3 Programmierung von Robotern

3.1 Arten der Programmierung

Arten

- Programmierung durch Beispiele
- Programmierung durch Training
- roboterorientierte Programmierung
- aufgabenorientierte Programmierung

3.1.1 Programmierung durch Beispiele

Programmierung durch Beispiele

- manuelle Programmierung, online
- keine Programmiersprache
- Einstellen und Speichern einer ausreichenden Anzahl von Bahnpunkten
- Master-Slave-Programmierung
- Teleoperation
- kein Einbezug von Sensoren
- keine Korrektur aufgrund von Sensorinformationen

3.1.2 Programmierung durch Training

Programmierung durch Training

- Aktion wird dem Roboter vorgeführt
- Sensoren nehmen die Aktion auf (Kamera)
- der Roboter trainiert die Ausführung der Aktion
- Forschungsthema und noch nicht für reale Aufgaben anwendbar

Probleme

- Bildverarbeitung und Bildverstehen
  - Objekterkennung
  - Positionen und Orientierungen im Raum
  - Verfolgen bewegter Objekte
- Sensorfusion (Einbeziehung unterschiedlicher Sensoren)
- Zerlegung einer Aktion in eine Folge elementarer Handlungen

3.1.3 Roboterorientierte Programmierung

Roboterorientierte Programmierung

- spezielle Roboterprogrammiersprache
- explizite Bewegungs- oder Greiferbefehle
- Der Programmierer legt fest, ***wie*** der Roboter eine Aufgabe zu lösen hat.

Roboterprogrammiersprache

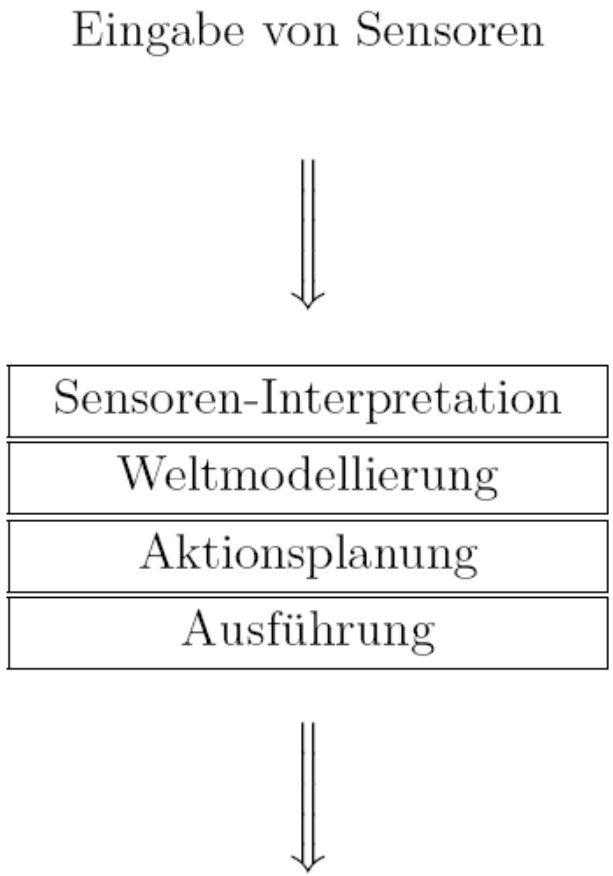
- enthält zusätzlich roboterspezifische Datentypen und Funktionen
- Datentypen zur Geometrie (Transformationsmatrizen)
- Funktionen zur Kinematik
- Funktionen zur Auswertung der Sensoren

3.1.4 Aufgabenorientierte Programmierung

Aufgabenorientierte Programmierung

- Programmierer legt nur noch fest, ***was*** der Roboter tun soll
- höhere Abstraktionsebene
- autonome mobile (intelligente) Roboter
- Einbeziehung von Sensoren
- Weltmodellierung
- Aktionsplanung
- Optimierung

Aufgabenorientierte Programmierung



Nachteile

- großer Speicherbedarf und enorme Rechenleistung
- vereinfachtes, unvollständiges Weltmodell
- Konstruktion des Weltmodells aus fehlerhaften Sensordaten
- Wenn sich die Welt zwischen den Zeitpunkt der Aufnahme der Sensordaten und der Aktion verändert, kann der Plan misslingen.

