5. Spezifikation von Diensten und Protokollen

Beispiel eines einfachen GSM-Zusatzdienstes: Call Forwarding Unconditional (CFU)

(entnommen aus: GSM 03.82 v5.0.0 1996)

Anrufweiterleitung zu einer definierten Zielnummer (unbedingt)

Basis-Dienstgruppen

Basic service group		Basic service (1) (2)	
number	**name**	**number (3)**	**name**
1	Speech	TS 11 TS 12	Telephony Emergency call
2	Short message service	TS 21 TS 22 TS 23	Short message MT/PP Short message MO/PP Short message CB
...	...	...	...

Die Benutzung von CFU läuft in 4 Schritten ab, sofern man CFU abonniert hat:

1. Registrierung von CFU:
   hiermit gibt man eine Zielnummer an, zu der immer umgeleitet werden soll
2. Aktivierung von CFU:
   damit ist CFU aktiv
3. Deaktivierung von CFU:
   CFU ist nicht mehr aktiv, aber die Zielnummer ist noch gespeichert
4. Löschen von CFU

Es sind i.w. die Mobilstation (MS) und das Home Location Register (HLR) beteiligt. MSC und VLR leiten die Nachrichten nur weiter.

Im folgenden Spezifikation mit MSC (Message Sequence Chart) und SDL (Specification and Description Language)

