergibt in einem synchronen System exakt alle 1000 ms das Signal SECOND, aber nicht in einem asynchronen System

- Anmerkung: MILLISECOND ist ein Signal, das mit einem Uhrentakt verbunden ist, eine entsprechende Anweisung kann mit jedem Signal geschrieben werden, z.B.

  ```
  every 60 MILLIMETER do
     emit CHECK
  end
  ```

- Wie realistisch ist die Annahme?
  - ⋆ Bearbeitungsschritt = alle Operationen, die beim Zustandsübergang ausgeführt werden müssen (Reaktion auf das Eingangssignal)
  - ⋆ synchrones System entspricht Ablauf in einem sequentiellen Automaten
  - ⋆ $\Rightarrow$ keine Unterbrechung von Bearbeitungsschritten
  - ⋆ $\Rightarrow$ nicht alle realen Systeme so modellierbar
  - ⋆ $\Rightarrow$ nur für bestimmte reaktive Systeme anwendbar, nämlich solche bei denen
    - ◦ die Arbeit des Automaten getaktet werden kann und Takt länger als längster Bearbeitungsschritt (Takt bestimmt Reaktionsfähigkeit)
    - ◦ sichergestellt ist, daß Abstand zwischen externen Signalen, die zum Zustandsübergang führen, wesentlich größer als jeder Bearbeitungsschritt (z.B. Stoppuhr, Verkaufsautomat)

- ESTEREL-Sprachelemente
  - ⋆ normale imperative Standard-Anweisungen
    - ◦ Zuweisung
    - ◦ Erzeugung Signale
    - ◦ Bedingungen
    - ◦ Schleifen
    - ◦ Exception-Behandlung
    - ◦ parallele Abläufe
  - ⋆ (temporale) Anweisungen mit Signalen
    - ◦ await <event> do
    - ◦ do <anweisungen> watching <event>
    - ◦ do <anweisungen> upto <event>
    - ◦ loop <anweisungen> each <event>
  - ⋆ Beispiel

```
do
   every STEP do
      emit JUMP
   end
watching 100 METER
```

- Signal
  - ⋆ broadcast zeitlos an alle Prozesse und die Umgebung
  - ⋆ Signal enthält auch noch Wert
  - ⋆ alle gleichbenannten Signale, die zur gleichen Zeit gesendet werden, werden in ein Signal verschmolzen
  - ⋆ Kombination der Werte der Einzelsignale spezifizierbar, z.B. Liste, Addition usw.
- parallele Anweisungen
  - ⋆ sequentielle Ausführung der folgenden Anweisung durch ; gekennzeichnet (sequencing operator)
  - ⋆ parallele Ausführung mit der folgenden Anweisung durch || gekennzeichnet (parallel operator)
  - ⋆ der Operator ; hat Vorrang vor ||
- formale Semantik vorhanden
- Übersetzung ESTEREL-Programme in sequentielle Automaten, die dann Kernmodul bilden

- Beispiel Stoppuhr

```
module STOP_WATCH:
   input START_STOP, HS;
   output TIME(integer);
   var AKT := 0 : integer in
      loop        % stopped state
         emit TIME(AKT);
         await START_STOP;
         do        % running state
            every HS do
               AKT := AKT + 1;
               emit TIME(AKT);
            end
         upto START_STOP
      end
end.
```