3.2 Subsumtion

3.2.1 Prinzip

Verhalten

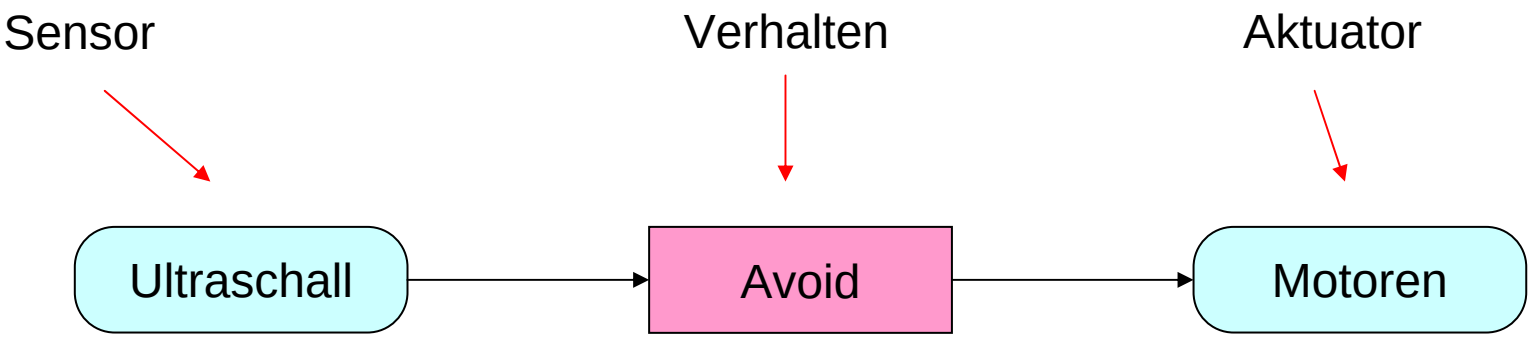
- Roboter bewegt sich
- Zubewegung auf eine Lichtquelle
- Kollisionserkennung
- Hinderniserkennung
- Reaktion auf Geräusche
- Auswertung des Batteriestandes und Bewegen zur Ladestation

Subsumtion

- Alternative zum vollständigen Planungsansatz
- sensorgesteuerte Verhalten
- Verhalten können parallel ablaufen
- Prioritätsstufen der Verhalten
- höherwertige Verhalten unterdrücken zeitweise niederwertige Verhalten
- kein geometrisches Weltmodell