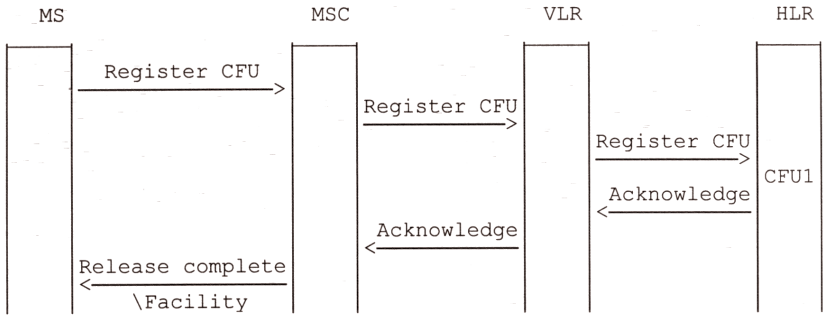


Figure 1.1: Registration of call forwarding unconditional

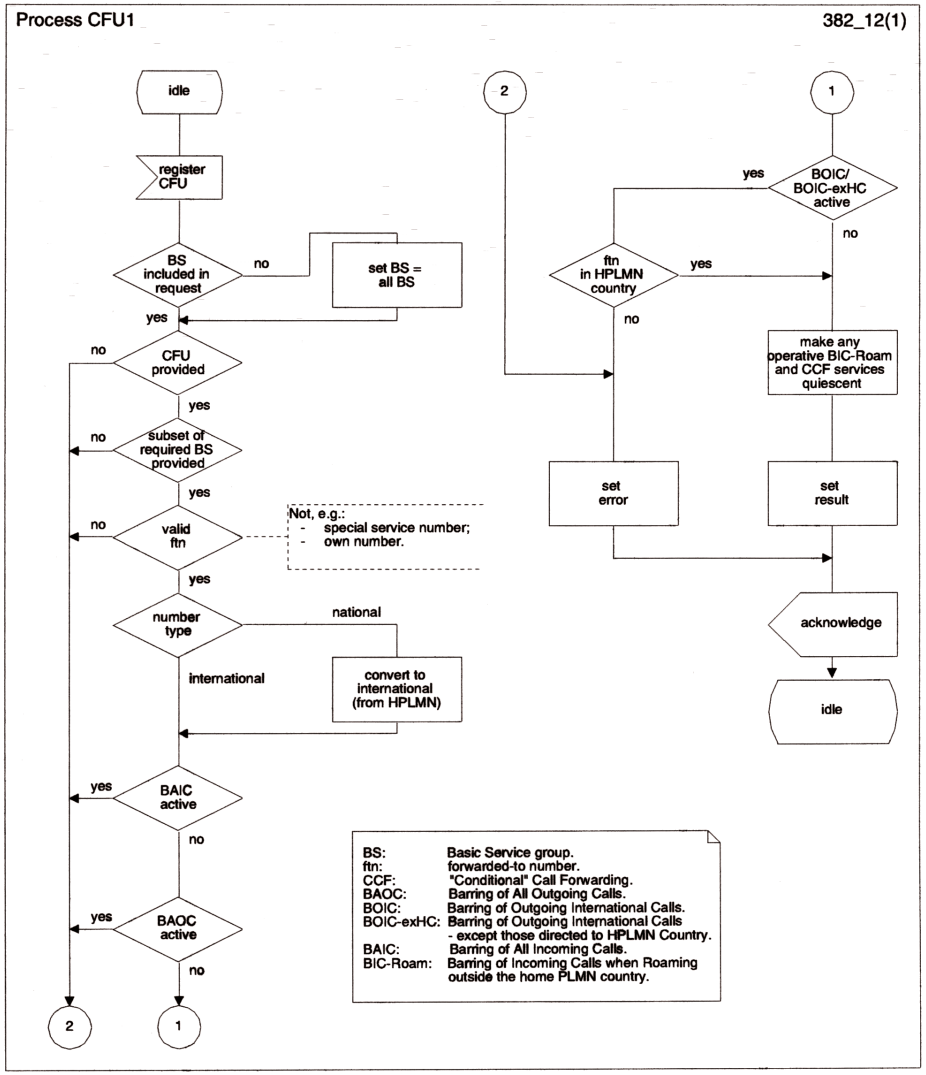


Figure 1.2: CFU1 Call forwarding unconditional registration process

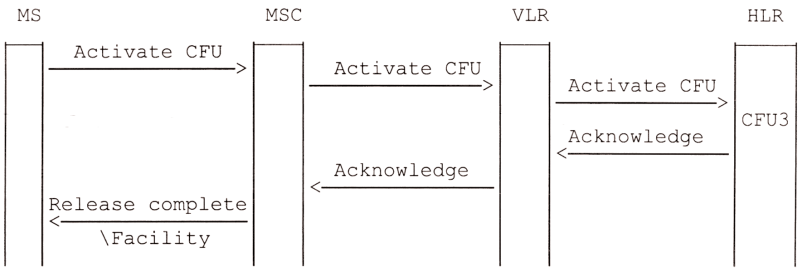


Figure 1.5: Activation of call forwarding unconditional

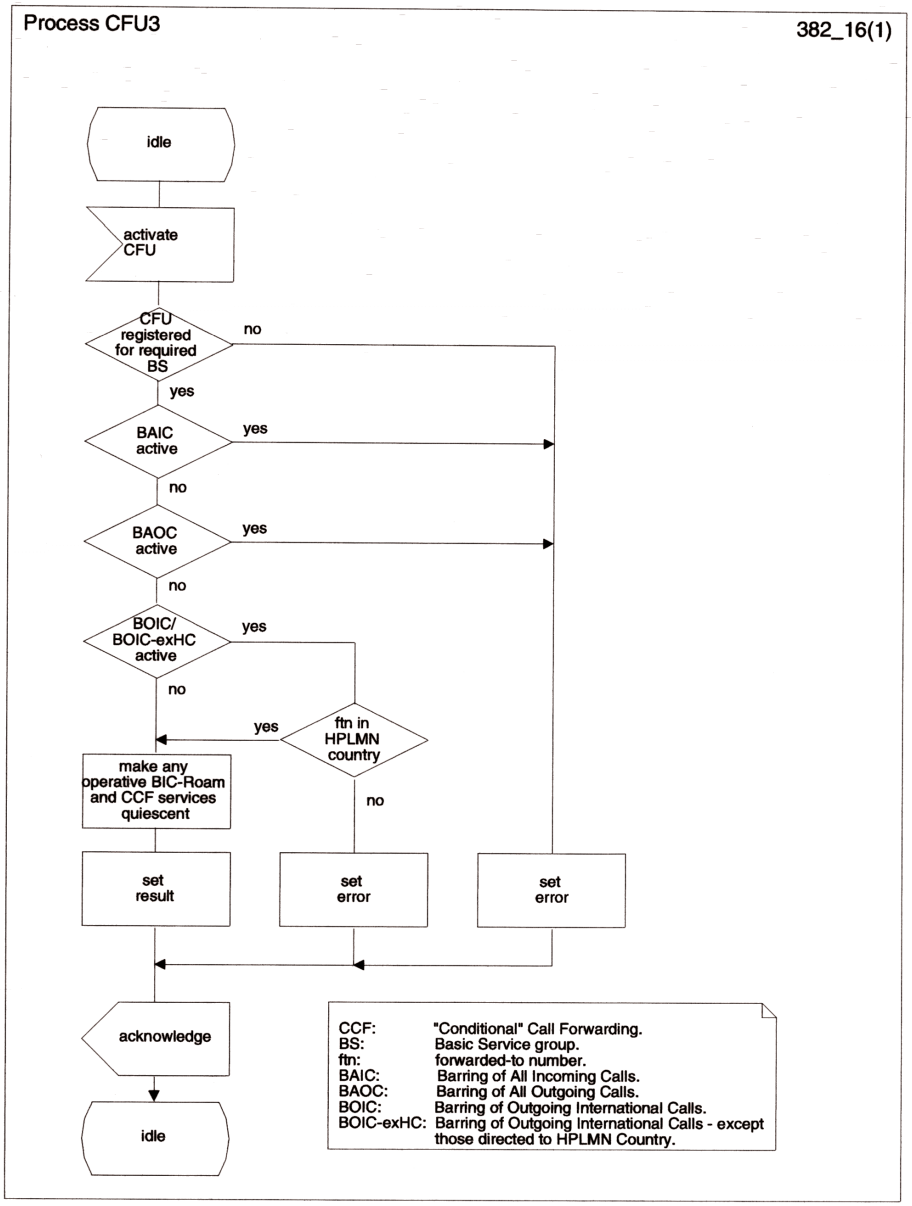


Figure 1.6: CFU3 Call forwarding unconditional activation process

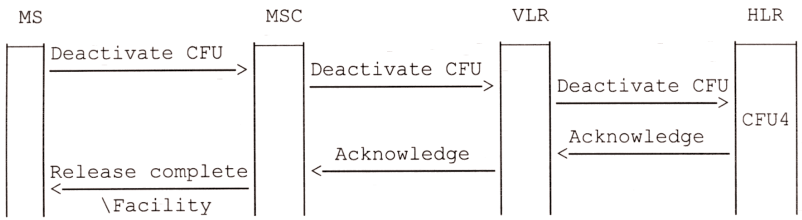


Figure 1.7: Deactivation of call forwarding unconditional

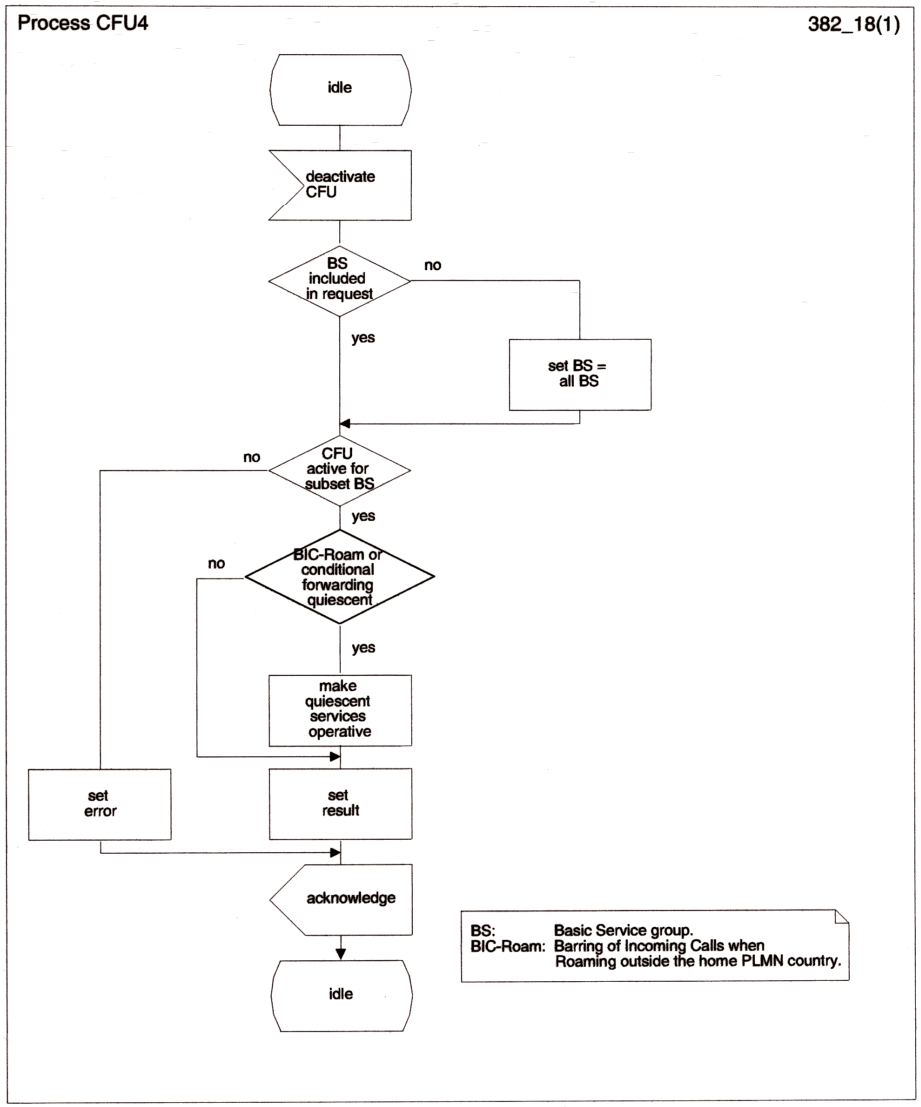


Figure 1.8: CFU4 Call forwarding unconditional deactivation process

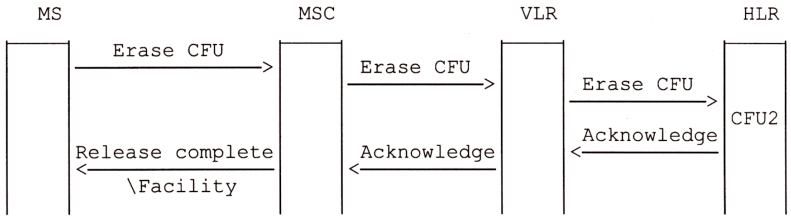


Figure 1.3: Erasure of call forwarding unconditional

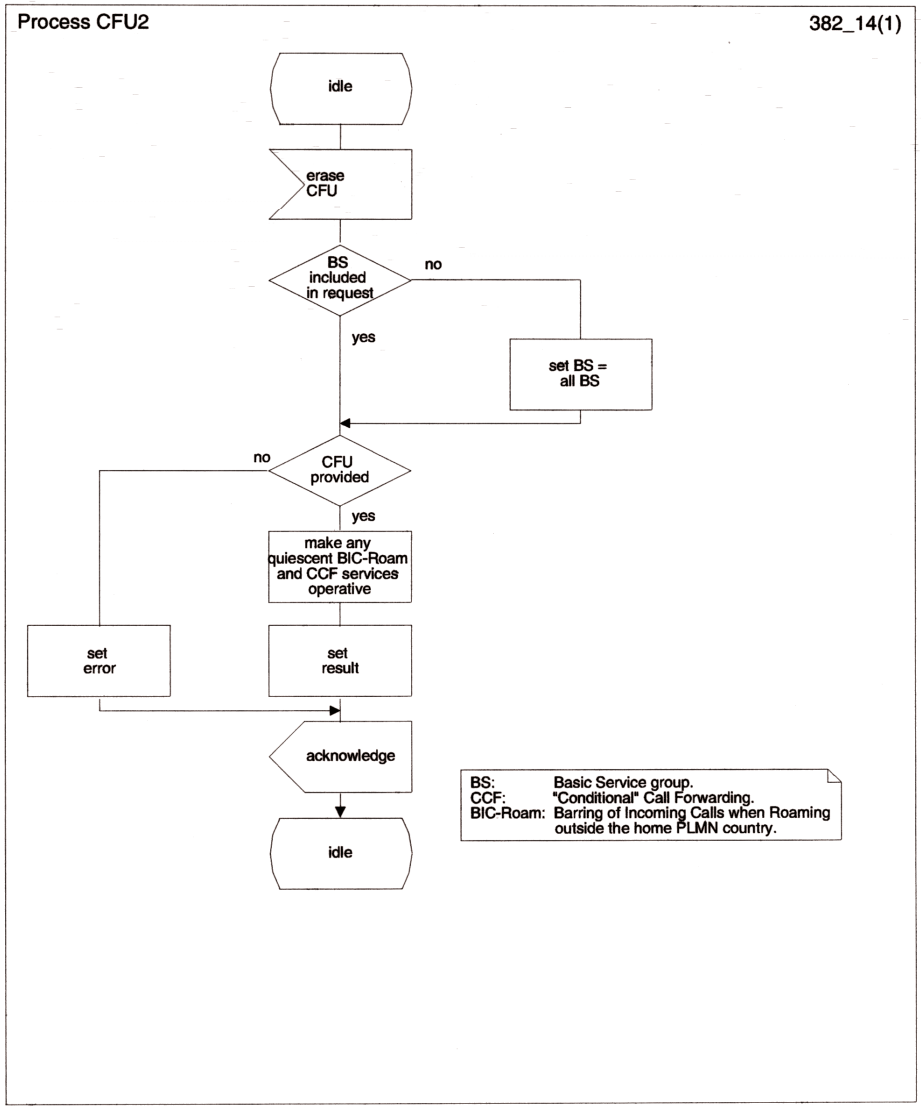
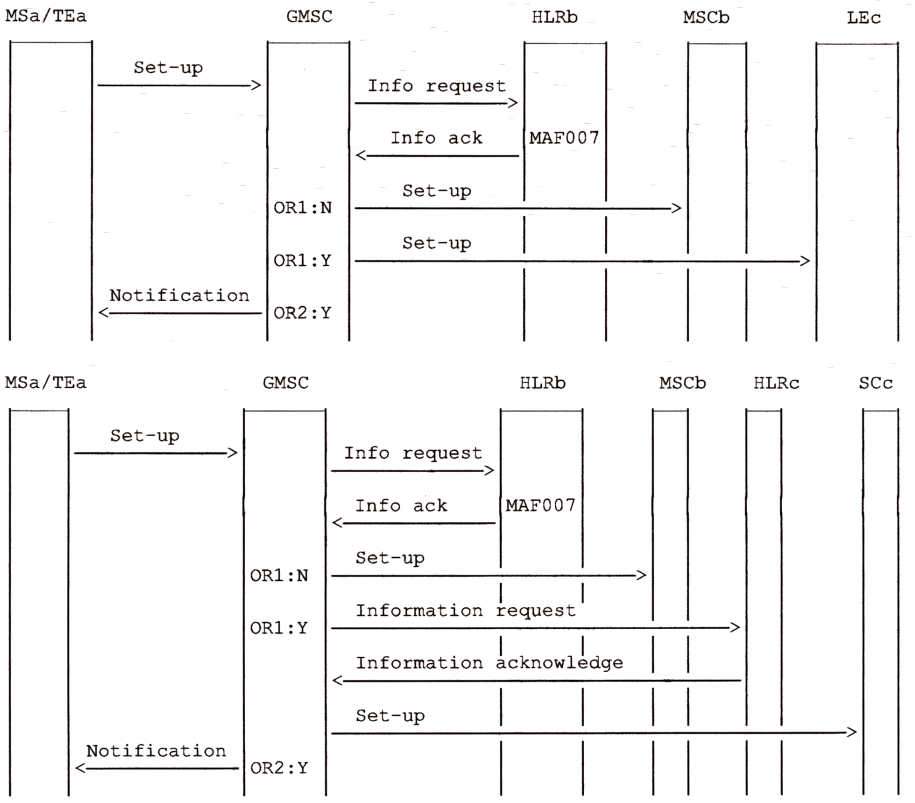


Figure 1.4: CFU2 Call forwarding unconditional erasure process



NOTE: info: information Y: Yes
req: request N: No
ack: acknowledge
OR1: Forwarding requested
OR2: Notification to calling subscriber required

Figure 1.11: Information flow for call forwarding unconditional

Beispiel eines komplexeren GSM-Zusatzdienstes: Completion of Calls to busy subscribers (CCBS)

(entnommen aus: GSM 03.93 v6.1.0 1998)

realisiert den automatischen Rückruf bei besetzt

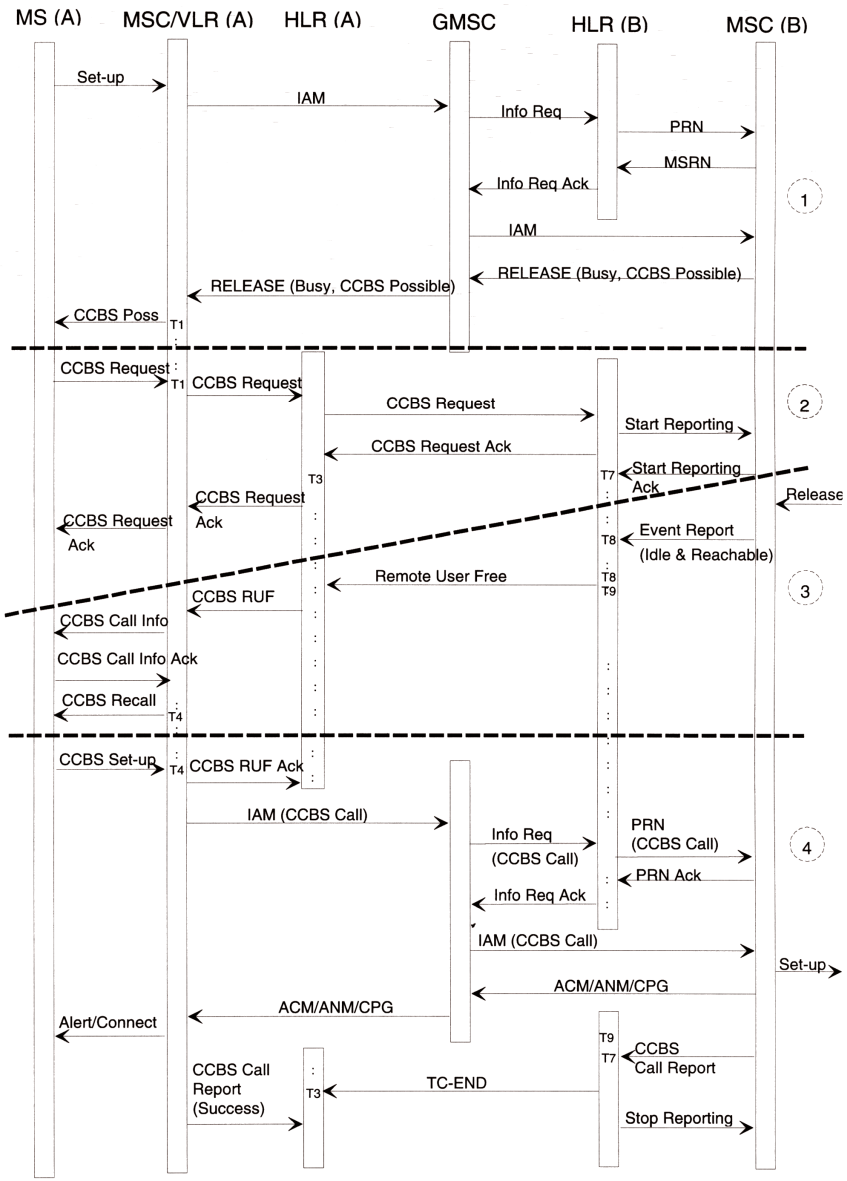
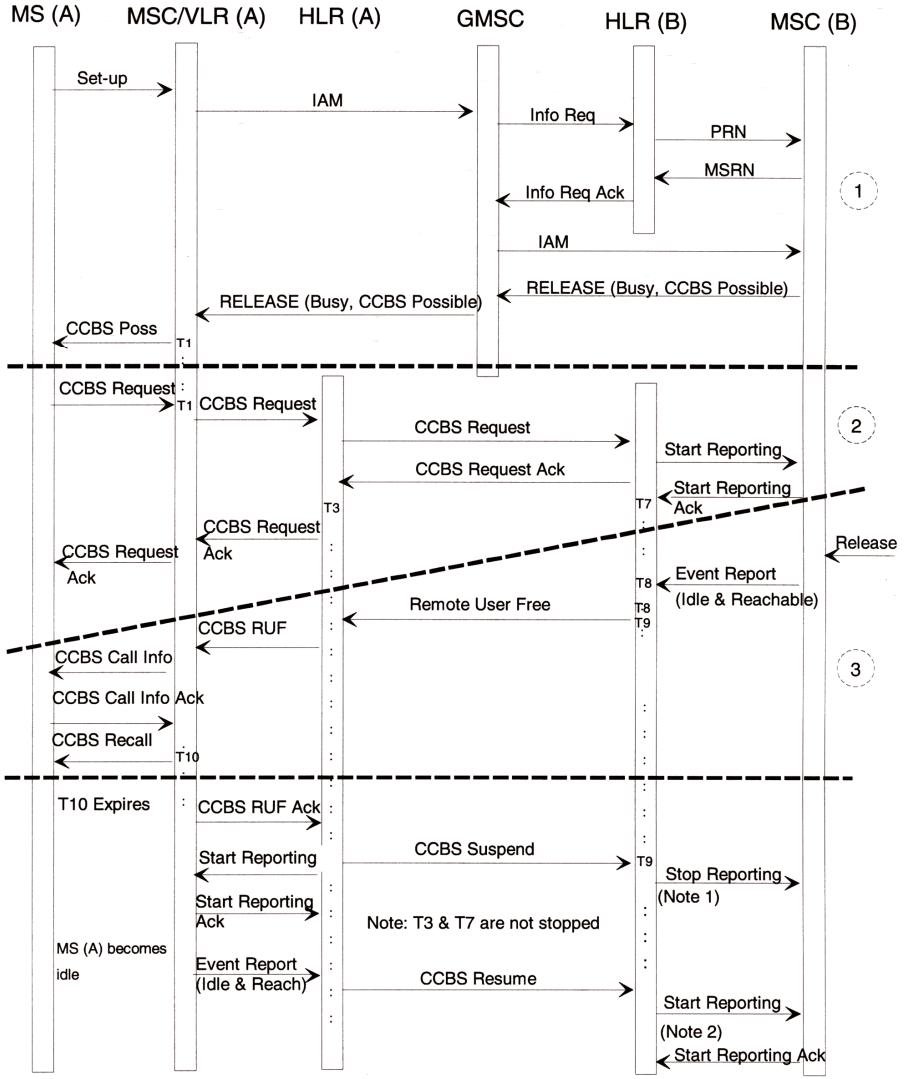


Figure 5.2.1: Successful CCBS request. Destination B busy when request activated, subscriber A free when destination B becomes free (mobile-to-mobile)



Note 1 : Stop Reporting is sent if there are only suspended requests in the queue

Note 2 : Start Reporting is sent if MSC B is not monitoring destination B

Figure 5.2.2: Subscriber A not idle when destination B becomes free (mobile-to-mobile); Processing of a single request

Die Spezifikation von CCBS im GSM umfaßt 161 Seiten, davon über 100 MSC und SDL.

An diesen Beispielen wird bereits deutlich, daß das Thema formale (präzise) Systemspezifikation einen sehr praktischen Hintergrund hat

5.1 Spezifikation

Was ist eine Spezifikation?

IEEE: "Ein Dokument, das in vollständiger, präziser und verifizierbarer Weise die Anforderungen, den Entwurf, das Verhalten oder andere Charakteristika eines Softwareproduktes beschreibt."

Bei **Dienst-** und **Protokollspezifikationen** für Kommunikationssysteme ist Präzision besonders wichtig!
Es gibt bereits einige Erfahrung im Umgang mit komplexen Spezifikationen. (z.B. in CCITT und ISO)

Teil der Erfahrung: Spezifikation ist eine **komplexe** und **fehleranfällige** Tätigkeit

Jede komplexe Spezifikation enthält **Fehler**, z.B.

- ISO Class 0 Transport Service
- CCITT X.21 Protocol
- Military Standard Transmission Control Protocol (TCP)
- ISO Session Protocol
- CCITT Signalling System No.7

Die **natürliche** Sprache ist zur präzisen Beschreibung ungeeignet:

"Lassen Sie es mich folgendermaßen erklären ..."
"Ich glaube, er hat gemeint ..."
"Nein, ich wollte sagen ..."

Die wichtigsten Probleme der **natürlichen** Sprache sind:

- oft ist keine eindeutige **Interpretation** möglich:
  "X soll zwischen Y und Z liegen."
- keine **Werkzeugunterstützung** bei der Spezifikation und der Verwendung der Spezifikation
- unkontrolliertes Springen zwischen **Abstraktionsebenen**
- **umständliche**, schwer verständliche Beschreibungen für **einfache** Sachverhalte (z.B. Beschreibung eines endlichen Automaten)

Vorteile formaler Spezifikationssprachen

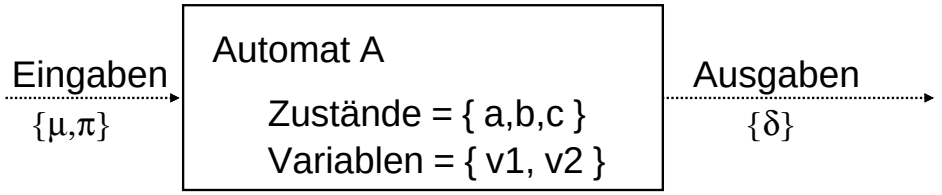
- eindeutige, präzise und vollständige Spezifikationen
- Basis für die Analyse auf Korrektheit und Vollständigkeit
- Basis für die Überprüfung auf Konformität
- Unterstützung durch Software-Werkzeuge in der Spezifikationsphase

Formale Spezifikationssprachen (FDT):

- **Estelle** (Extended State Machine Model, ISO IS 9074)
- **LOTOS** (Language Of Temporal Ordering Specifications, ISO IS 8807)
- **SDL** (Specification and Description Language, CCITT Empfehlung Z.100)

5.2. Estelle

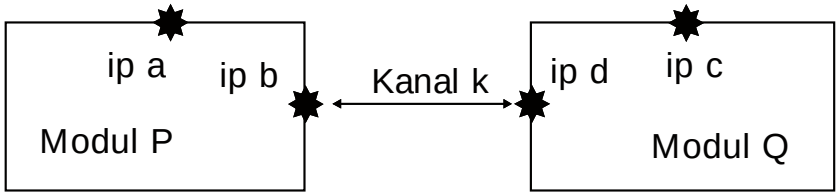
Basiert auf der Idee des erweiterten endlichen Automaten



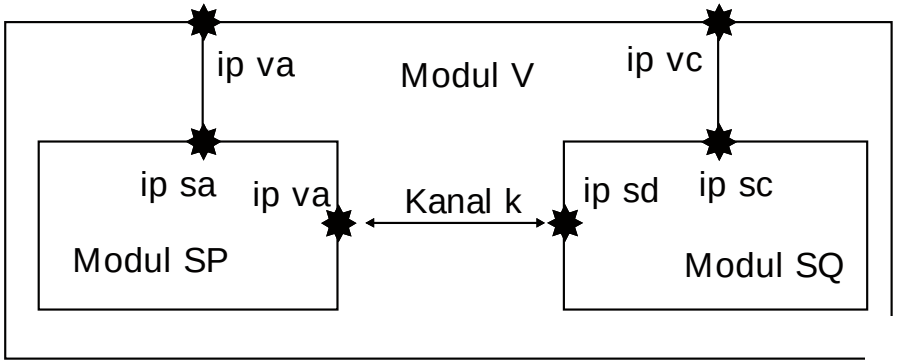
Gesamtmenge der Zustände:

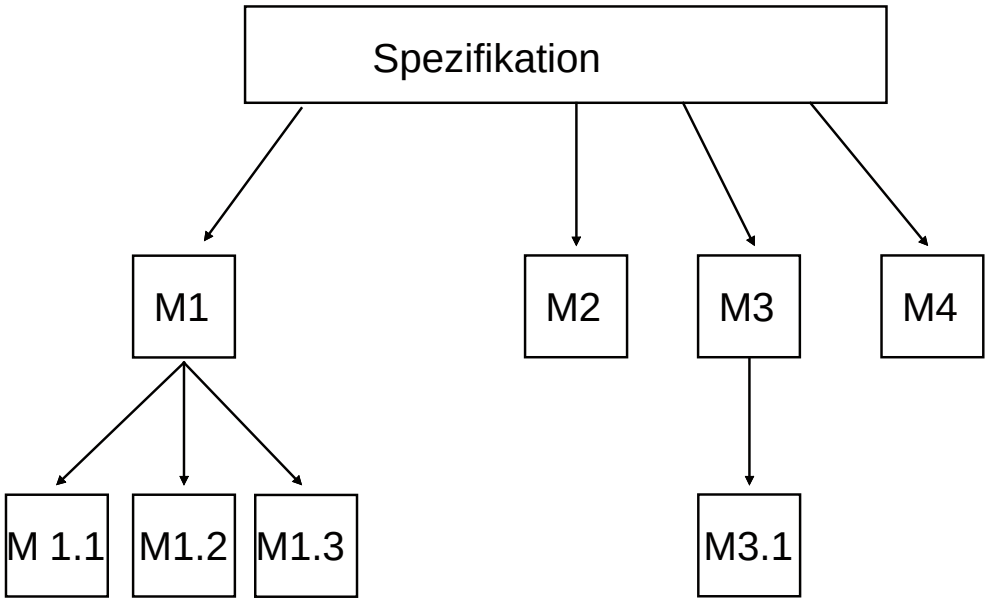
$$Z = \{a,b,c\} \times Typ(v1) \times Typ(v2)$$

- Module
- Kanäle
- Interaktionspunkte

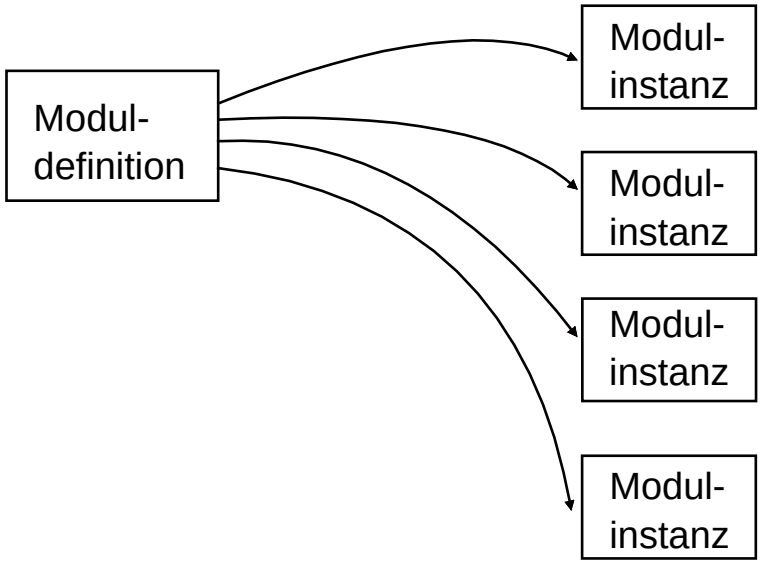


Eine Estelle-Spezifikation ist **hierarchisch** aus kommunizierenden Modulen aufgebaut

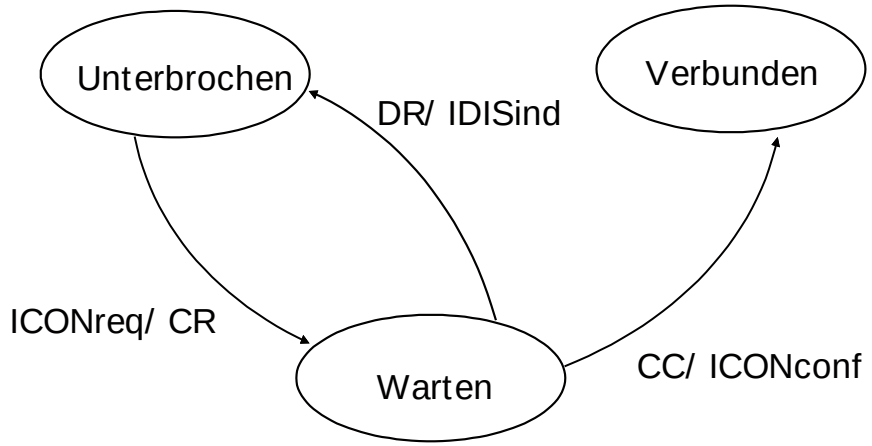




Module können dynamisch erzeugt und beendet
werden. Man muß also zwischen Moduldefinition
und Modulinstanz unterscheiden.



Zustände und Zustandsübergänge




```
state UNTERBROCHEN, WARTEN, VERBUNDEN;

trans

from UNTERBROCHEN to WARTEN
  when Benutzer.ICONreq          begin
    output PDU.CR                end;

from WARTEN to VERBUNDEN
  when PDU.CC                    begin
    output Benutzer.ICONconf     end;

from WARTEN to UNTERBROCHEN
  when PDU.DR                    begin
    output Benutzer.IDISind      end;
```


Nicht-Determinismus

```
from ZUSTAND_A to ZUSTAND_B
 when IP.signal                  begin
       {tu' etwas}               end;

from ZUSTAND_A to ZUSTAND_C
 when IP.signal                  begin
       {tu' etwas anderes}       end;
```

Spontane Transition

```
from ZUSTAND_A to ZUSTAND_B      begin
       {tu' etwas}               end;
```

Definition und Gebrauch von Daten

```
type ISDUTyp = ... ;
var alteISDU : ISDUTyp;
    Zaehler : 0..4;

from VERBUNDEN to SENDEND
  when BENUTZER.IDATreq(ISDU)   begin
    output PDU.DT(ISDU);
    alteISDU := ISDU            end;

from SENDEND
  when PDU.AK(ISDU)
    provided ISDU = alteISDU
      to VERBUNDEN              begin
        Zaehler := 0            end;
    provided otherwise
      to same;

from SENDEND
  delay (5)
    provided Zaehler < 4
      to same                   begin
        output PDU.DT(alteISDU);
        Zaehler := Zaehler + 1  end;
    provided otherwise
      to UNTERBROCHEN           begin
        output PDU.DR;
        output BENUTZER.IDISind  end;
```


5.3 LOTOS

Grundlegende Idee von LOTOS:

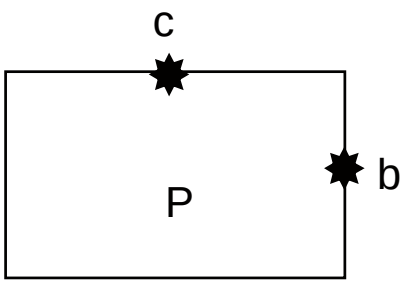
Einen Prozeß durch sein beobachtbares Verhalten definieren.

Diese Sicht ist konsistent mit einer allgemeineren Forderung im Software Engineering:

Die Spezifikation von Anforderungen an ein System soll implementierungsunabhängig sein.

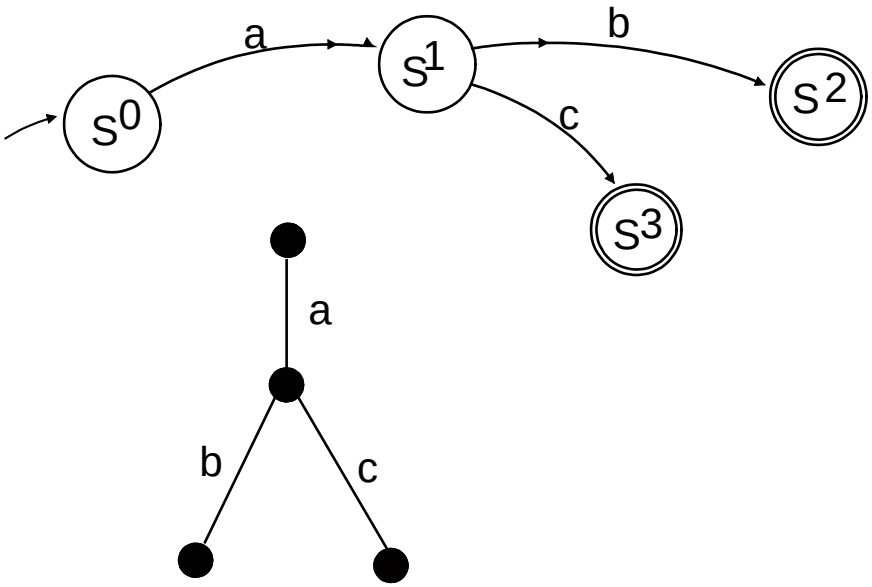
ISO: "Only the external behaviour of Open System is retained as the standard behaviour of real Open Systems."

Das beobachtbare Verhalten besteht aus einer Sequenz von Ereignissen am "Rand" eines Prozesses.



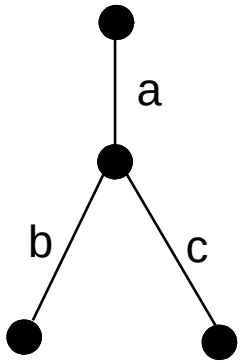
LOTOS ist eine Sprache zur Beschreibung von Ereignissequenzen.

Das beobachtbare Verhalten eines Prozesses

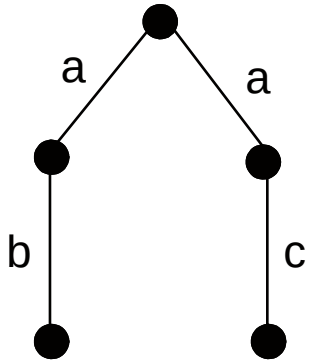


Die bekanntesten Theorien um das beobachtbare Verhalten:

- CCS (Milner, 1980)
- CSP (Hoare, 1978)



Prozess P



Prozess Q

P und Q sind:

- identisch im Sinne der **trace-equivalence** (erkannte Wortmenge {ab,ac})
- nicht identisch im Sinne der **observation-equivalence**

trace equivalence
failure equivalence
ready equivalence
ready trace equivalence
observation equivalence
observation congruence
strong congruence

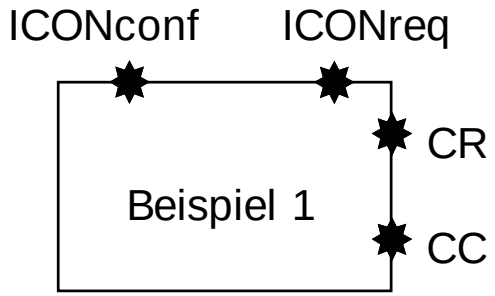
Verhaltensausdrücke und Ereignissequenzen

Inaktives Verhalten

`stop`

Action-Prefix

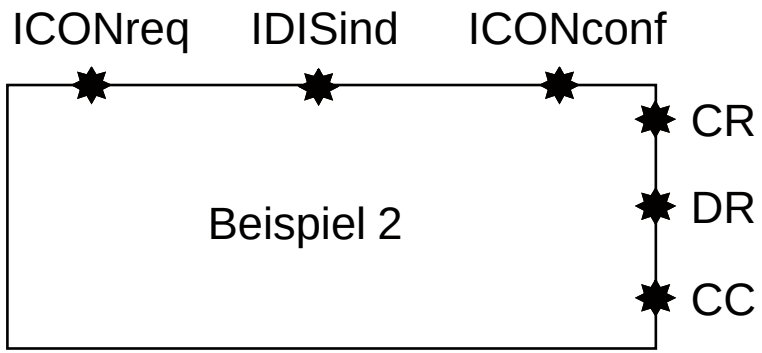
`a;B`




```
process Beispiel1
[ICONreq,CR,CC,ICONconf] :=
   ICONreq;CR;CC;ICONconf; stop
endproc
```

Auswahloperator

```
P [ ] Q
```




```
process Beispiel2
[ICONreq,IDISind,CR,DR,
   CC,ICONconf] :=
   ICONreq;CR;(CC;ICONconf;stop
      [ ]DR;IDISind;stop)
endproc
```


Verhaltensrekursion

```
process Beispiel 3   [ICONreq,IDISind,
       CR,DR,CC,ICONconf]  :=
    ICONreq;CR;(CC;ICONconf;DR;IDISind;Beispiel3
       []DR;IDISind;Beispiel3)
endproc
```

Nicht-Determinismus

```
   a;b;stop
[] a;c;stop

   a;b;stop
[] i;c;d;stop
```

```
process Beispiel4 [ICONreq,IDISind,CR,DR,
       CC,ICONconf] :=
    ICONreq;CR;(CC;ICONconf;DR;IDISind;Beispiel4
    []DR;IDISind;Beispiel4
    []i;IDISind;Beispiel4)
endproc
```

Sequentielle Komposition

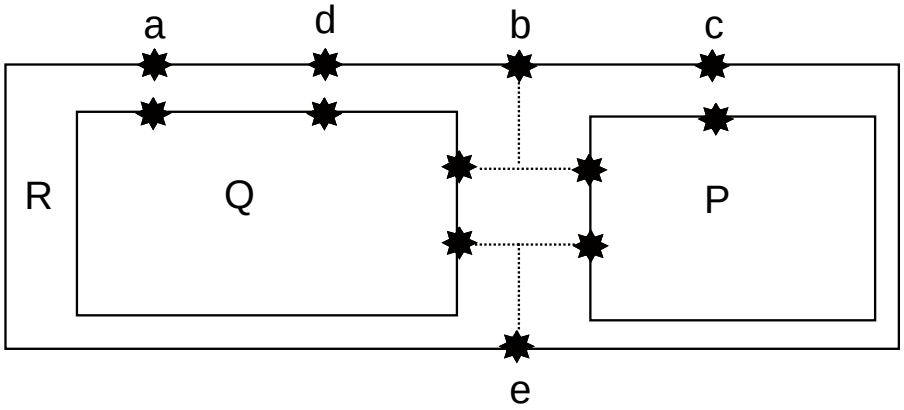
```
a;b;c;Prozeß1
Prozeß1>>Prozeß2
```

Unterbrechung

```
P [> Q
```

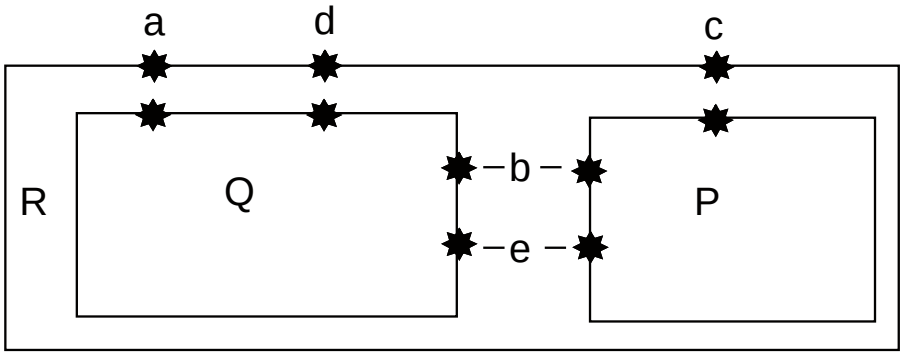
Parallelität

```
P |[a1,...,an]| Q
```




```
process R[a,b,c,d,e] :=
     Q[a,b,d,e]|[b,e]|P[b,c,e]
endproc
```


Der hiding-Operator




```
process R[a,b,c,d,e] :=
hide b,e in
     Q[a,b,d,e]|[b,e]|P[b,c,e]
endproc
```

Zusätzlich zu den bisher vorgestellten Sprachelementen gibt es in LOTOS auch die Möglichkeit komplexe Datenstrukturen zu definieren. Hierfür steht das Konzept der ADT zur Verfügung.

Semantik von LOTOS

z.B. Action-Prefix

a;B -a-> B

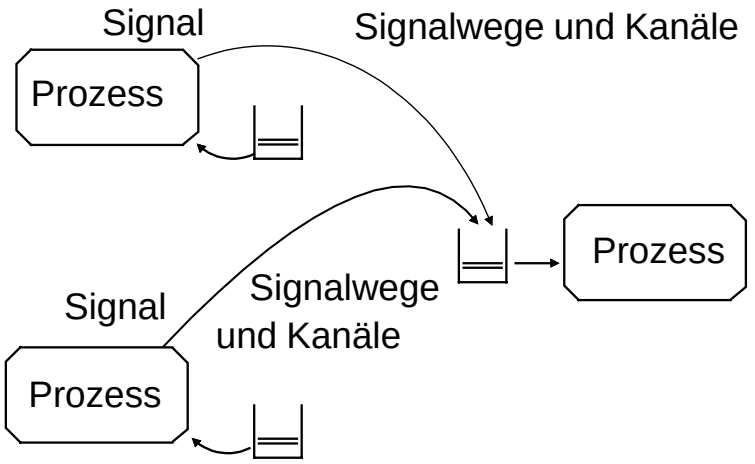
z.B. Auswahloperator

B1 -g-> B1'	B2 -g-> B2'
----------	----------
B1[]B2 -g-> B1'	B1[]B2 -g-> B2'

5.4 Die Sprachelemente von SDL

Basiskonzepte

Das Grundelement in SDL ist der Prozeß

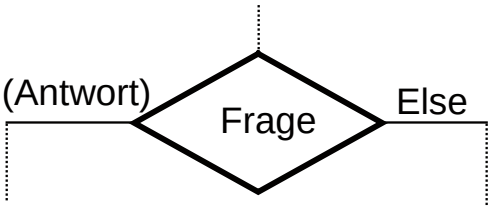


Charakterisierung eines Prozesses:

- ein Prozeß befindet sich entweder gerade in einem Zustandsübergang oder wartet auf eine Eingabe;
- es gibt eine Eingabewarteschlange für jeden Prozeß;
- wenn zwei Eingaben einen Prozeß gleichzeitig erreichen, werden sie in zufälliger Reihenfolge in der Eingabewarteschlange gepuffert.