- Erweiterung der Stoppuhr durch reset

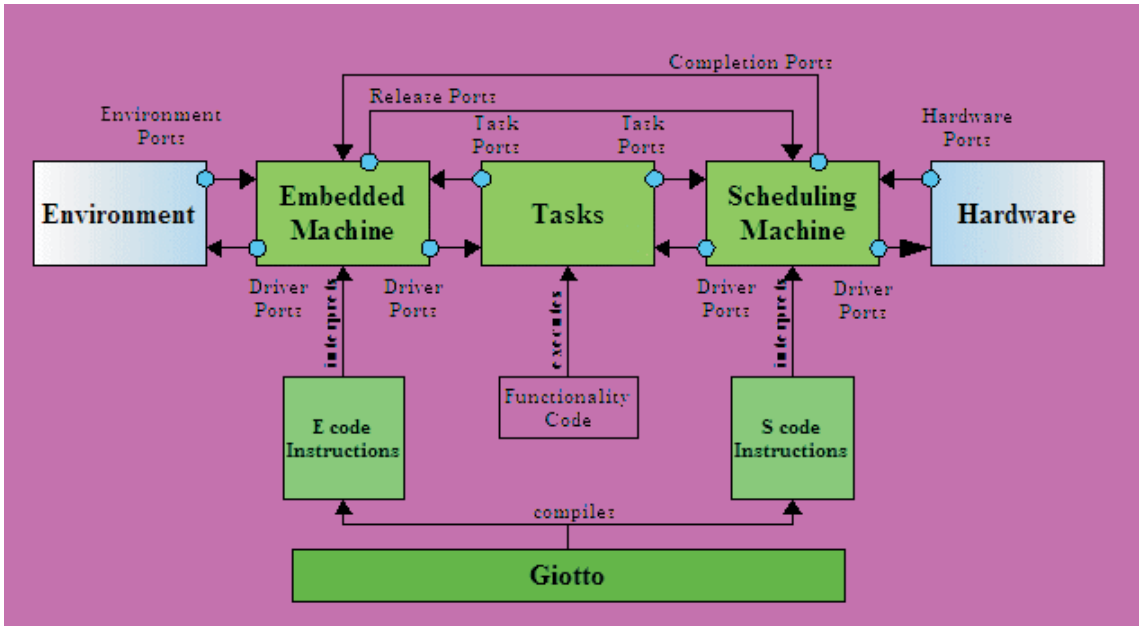
```
module STOP_WATCH_RESET:
   input START_STOP, HS, RESET;
   output TIME(integer);
   loop
      run STOP_WATCH
   each RESET
end.
```

10.14 Programmiermethode für Embedded Sytems: GIOTTO

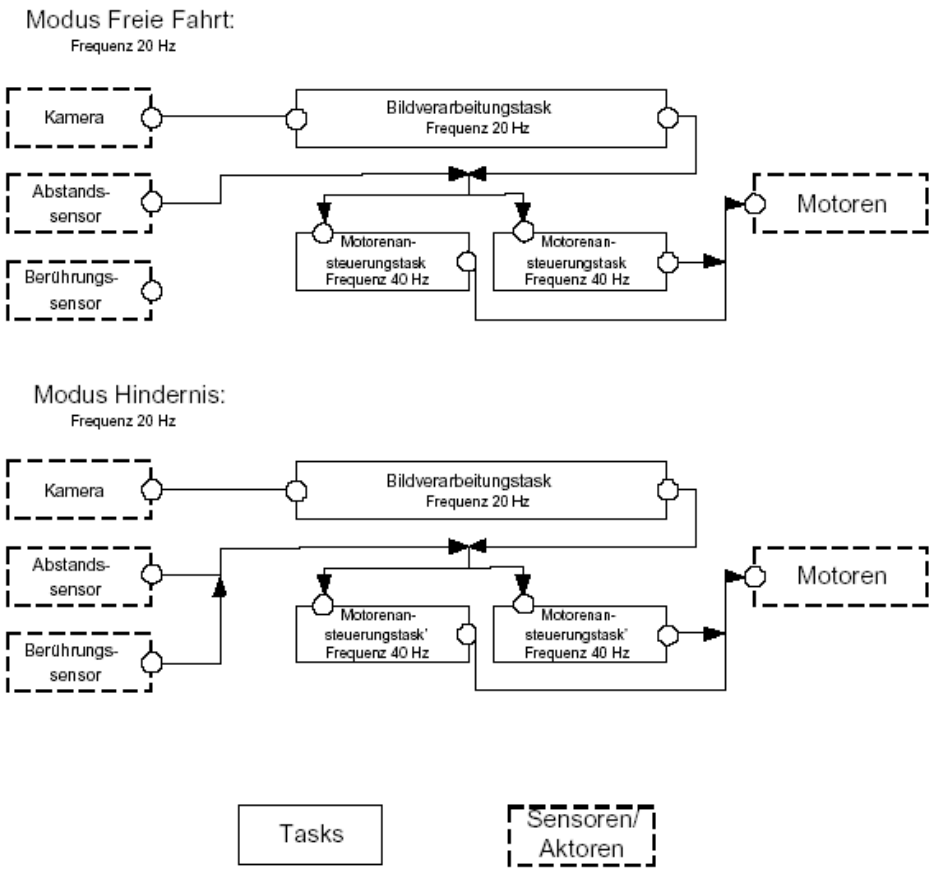
- T. Henzinger, C. Kirsch: University of Berkeley, ab 2000
- Konzepte und Ziel
  - ⋆ Trennung des Programms von der Plattform
  - ⋆ Abstraktion des Kontrollproblems in Tasks und Modi
  - ⋆ Kommunikation über Ports
  - ⋆ (zeitlose) Treiber zur Ein/Ausgabe
  - ⋆ Mapping durch den Compiler
  - ⋆ Annotation für den Compiler
- Ports
  - ⋆ sämtliche Datenkommunikation
  - ⋆ Variable eindeutigen Typs
  - ⋆ Sensorport durch Umgebung aktualisiert
  - ⋆ Aktorports und Taskports durch Giotto-Programme