3.2.2 Einfache Verhaltensnetze

Kollisionsvermeidung



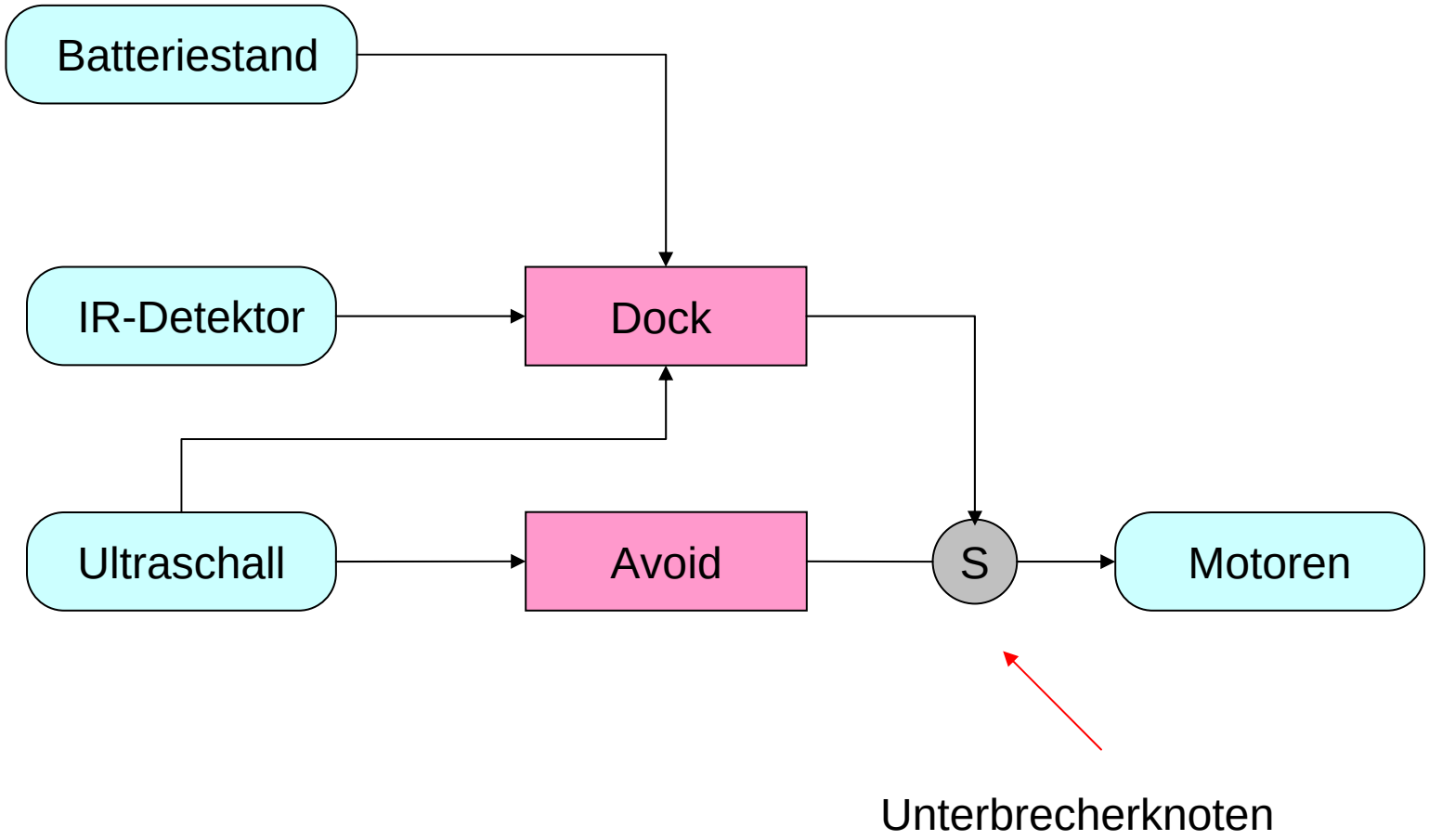
Verhalten - Avoid

- Solange alle Ultraschallsensoren einen bestimmten Grenzwert nicht unterschreiten, tut Avoid nichts.
- Wenn der vordere Ultraschallsensor ein Hindernis meldet, stoppt Avoid die Vorwärtsbewegung
- Wenn ein Ultraschallsensor außer dem hinteren ein Hindernis sieht, dreht Avoid den Roboter so lange, bis der hintere Ultraschallsensor zum Hindernis weist.
- Wenn der hintere Ultraschallsensor ein Hindernis sieht, bekommen die Motoren den Befehl, den Roboter vorwärts zu bewegen.

Verhalten - Dock

- prüft Batteriestand
- bewegen zur Ladestation (mit Infrarotsender), wenn Batterie leer
- weiterhin allen Hindernissen mit Ausnahme der Ladestation ausweichen
- höherwertiger als Avoid

Auswertung des Batteriestandes

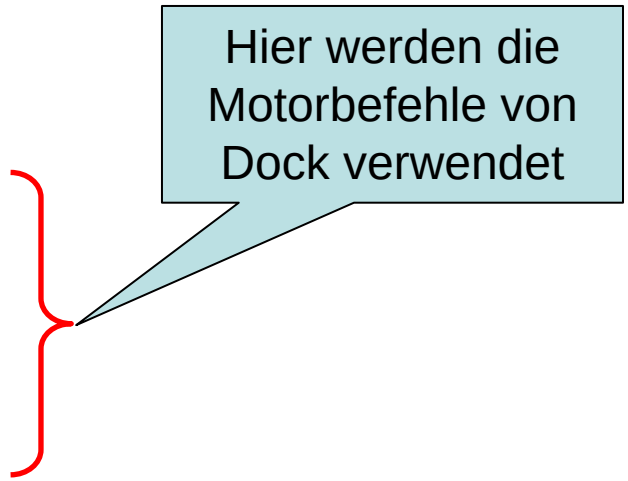


Unterbrecherknoten

- Falls keine Signale von oben (Dock), werden die Signale von Avoid zum Motor gesendet.
- Signale von oben (Dock) unterdrücken Signale von Avoid
- Die unterdrückten Signale werden nicht gespeichert und später übertragen.
- Sie gehen einfach verloren.