Die zwei syntaktischen Formen SDL/GR und SDL/PR

SDL/GR



SDL/PR

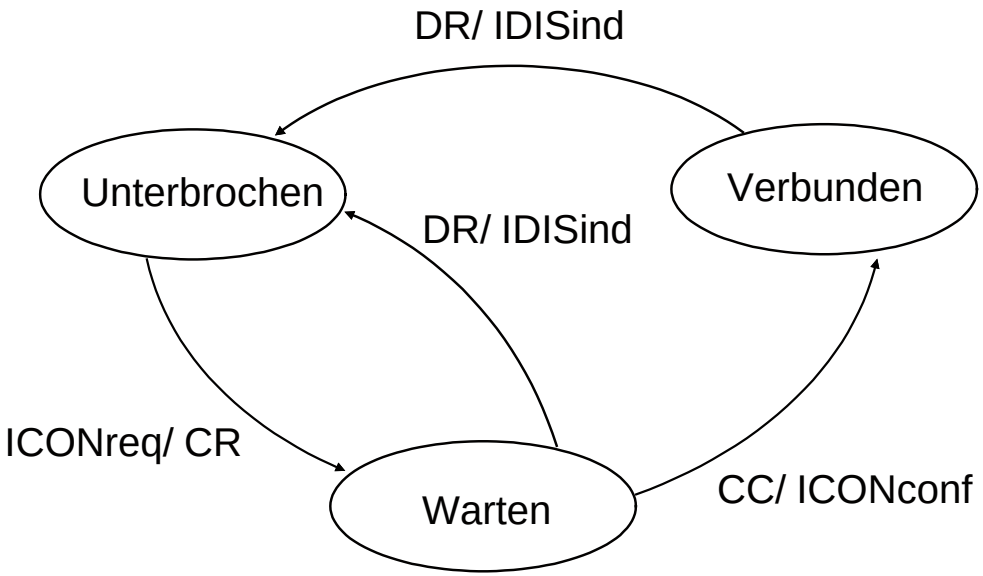
```
DECISION   Frage
  (Antwort): .....
   ELSE      : .....
ENDDECISION;
```

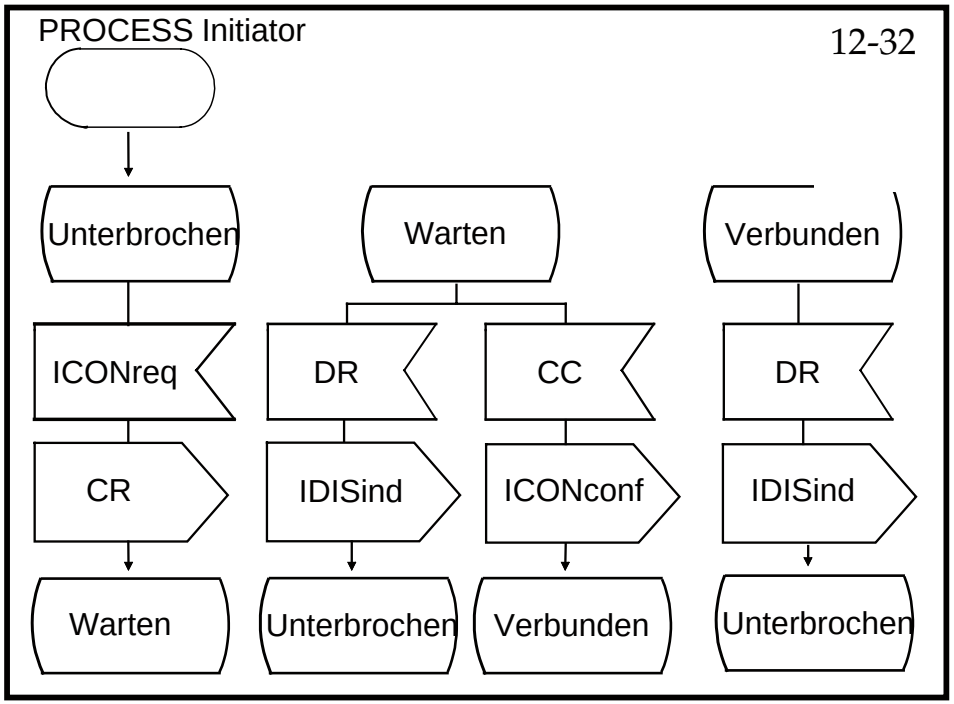
SDL/GR

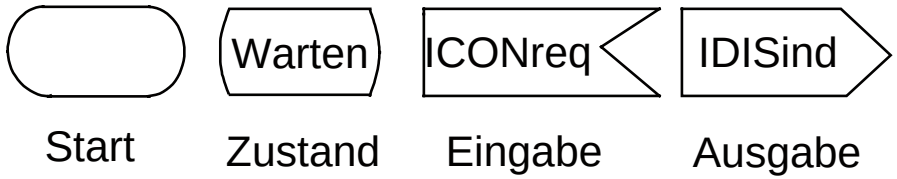
SDL/PR

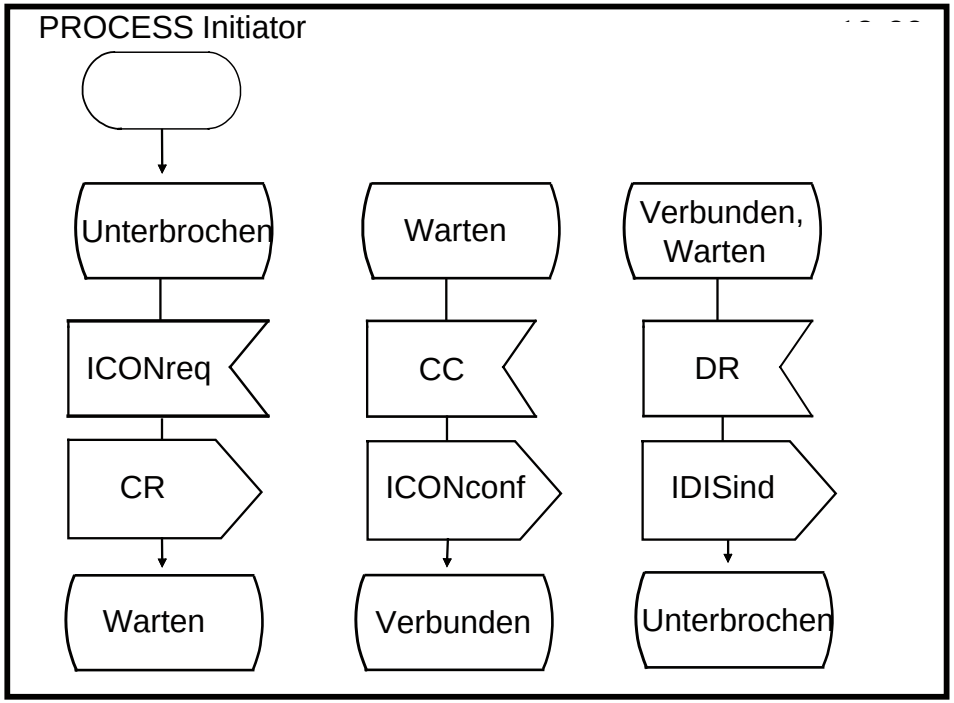
```
PROCESS Empfänger (1,1);
  REFERENCED;
```

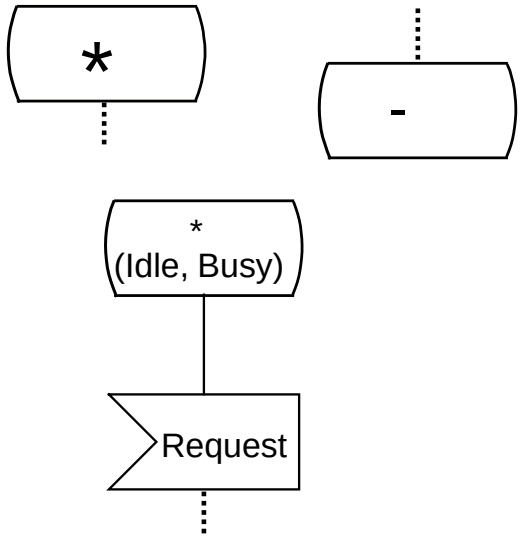
Zustände und Zustandsübergänge



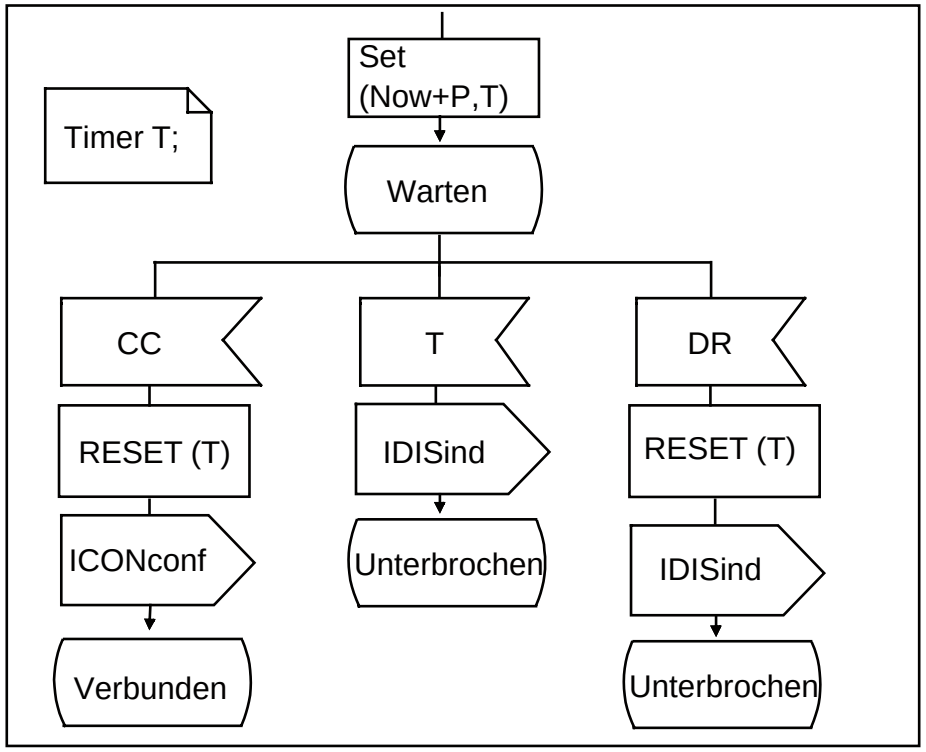


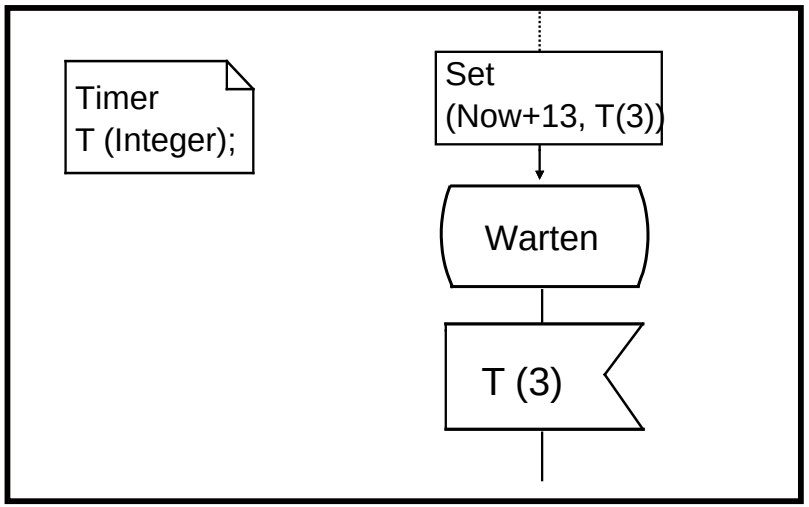



PROCESS Initiator
Unterbrochen
Warten
Verbunden,
Warten
ICONreq
CC
DR
CR
ICONconf
IDISind
Warten
Verbunden
Unterbrochen

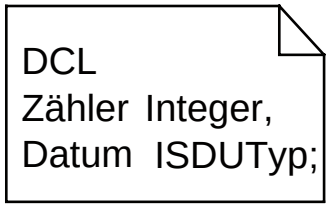

*
-
*
(Idle, Busy)
Request

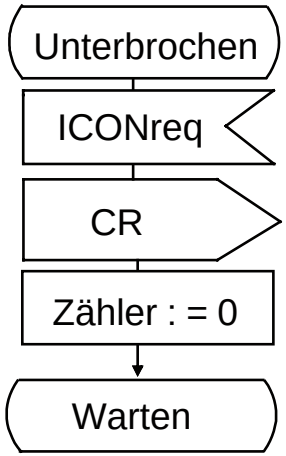
Zeit in SDL


Timer T;
Set
(Now+P,T)
Warten
CC
T
DR
RESET (T)
IDISind
RESET (T)
ICONconf
Unterbrochen
IDISind
Verbunden
Unterbrochen

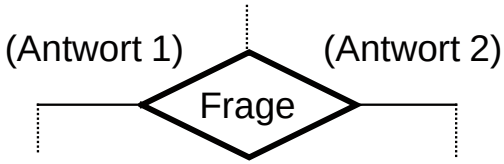

Timer
T (Integer);
Set
(Now+13, T(3))
Warten
T (3)