- Tasks
  - ⋆ formal: Funktion f als Abbildung der Eingabeport und privaten Ports auf Ausgabeports
  - ⋆ alle Tasks periodisch
  - ⋆ keine Synchronisation innerhalb der Task
  - ⋆ Frequenz der Task in Annotation vorgegeben
- Modi
  - ⋆ jedem Modus ein Satz von Tasks zugeordnet
  - ⋆ definierte Frequenz für Moduswechsel
  - ⋆ Treiber mit Wächter und Zielmodus

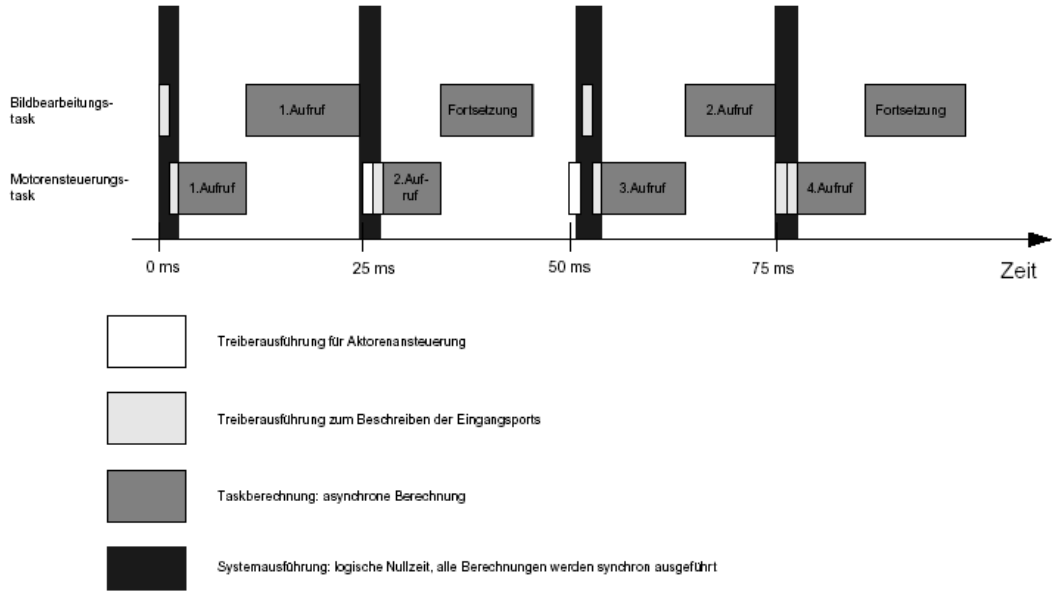
Giotto Environment



Modi und Tasks



Zeitlicher Ablauf



Beispielprogram Aufzug

```
sensor
   elevator.PortButtons  buttons  uses elevator.GetButtons;
   elevator.PortPosition position uses elevator.GetPosition;

actuator
   elevator.PortMove     motion  uses elevator.PutMoveMotor;
   elevator.PortDoor     door    uses elevator.PutDoorMotor;

output
   elevator.PortMove     tmotion := elevator.InitPortMove;
   elevator.PortDoor     tdoor   := elevator.InitPortDoor;
   bool_port             openwin := elevator.Elevator;   // used to open the frame

task Idle(elevator.PortButtons b) output (tmotion, tdoor) state () {
   schedule elevator.TaskIdle(b, tmotion, tdoor)
}

task Up(elevator.PortButtons b) output (tmotion, tdoor) state () {
   schedule elevator.TaskUp(b, tmotion, tdoor)
}

task Down(elevator.PortButtons b) output (tmotion, tdoor) state () {
   schedule elevator.TaskDown(b, tmotion, tdoor)
}

task Open(elevator.PortButtons b) output (tmotion, tdoor) state () {
   schedule elevator.TaskOpen(b, tmotion, tdoor)
}

task Close(elevator.PortButtons b) output (tmotion, tdoor) state () {
   schedule elevator.TaskClose(b, tmotion, tdoor)
}

// Actuator driver

driver Move(tmotion) output (elevator.PortMove m) {
  if constant_true() then copy_elevator.PortMove(tmotion, m)
}

driver Door(tdoor) output (elevator.PortDoor d) {
  if constant_true() then copy_elevator.PortDoor(tdoor, d)
}

// Input driver
driver getButtons (buttons) output (elevator.PortButtons b) {
  if constant_true() then copy_elevator.PortButtons(buttons, b)
}

// Mode switch driver
driver PGTC(buttons, position) output () {
   if elevator.CondPosGTCall(buttons, position) then dummy()
}

driver PLTC(buttons, position) output () {
```

```
   if elevator.CondPosLTCall(buttons, position) then dummy()
}

driver PEQC(buttons, position) output () {
   if elevator.CondPosEQCall(buttons, position) then dummy()
}
driver True() output () {
   if constant_true() then dummy()
}

start idle {

    mode idle() period 500 {
        actfreq 1 do motion(Move);
        actfreq 1 do door(Door);
        exitfreq 1 do up(PLTC);
        exitfreq 1 do down(PGTC);
        exitfreq 1 do open(PEQC);
        taskfreq 1 do Idle(getButtons);
    }

    mode up() period 500 {
        actfreq 1 do motion(Move);
        actfreq 1 do door(Door);
        exitfreq 1 do open(PEQC);
        taskfreq 1 do Up(getButtons);
    }

    mode down() period 500 {
        actfreq 1 do motion(Move);
        actfreq 1 do door(Door);
        exitfreq 1 do open(PEQC);
        taskfreq 1 do Down(getButtons);
    }

    mode open() period 500 {
        actfreq 1 do motion(Move);
        actfreq 1 do door(Door);
        exitfreq 1 do close(True);
        taskfreq 1 do Open(getButtons);
    }

    mode close() period 500 {
        actfreq 1 do motion(Move);
        actfreq 1 do door(Door);
        exitfreq 1 do idle(True);
        taskfreq 1 do Close(getButtons);
    }
}
"
```