Folge von Signalen, die sich im Knoten S treffen

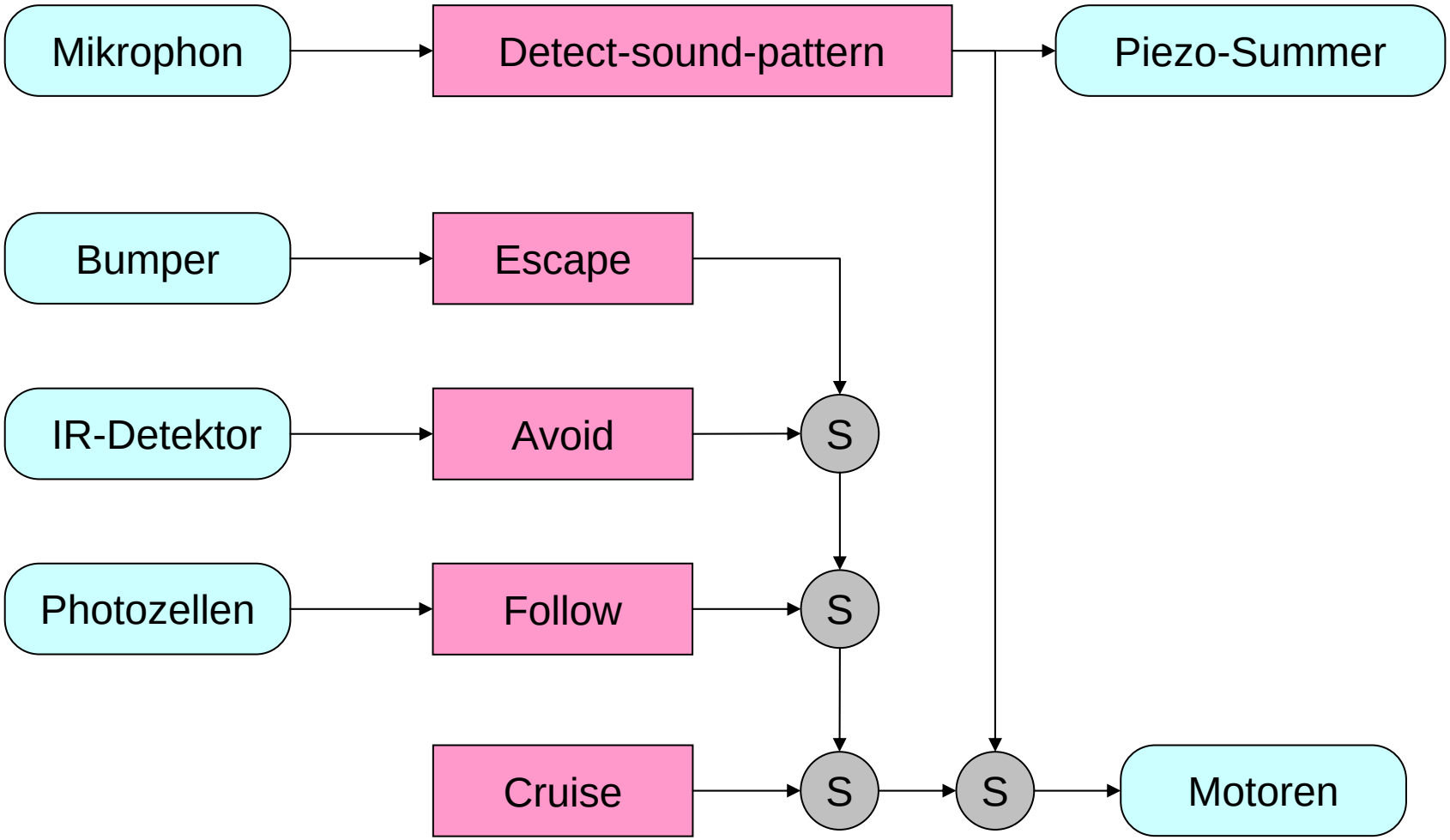
Motoren	*Avoid*	*Dock*
Links	*Links*	
Links	*Links*	
Vorwärts	*Vorwärts*	
Rechts	*Rechts*	
Vorwärts		*Vorwärts*
Rechts		*Rechts*
Vorwärts		*Vorwärts*
Vorwärts	*Rechts*	*Vorwärts*
Links	*Rechts*	*Links*
Vorwärts	*Stop*	*Vorwärts*
Links	*Stop*	*Links*
Vorwärts	*Stop*	*Vorwärts*
Stop	*Stop*	*Stop*



Verhalten

Detect-sound-pattern	→	Reagiert auf Mikrophon, Ausgabe einer Melodie, Roboter anhalten
Escape	→	Kollision, Roboter vom Hindernis wegbewegen
Avoid	→	Hindernissen ausweichen
Follow	→	Helligkeitsdifferenz an den Fotozellen Roboter in Richtung der helleren Seite drehen (Zubewegung auf eine Lichtquelle
Cruise	→	Roboter vorwärts fahren