Deklaration und Gebrauch von Daten

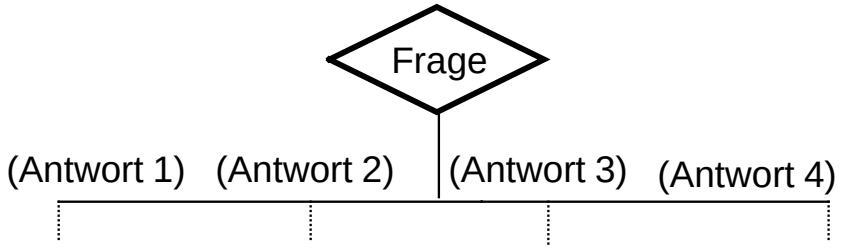




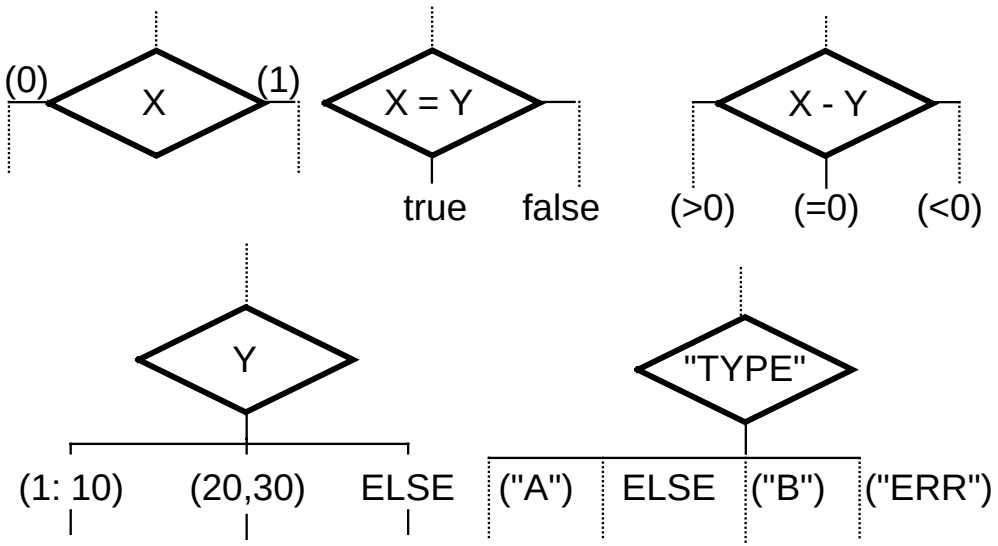
Die Alternative

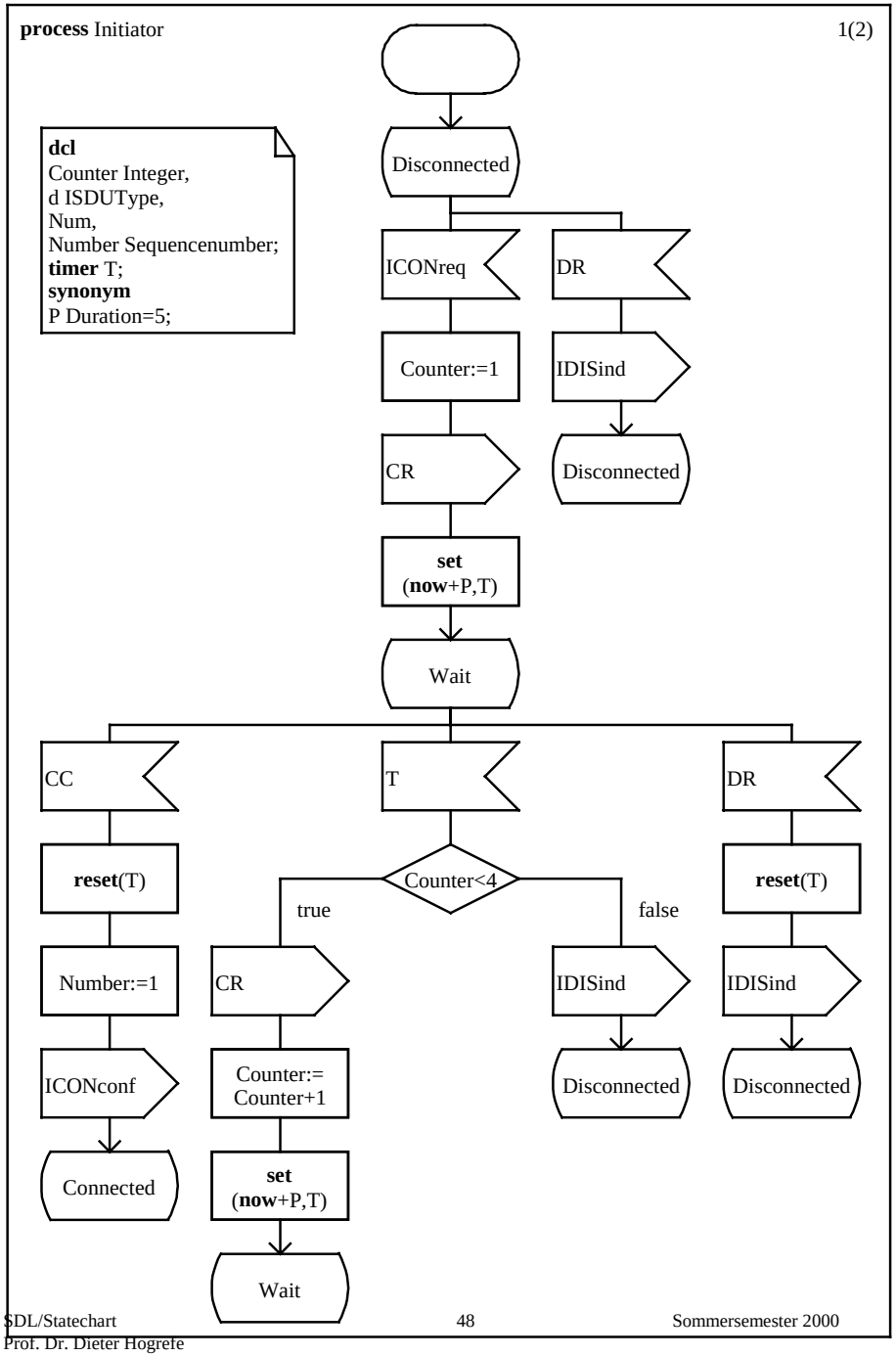


einfache Alternative

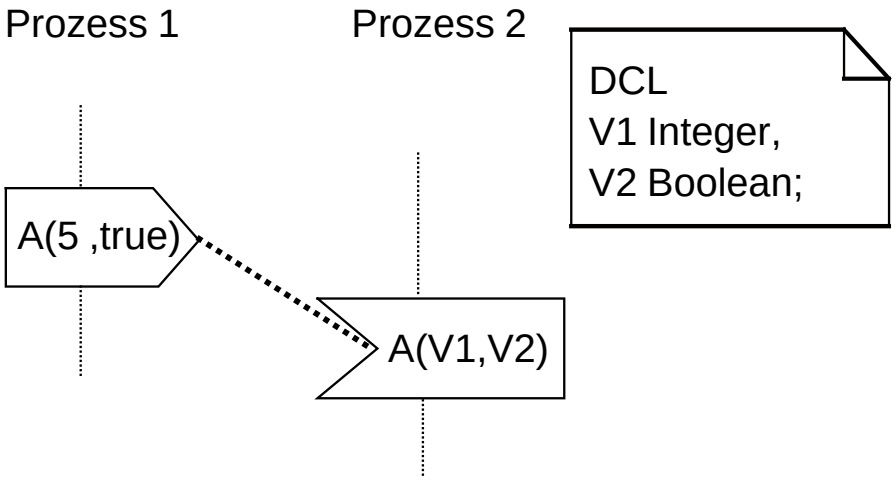


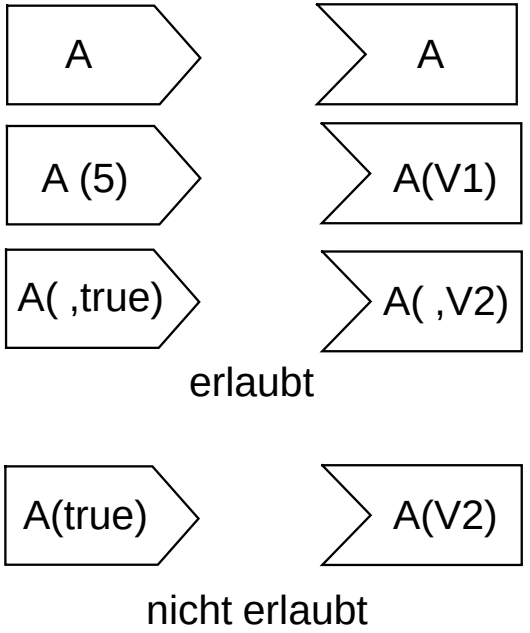
mehrfache Alternative


(0)
X
(1)
X = Y
true
false
X - Y
(>0)
(=0)
(<0)
Y
(1: 10)
(20,30)
ELSE
"TYPE"
("A")
ELSE
("B")
("ERR")

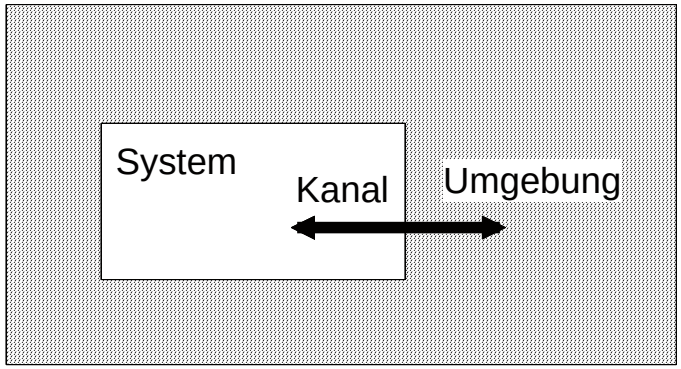

process Initiator
1(2)
dcl
Counter Integer,
d ISDUType,
Num,
Number Sequencenumber;
timer T;
synonym
P Duration=5;
Disconnected
ICONreq
DR
Counter:=1
IDISind
CR
Disconnected
set
(now+P,T)
Wait
CC
T
DR
reset(T)
Counter<4
reset(T)
true
false
Number:=1
CR
IDISind
IDISind
ICONconf
Counter:=
Counter+1
Disconnected
Disconnected
Connected
set
(now+P,T)
Wait

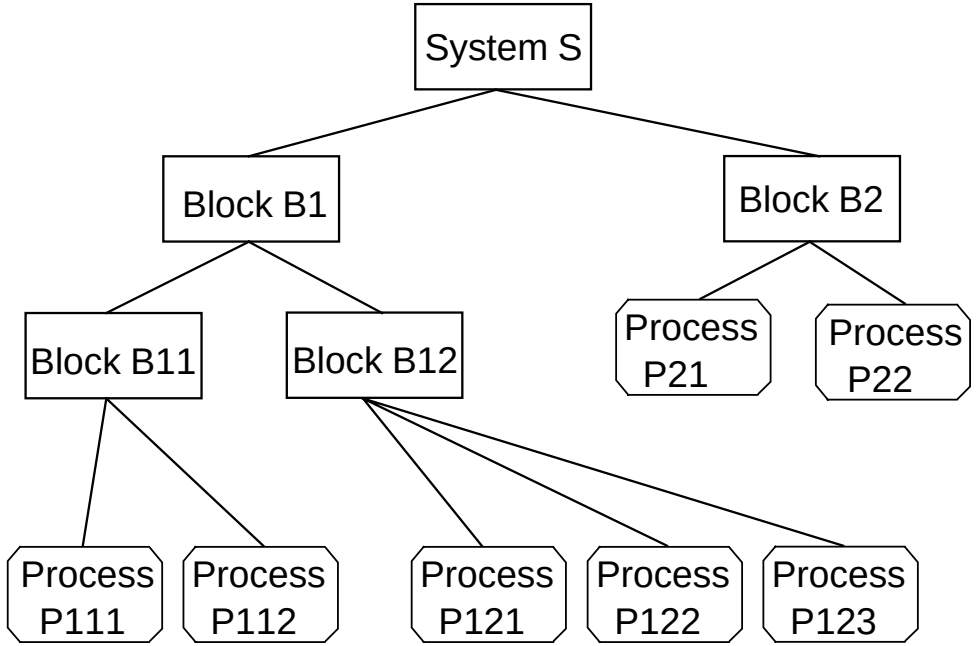
Signale mit Daten





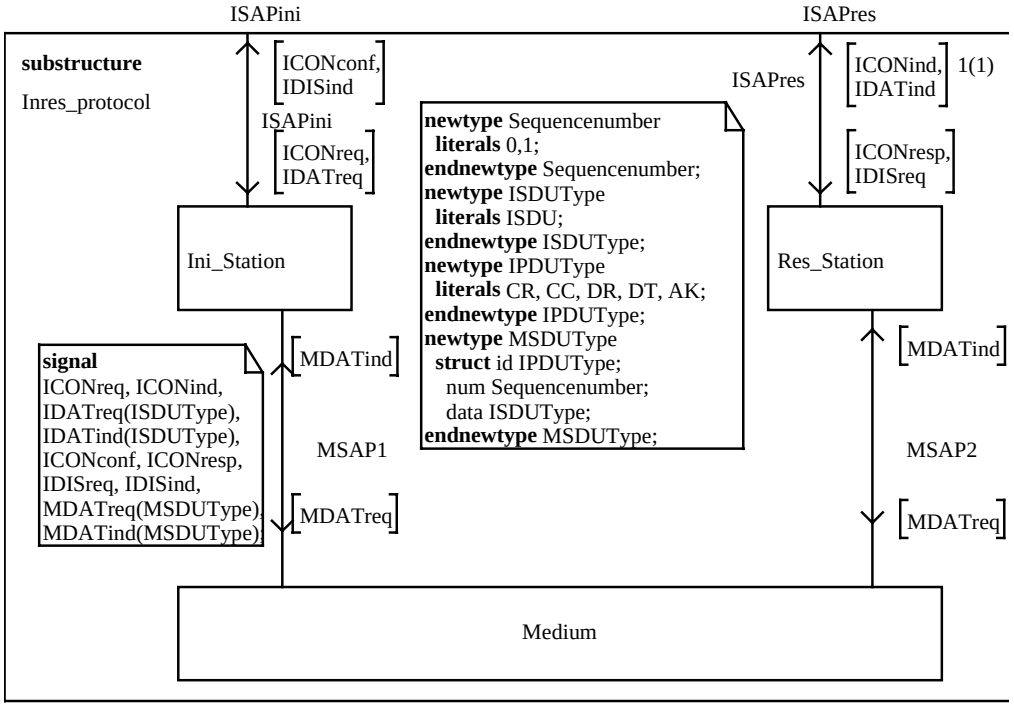
Strukturierung und Prozeßkommunikation

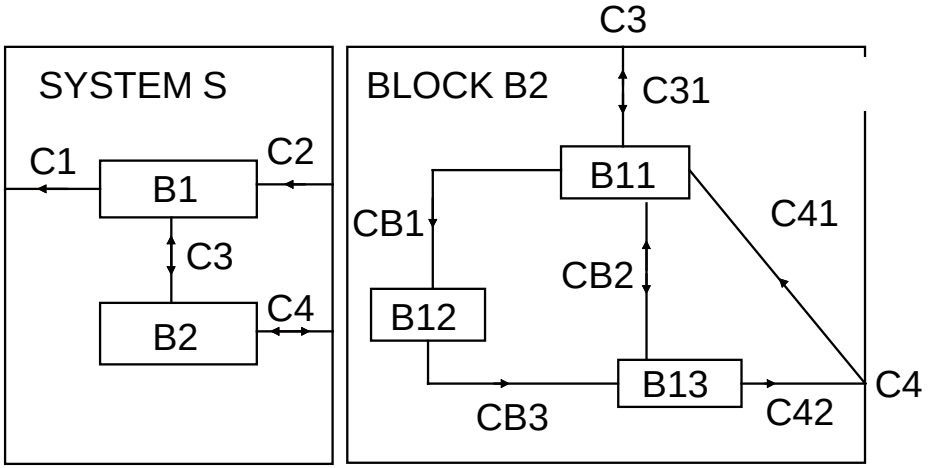




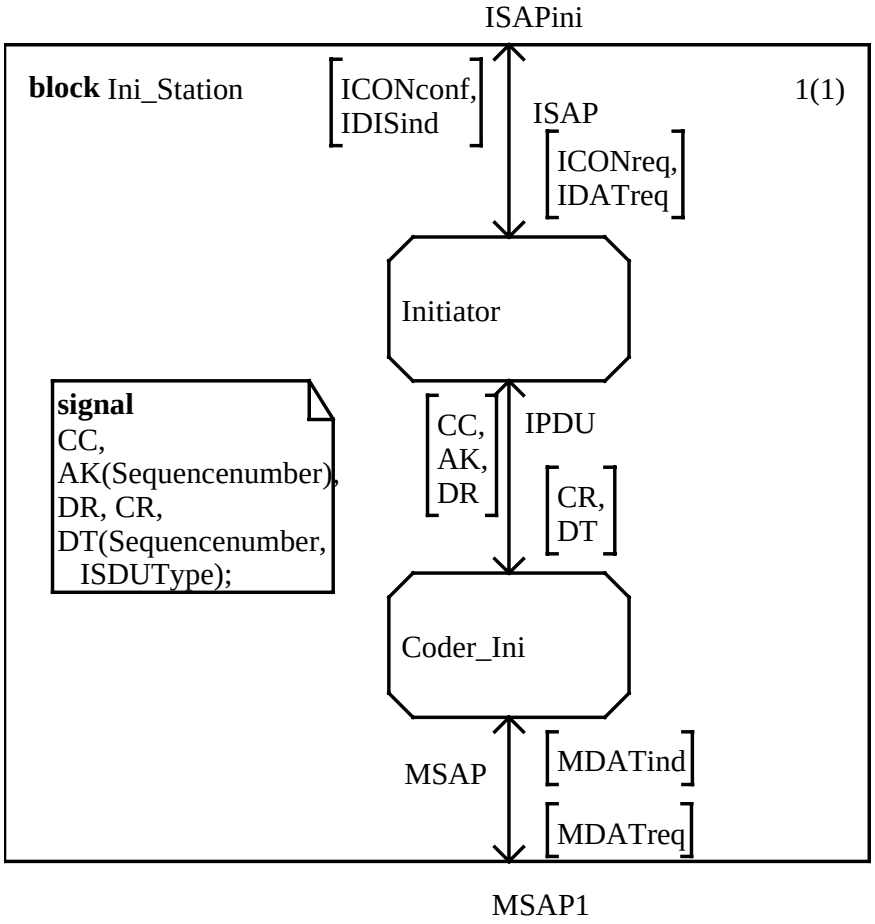
- Prozeßdiagramme
- Prozeßinteraktionsdiagramme
- Blockinteraktionsdiagramme

Blockinteraktionsdiagramme





Prozeßinteraktionsdiagramme



Prozeßkommunikation und Adressierung

Ein Ziel kann auf unterschiedliche Weise spezifiziert werden:

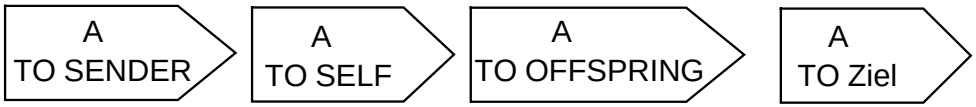
- durch direkte Angabe einer Zieladresse

- durch indirekte Adressierung, indem das Ziel eindeutig aus der Systemstruktur hervorgeht,
- durch Angabe eines Kanals oder Signalwegs, über den gesendet werden soll.