Verhalten – Rug Warrior



Vorteile der Subsumtionsarchitektor

- Die enge Verbindung von Wahrnehmung und Aktion führt dazu, dass die meisten Verhalten als einfache Reflexe betrachtet werden können.
- einfache Berechnungen
- kein Weltmodell
- sehr kleiner Speicherbedarf
- schrittweise Erweiterung um neue Verhalten

3.2.3 Programmieren der Subsumtion

Programmieren der Subsumtion

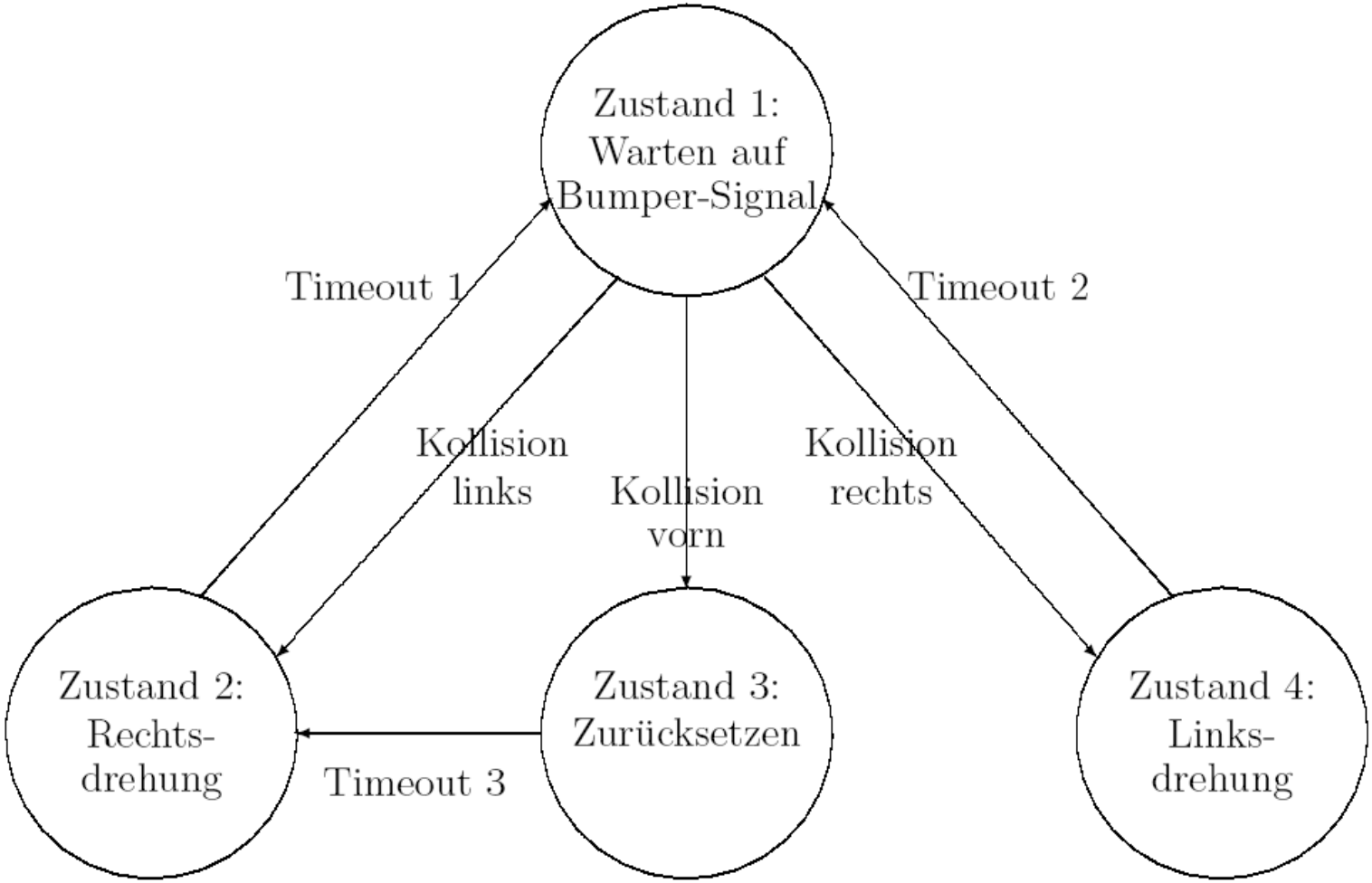
- paralleler Ablauf vieler vernetzter Verhalten (Prozess, Task)
- ***Multitasking***
- ***Scheduler*** – Steuerprogramm, dass entscheidet, wann alle anderen Programme ablaufen dürfen.
- Ein Scheduler gibt einem Prozess nur für eine kurze Zeit die Kontrolle über den Computer, dann übergibt er die Kontrolle an den nächsten Prozess und so fort.

Multitasking - Arten

- ***präemptives Multitasking*** – Der Scheduler ist fähig, einen Prozess nach einer bestimmten Zeit zu unterbrechen und dann einen anderen Prozess zu laden und auszuführen (Interactive C).
- ***kooperatives Multitasking*** – Hier entscheidet der Prozess, wann die Kontrolle an den Scheduler zurückgegeben wird, so dass der nächste Prozess ablaufen kann (endliche Automaten).

3.2.4 Multitasking mit endlichen Automaten

Verhalten – Escape - Kollisionsüberwachung



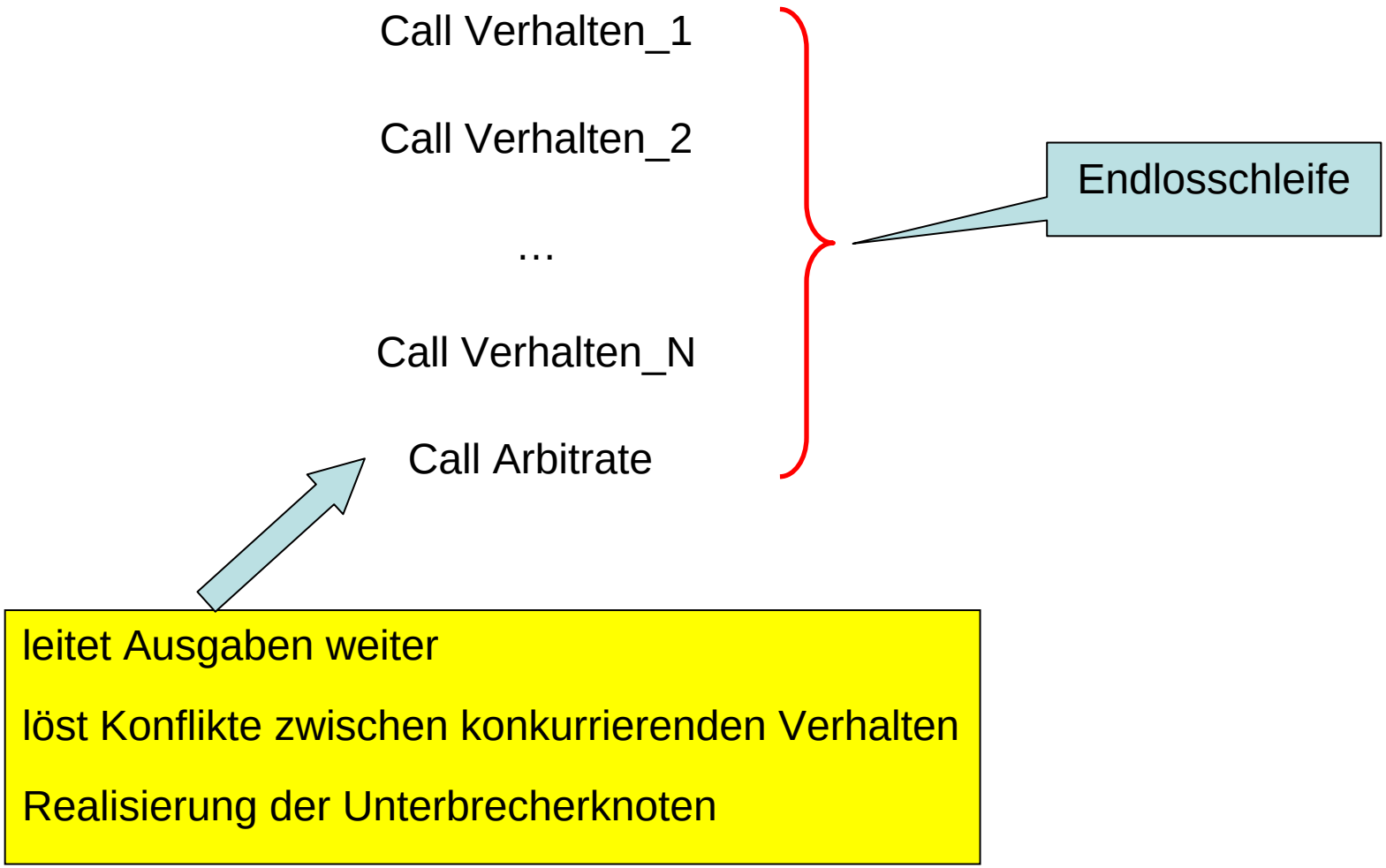
Kollisionsüberwachung

```
Escape
  Eingabesignale: bumper_hit
  Ausgabesignale: motor_command

  Zustand_1: case bumper_hit of
              NIL   : release      /* Rueckgabe der Kontrolle an den
                                      Scheduler */
              LEFT  : goto Zustand_2
              RIGHT : goto Zustand_4
              VORN  : goto Zustand_3

  Zustand_2: if time_in_this_state > timeout_1
                then goto Zustand_1
                else begin
                       motor_command = turn_right
                       release
                     end

  Zustand_3: if time_in_this_state > timeout_3
                then goto Zustand_2
                else begin
                       motor_command = back_up
                       release
                     end

  Zustand_4: if time_in_this_state > timeout_2
                then goto Zustand_1
                else begin
                       motor_command = turn_left
                       release
                     end
```