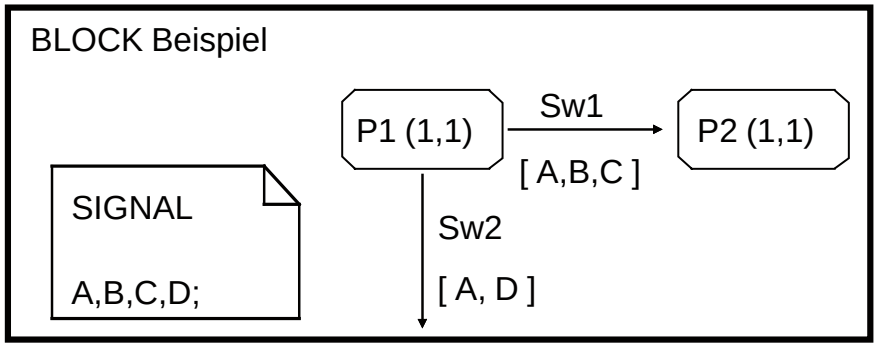
Adresse eines Prozesses**:**

- *SELF*: dieser Ausdruck liefert die eigene Adresse;
- *SENDER*: dieser Ausdruck liefert die Adresse desjenigen Prozesses, von dem zuletzt eine Eingabe erhalten wurde;
- *OFFSPRING*: dieser Ausdruck liefert die Adresse derjenigen Prozeßinstanz, die zuletzt von der Prozeßinstanz dynamisch erzeugt wurde;
- *PARENT*: dieser Ausdruck liefert die Adresse derjenigen Prozeßinstanz, durch die die Prozeßinstanz dynamisch erzeugt wurde;

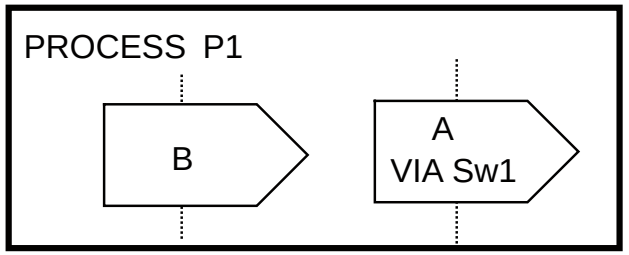
Direkte Adressierung



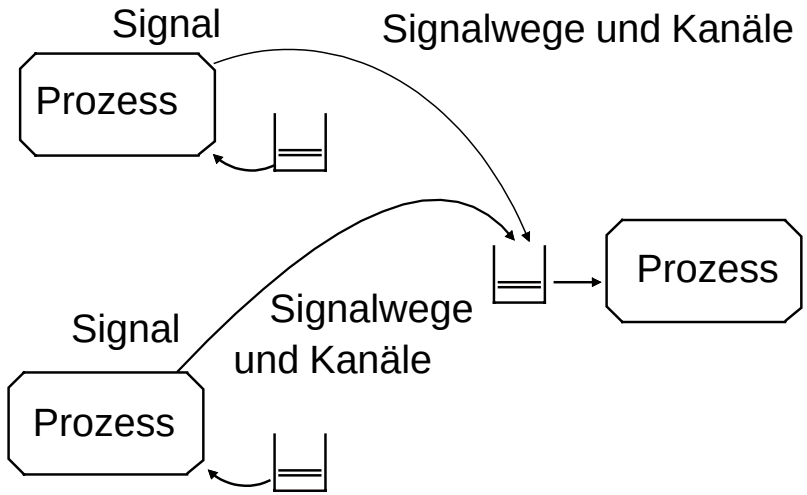
Indirekte Adressierung und Addressierung durch Angabe eines Wegs

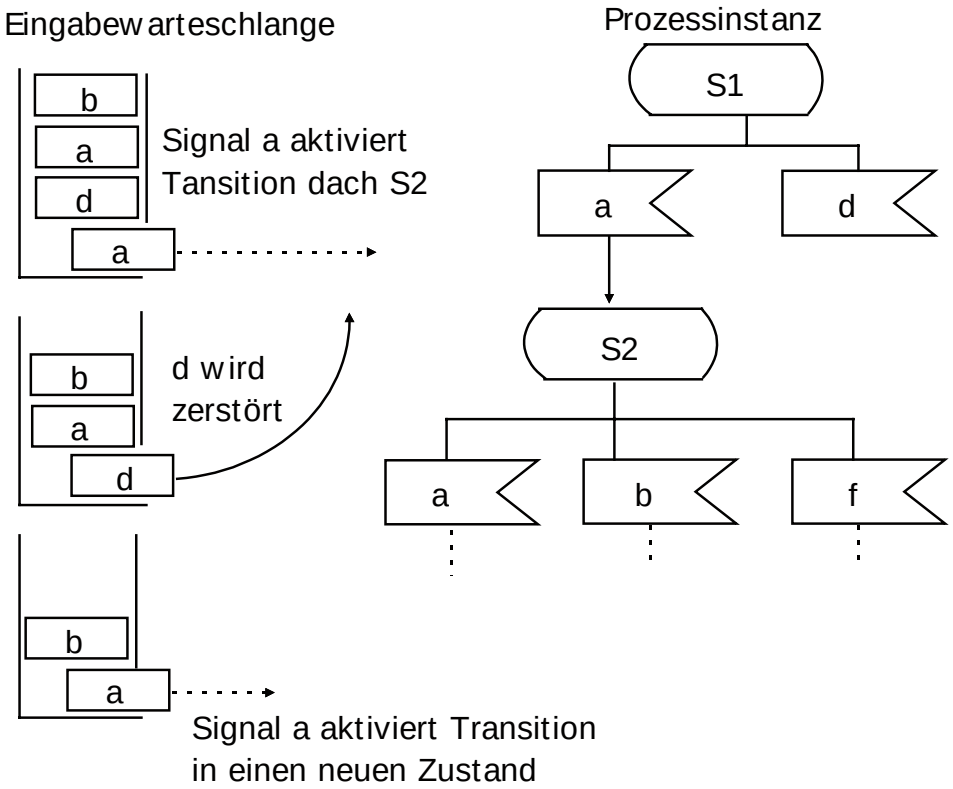


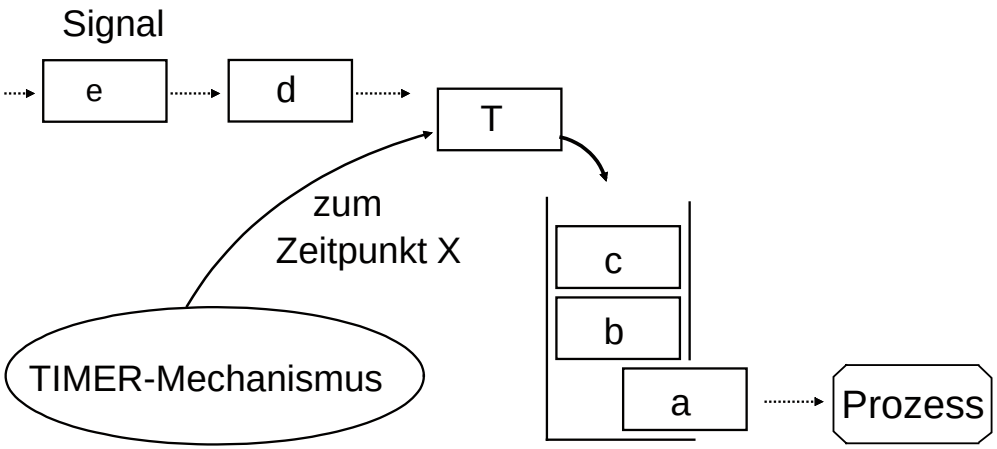
Kanal 1



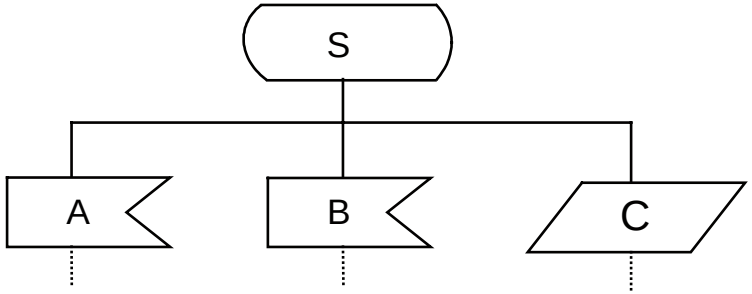
Semantik der Prozeßkommunikation

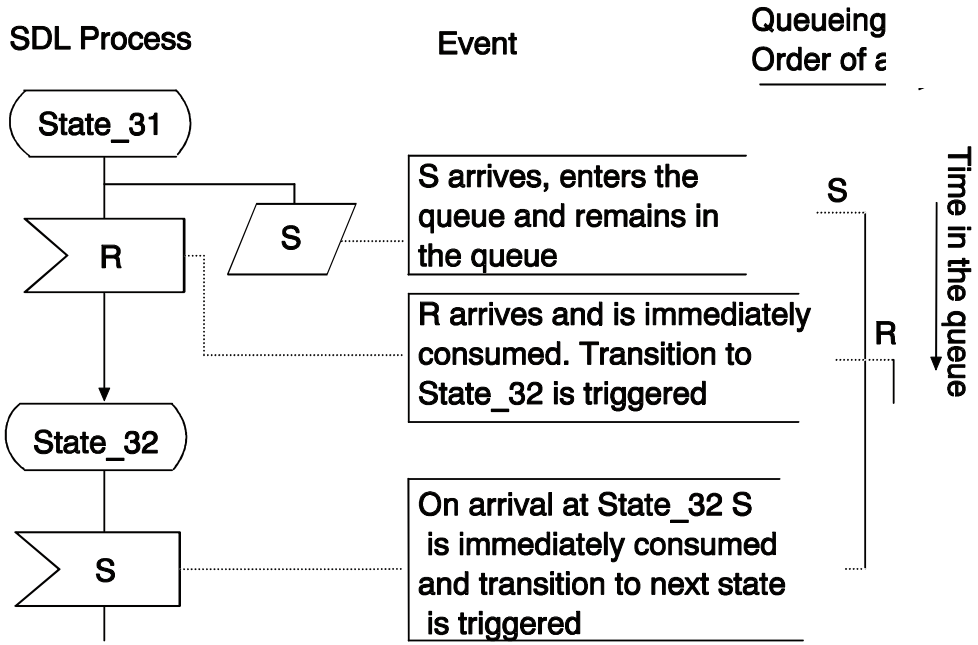


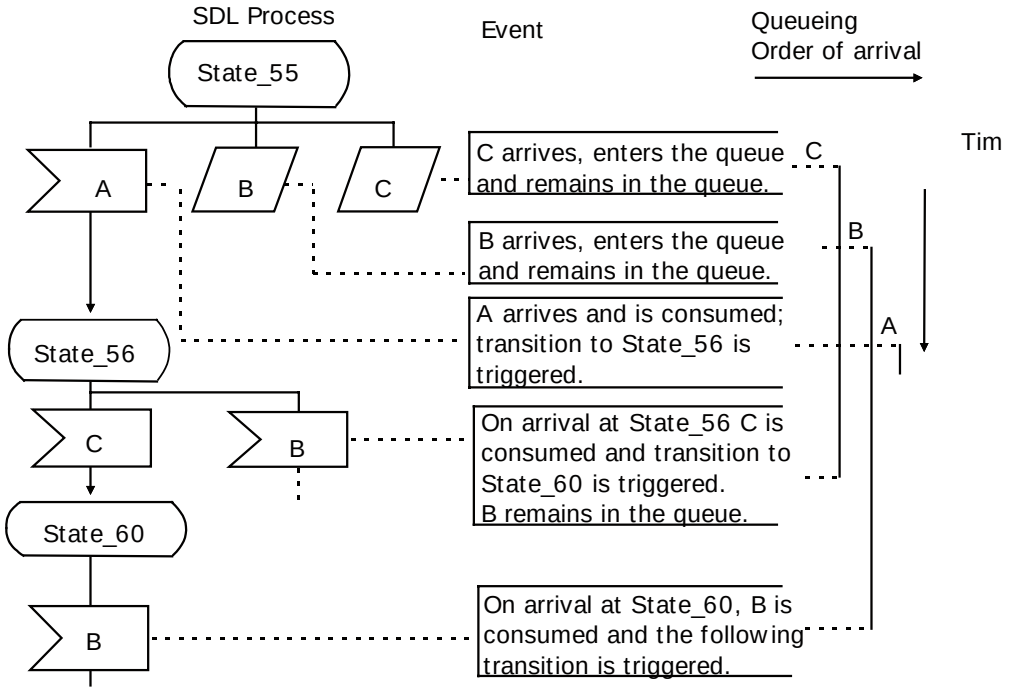


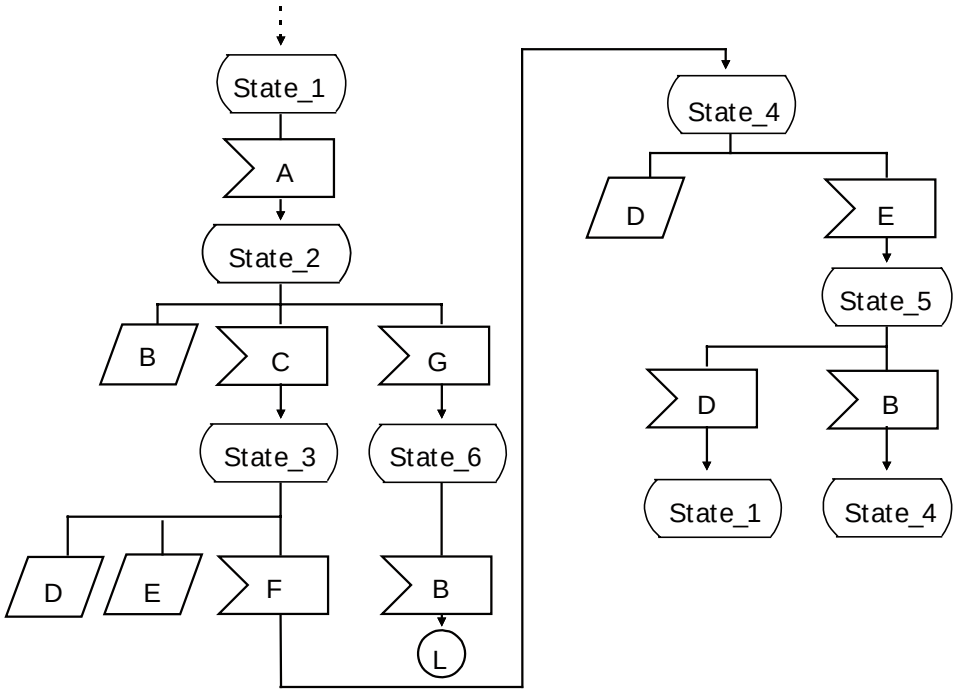


Der SAVE-Mechanismus

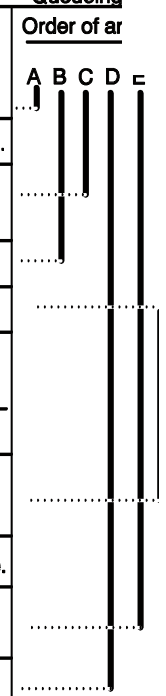



SDL Process
Event
Queueing
Order of a
State_31
R
S
State_32
S
S arrives, enters the queue and remains in the queue
R arrives and is immediately consumed. Transition to State_32 is triggered
On arrival at State_32 S is immediately consumed and transition to next state is triggered
S
R
Time in the queue

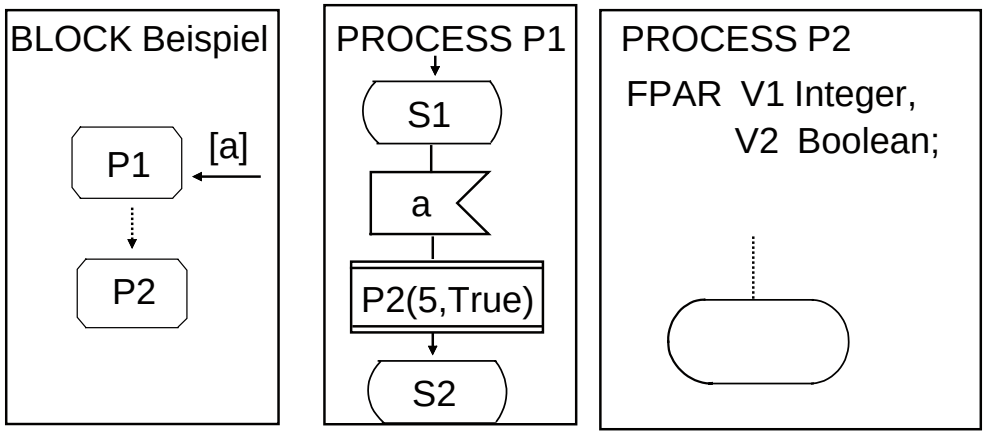

SDL Process
Event
Queueing
Order of arrival
Tim
State_55
A
B
C
C arrives, enters the queue and remains in the queue.
B arrives, enters the queue and remains in the queue.
A arrives and is consumed; transition to State_56 is triggered.
State_56
C
B
On arrival at State_56 C is consumed and transition to State_60 is triggered.
B remains in the queue.
State_60
B
On arrival at State_60, B is consumed and the following transition is triggered.
C
B
A

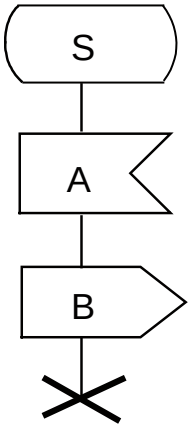


Curr. state	Event
State_1	(Process arrives at State_1 with signals A, B, C, D, E in queue) The first signal in queue, A, is consumed and transition to State_2 triggered.
State_2	The first signal in queue, B, appears in a save symbol and remains in queue.
State_2	The second signal, C, is consumed (explicit input) and transition to State_3 is triggered.
State_3	The first Signal in queue, B, is consumed (implicit input).
	Signal F arrives and enters queue.
State_3	(On reaching State_3 again) the first signal in queue, D, appears in a save symbol and remains in queue.
State_3	The second signal, E, appears in a save symbol and remeins in queue.
State_3	The third signal, F, is consumed (explicit input) and transition to State_4 is triggered.
State_4	The first signal in queue, D, appears in a save symbol and remains in queue.
State_4	The second signal, E, is consumed (explicit input) and transition to State_5 is triggered.
State_5	The first (and only) signal in queue, D, is consumed (explicit input) and transition to State_1 is triggered.

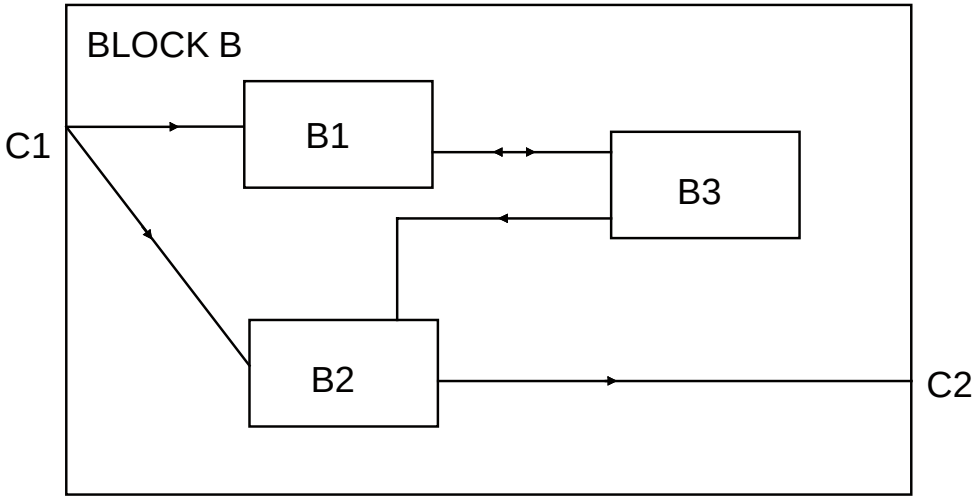


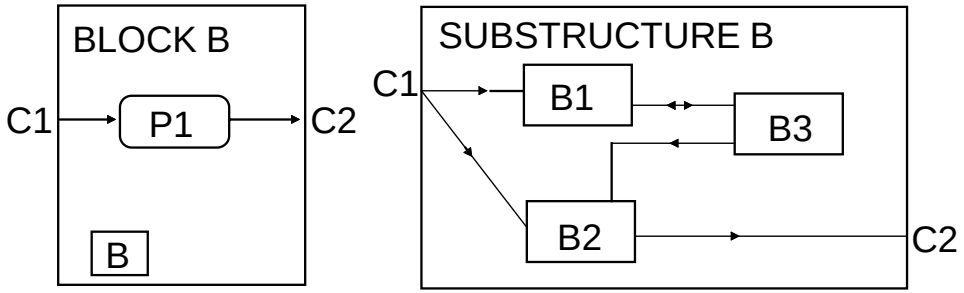
Dynamische Prozeßkreierung und beendigung