Verhalten (Prozess)

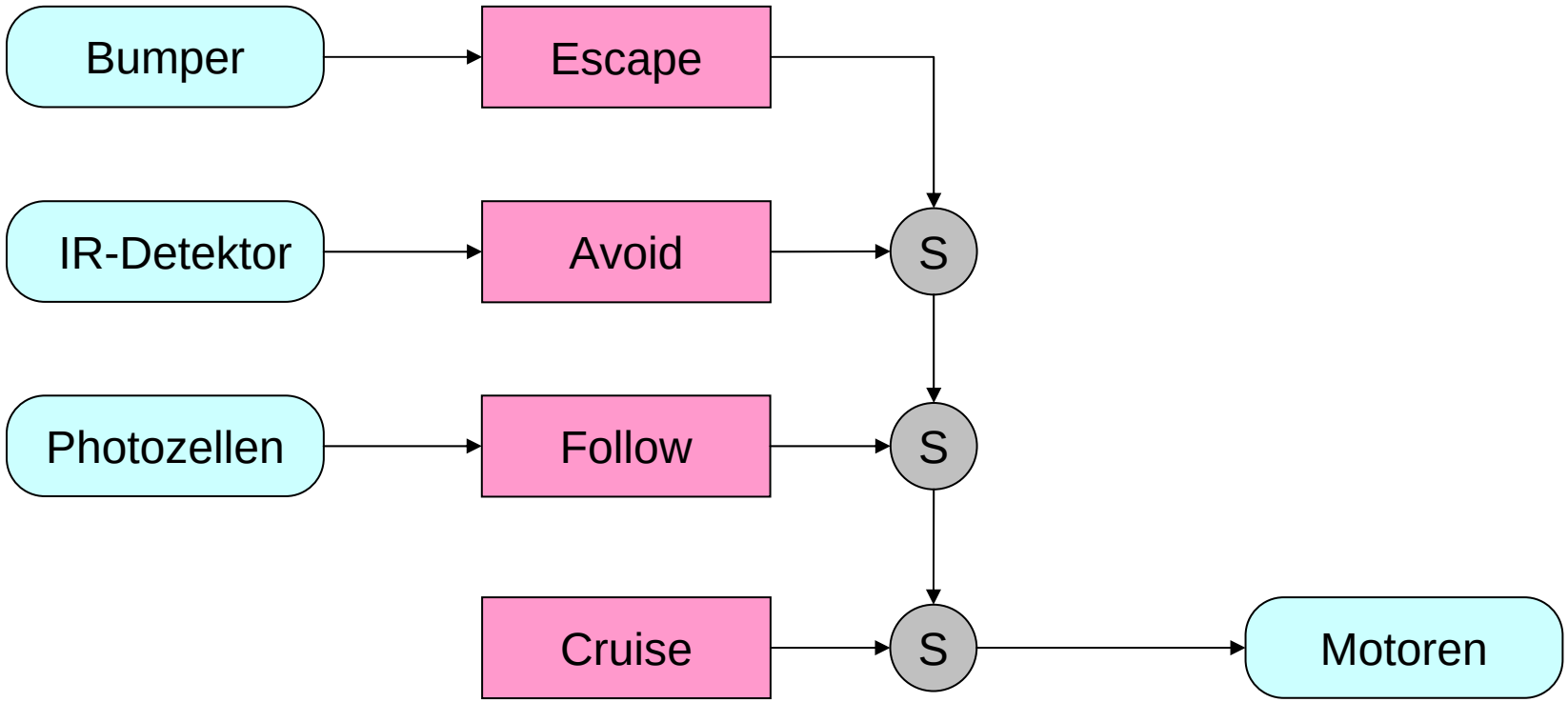
- Eingabesignale
- Ausgabesignale
- lokale Variablen
- Zustände mit Operationen

Scheduler


Call Verhalten_1
Call Verhalten_2
...
Call Verhalten_N
Call Arbitrate
Endlosschleife
leitet Ausgaben weiter
löst Konflikte zwischen konkurrierenden Verhalten
Realisierung der Unterbrecherknoten

3.2.5 Subsumtion mit IC

Ein einfaches Netz



Verhalten Cruise

```
int cruise_command                  /* Befehl an Motoren */
int cruise_output_flag              /* Ausgabe aktiv = 1 sonst 0 */

void cruise()
{ while(1)
  { cruise_command = FORWARD;       /* Rug Warrior bewegt sich vorwaerts */
    cruise_output_flag = 1;         /* Befehl ist aktiviert */
    /* wait(1000); */               /* Einmal pro Sekunde kreuzen */
  }   }
```

Verhalten Follow

```
int follow_command
ini follow_output_flag

void follow()                          /* Verfolge ein Licht */
{ int left_photo, right_photo,         /* Helleres Licht --> */
     delta;                            /* niedrigere Zahl
  while(1)
  { left_photo = analog(1);            /* Lies Analog-Eingang 1 */
    right_photo = analog(0);           /* Lies Analog-Eingang 0 */
    delta = right_photo - left_photo;
    if (abs(delta) > photo_dead_zone)
    { if  (delta > 0)                  /* Licht links */
        follow_command = LEFT_TURN;    /* nach links drehen*/
      else
        follow_command = RIGHT_TURN; /* nach rechts drehen*/
      follow_output_flag = 1;          /* Aktivieren wenn entdeckt */
    }
    else
      follow_output_flag = 0;          /* Kein Unterschied */
                                       /* Deaktivieren */
  }   }
```

Verhalten Avoid

```
int avoid_command
ini avoid_output_flag

void avoid()                              /* Infrarot-Sensor */
{ int val;
  while(1)
  { val = ir_detect();
    if (val = 0b11)
          /* Sowohl links als auch rechts sehen etwas */
    { avoid_output_flag = 1;
      avoid_command = LEFT_ARC;}  /* Kreisbogen links */
    else  if (val = 0b10)         /* Linker IR sieht etwas */
    { avoid_output_flag = 1;
      avoid_command = RIGHT_ARC;} /* Kreisbogen rechts */



    else  if (val = 0b01)          /* Rechter IR sieht etwas */
    { avoid_output_flag = 1;
      avoid_command = LEFT_ARC;}  /* Kreisbogen links */
    else                           /* Nichts zu sehen */
    { avoid_output_flag = 0; }
  }   }
```

Verhalten Escape

```
int escape_command
ini escape_output_flag

void escape()
{ while(1)
  { bump_check();                    /* Frage Bumperzustand ab */
    if (bump_left && bump_right)     /* Kollision vorn */
    { escape_output_flag = 1;
      escape_command = BACKWARD;}    /* Eine Zeitlang rueckwaerts */
      sleep(0.2);                    /* fahren */
      escape_command = LEFT_TURN;    /* dann nach links drehen */
      sleep(0.4); }
    else if (bump_left)              /* Kollision links */
    { escape_output_flag = 1;
      escape_command = RIGHT_TURN;   /* Eine Zeitlang nach rechts */
      sleep(0.4); }                  /* drehen */
    else if (bump_right)             /* Kollision rechts */
    { escape_output_flag = 1;
      escape_command = LEFT_TURN;    /* Eine Zeitlang nach links */
      sleep(0.4); }                  /* drehen */
    else if (bump_back)              /* Kollision hinten */
    { escape_output_flag = 1;
      escape_command = LEFT_TURN;    /* Zu Angreifer umdrehen  */
      sleep(0.2); }
    else                             /* Keine Kollision */
      escape_output_flag = 0;
  }  }
```

Hauptprogramm

```
void main()
{ start_process(motor_driver());
  start_process(cruise());
  start_process(follow());
  start_process(avoid());
  start_process(escape());
  start_process(arbitrate());
}
```

Funktion arbitrate

```
void arbitrate()
{ while(1)
  { if (cruise_output_flag = 1)
       { motor_input = cruise_output; }
  { if (follow_output_flag = 1)
       { motor_input = follow_output; }
  { if (avoid_output_flag = 1)
       { motor_input = avoid_output; }
  { if (escape_output_flag = 1)
       { motor_input = escape_output; }
  sleep(tick);    /* Meldungskontrolle waehrend eines Tickens */
}  }
```