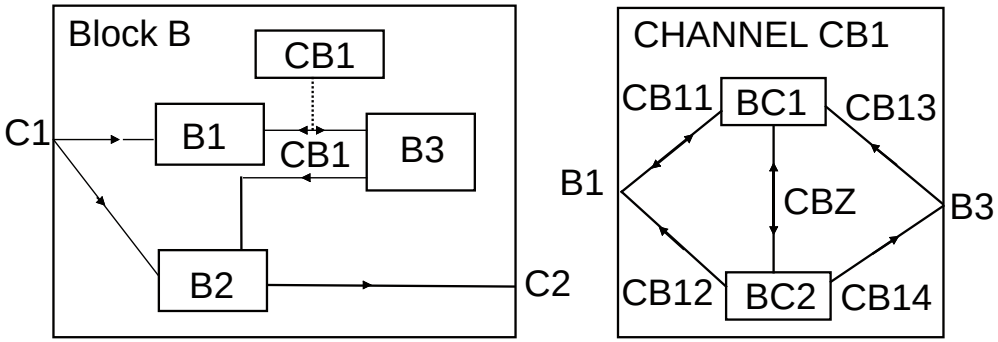


Die Blockunterstruktur

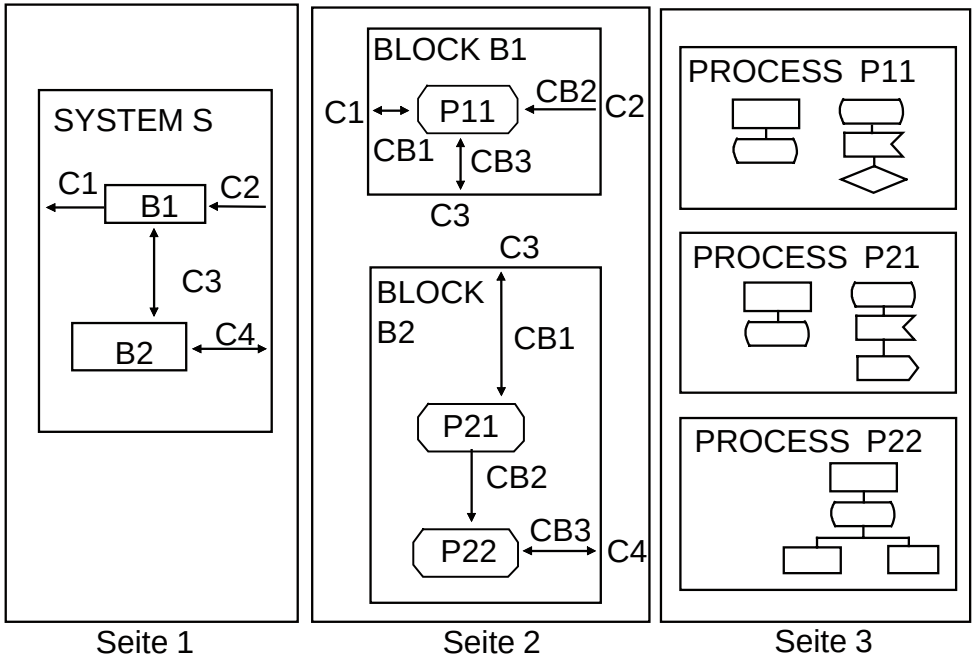


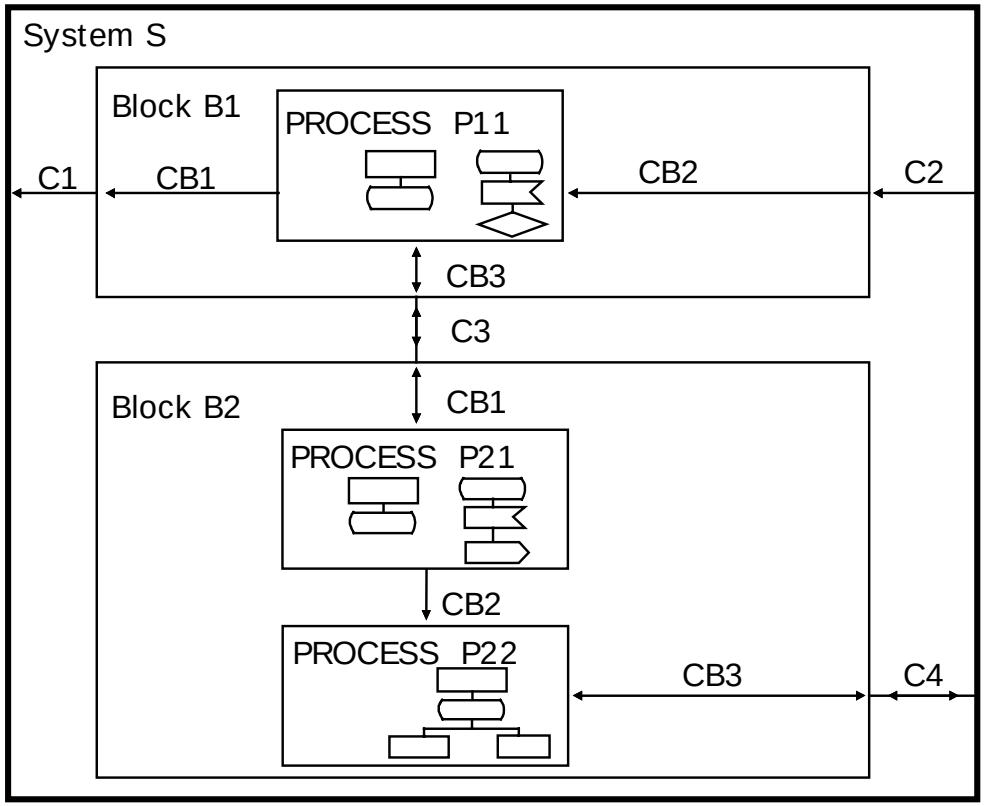


Die Kanalunterstruktur



Dokumentation
Darstellung durch Referenzierung

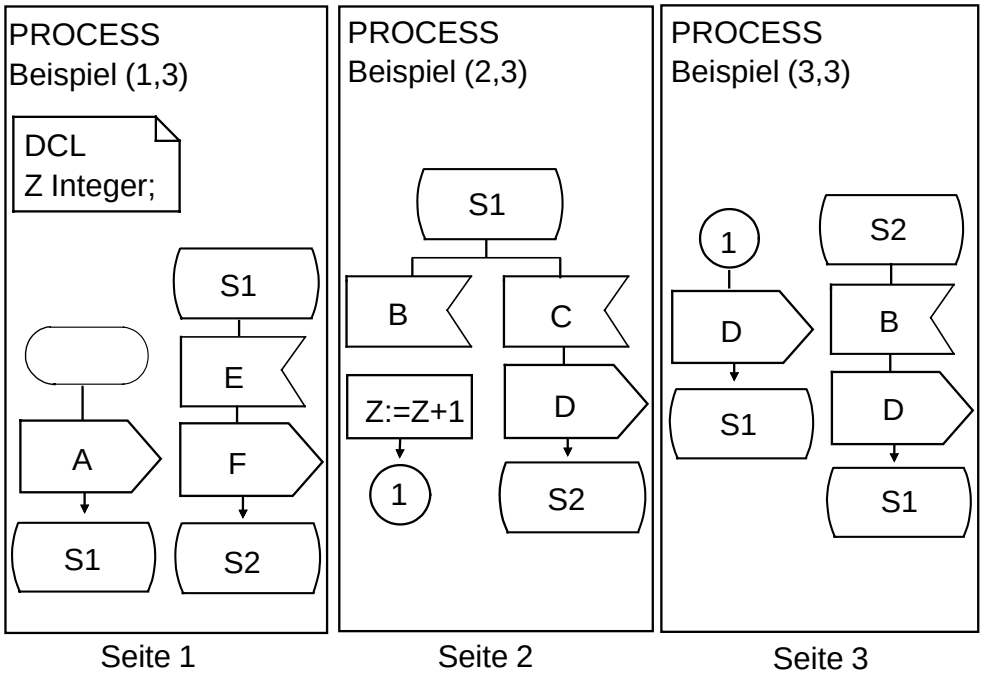




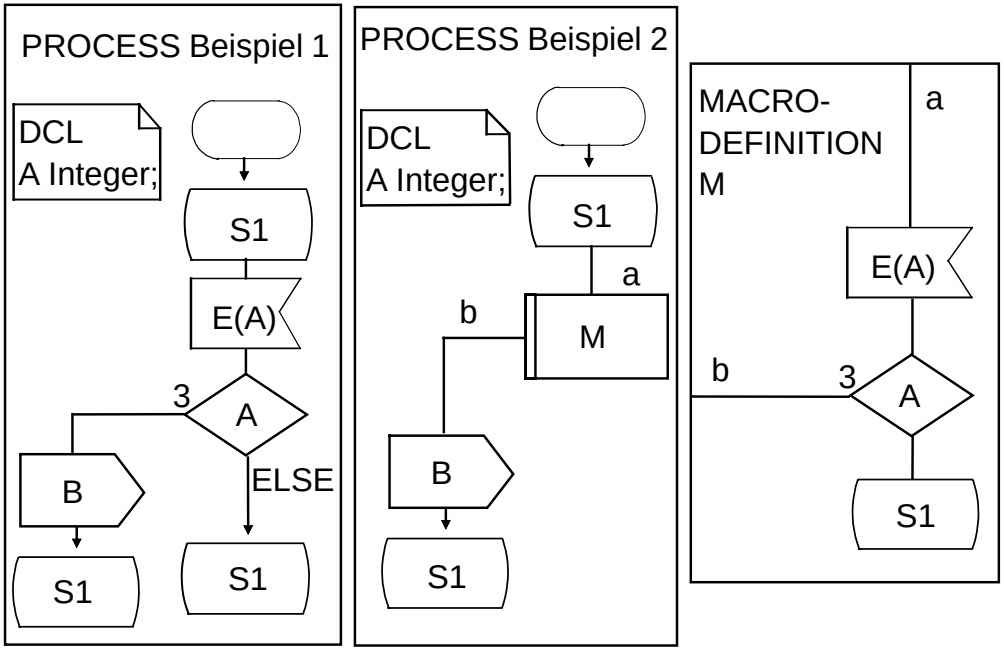
Gemischte Darstellung

Kompromiß zwischen Referenzierung und Verschachtelung

Dokumentation umfangreicher Prozeßdiagramme



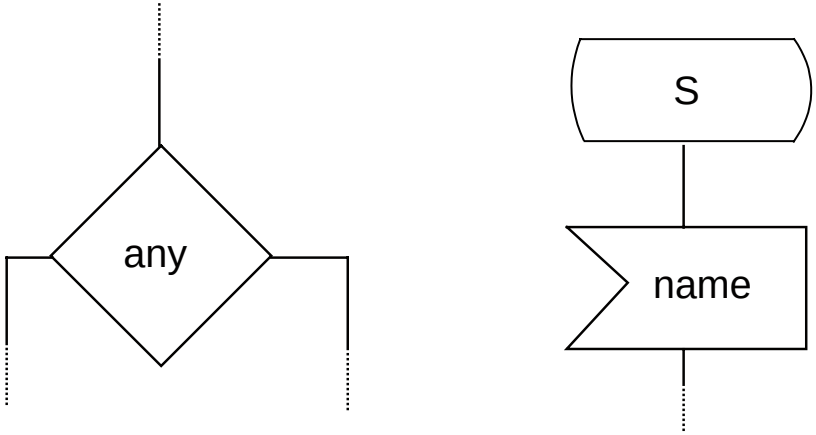
Makros



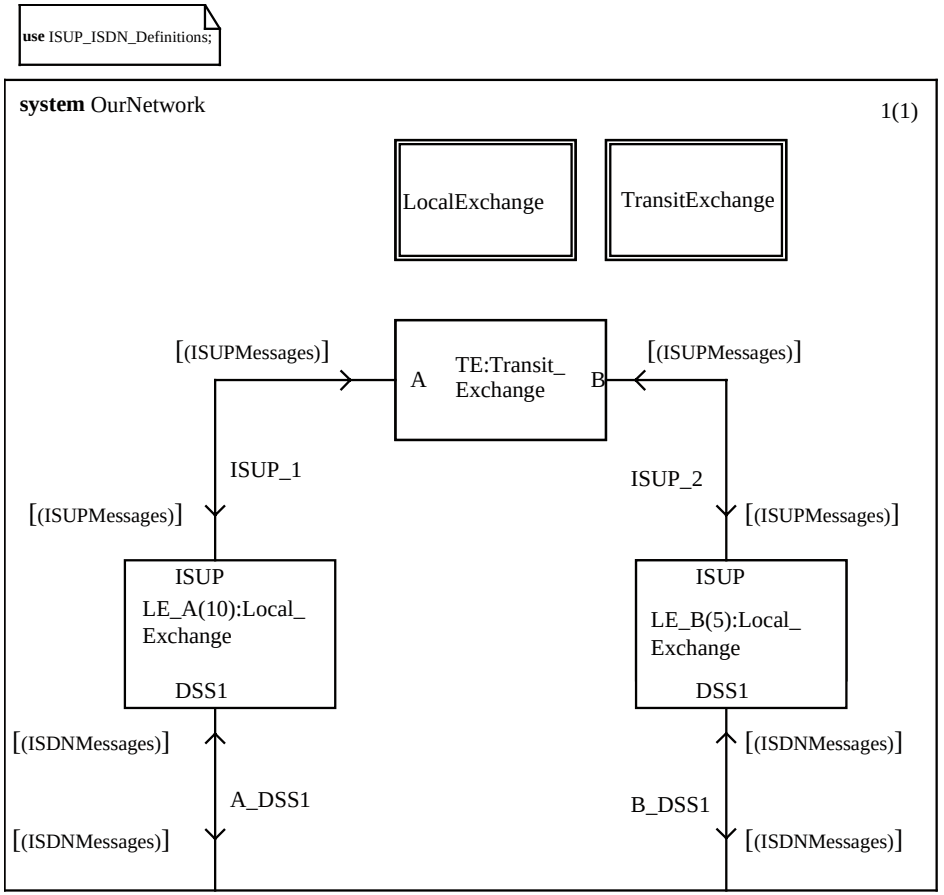
Weitere Sprachkonstrukte

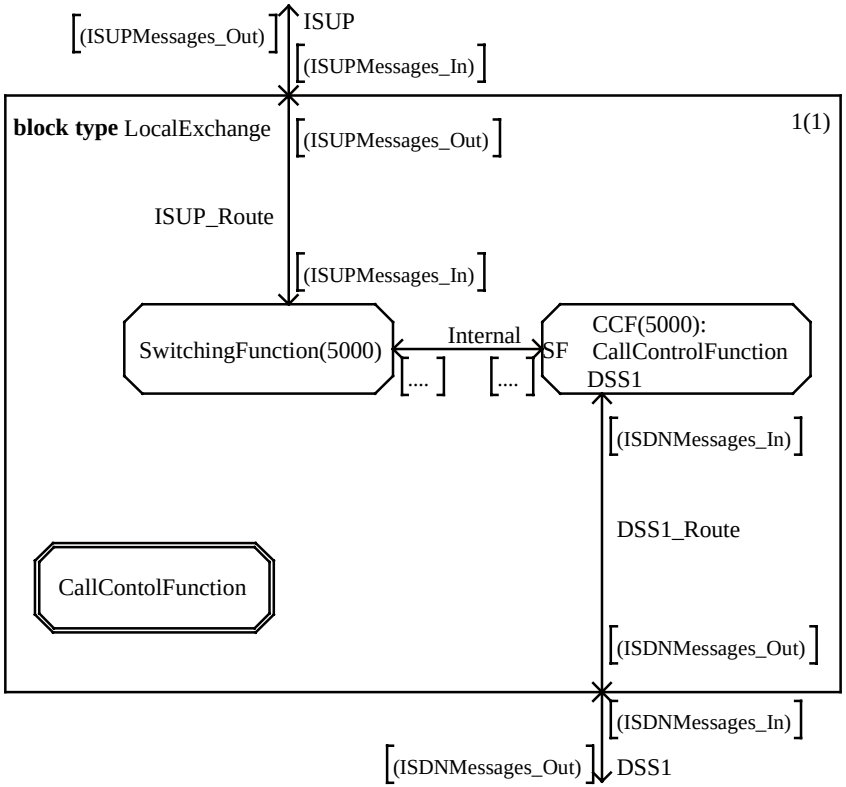
VIEW
REVIEW
IMPORT
EXPORT
enabling condition
continuous signal
SERVICE
PROCEDURE

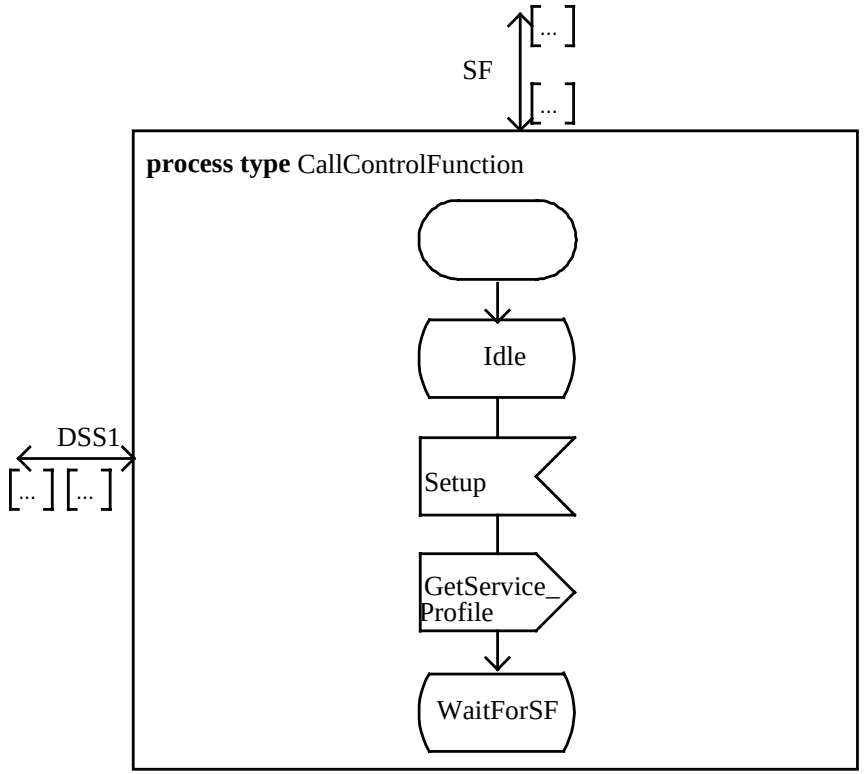
Nicht-Determinismus und SDL

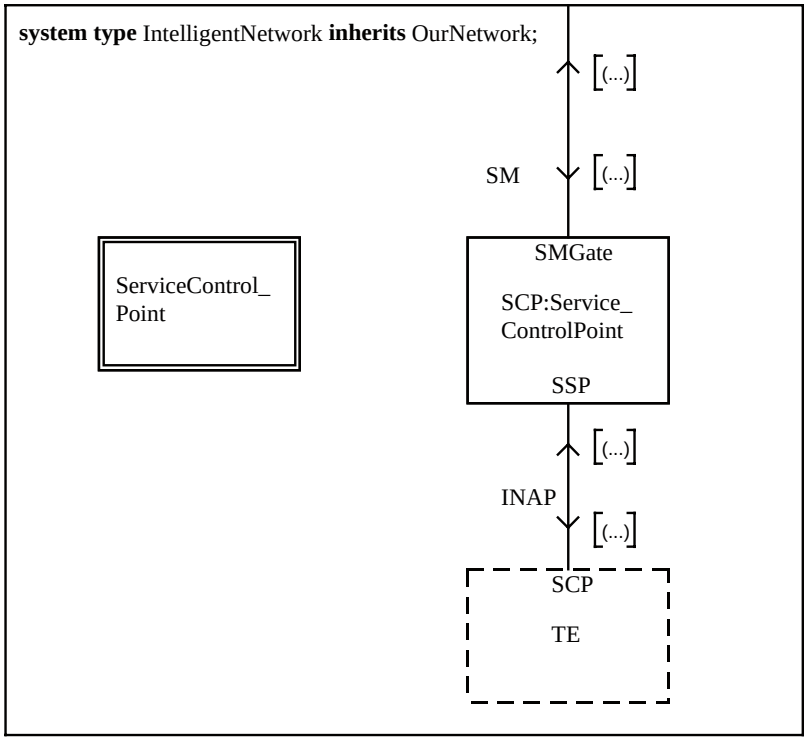


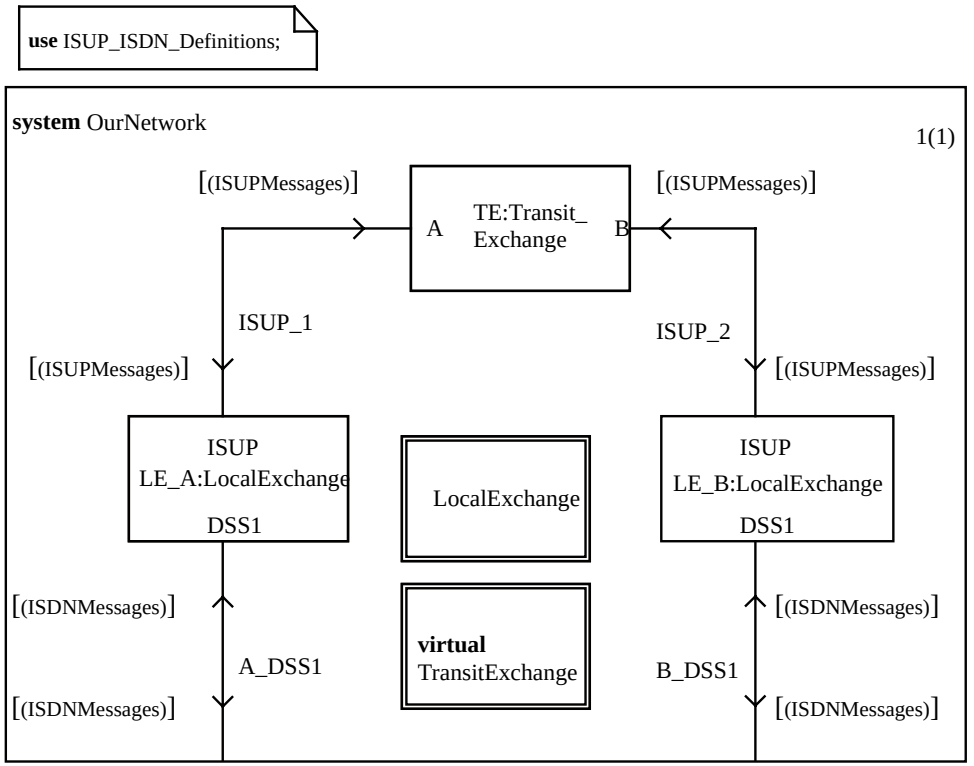
Objekt-orientierung in SDL

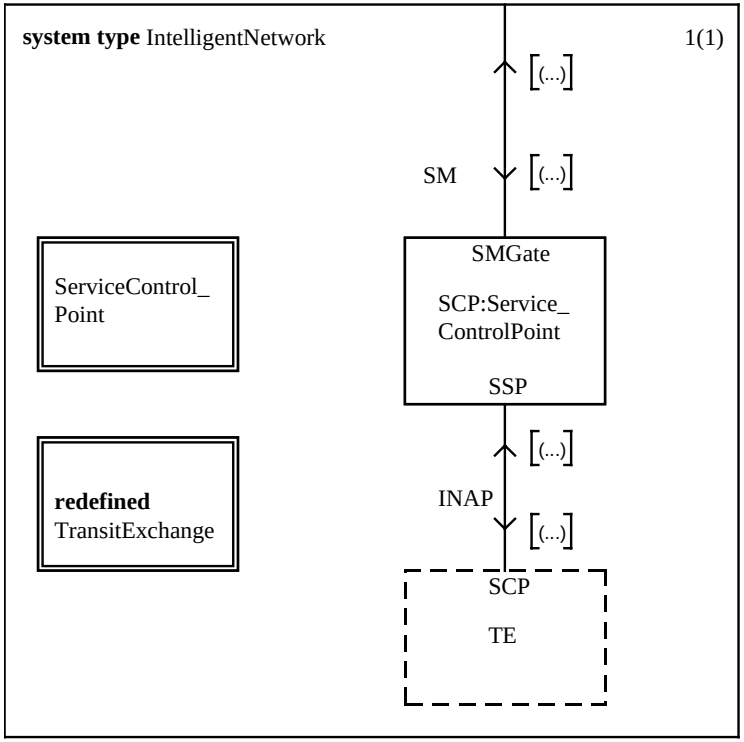


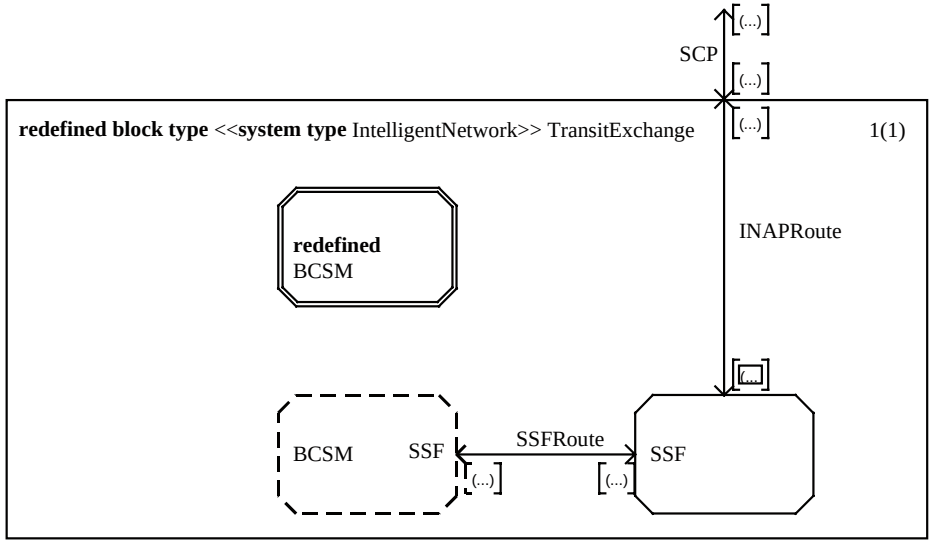

(ISUPMessages_Out)
ISUP
(ISUPMessages_In)
block type LocalExchange
1(1)
(ISUPMessages_Out)
ISUP_Route
(ISUPMessages_In)
SwitchingFunction(5000)
Internal
SF
CCF(5000):
CallControlFunction
DSS1
....
....
(ISDNMessages_In)
DSS1_Route
CallContolFunction
(ISDNMessages_Out)
(ISDNMessages_In)
(ISDNMessages_Out)
DSS1

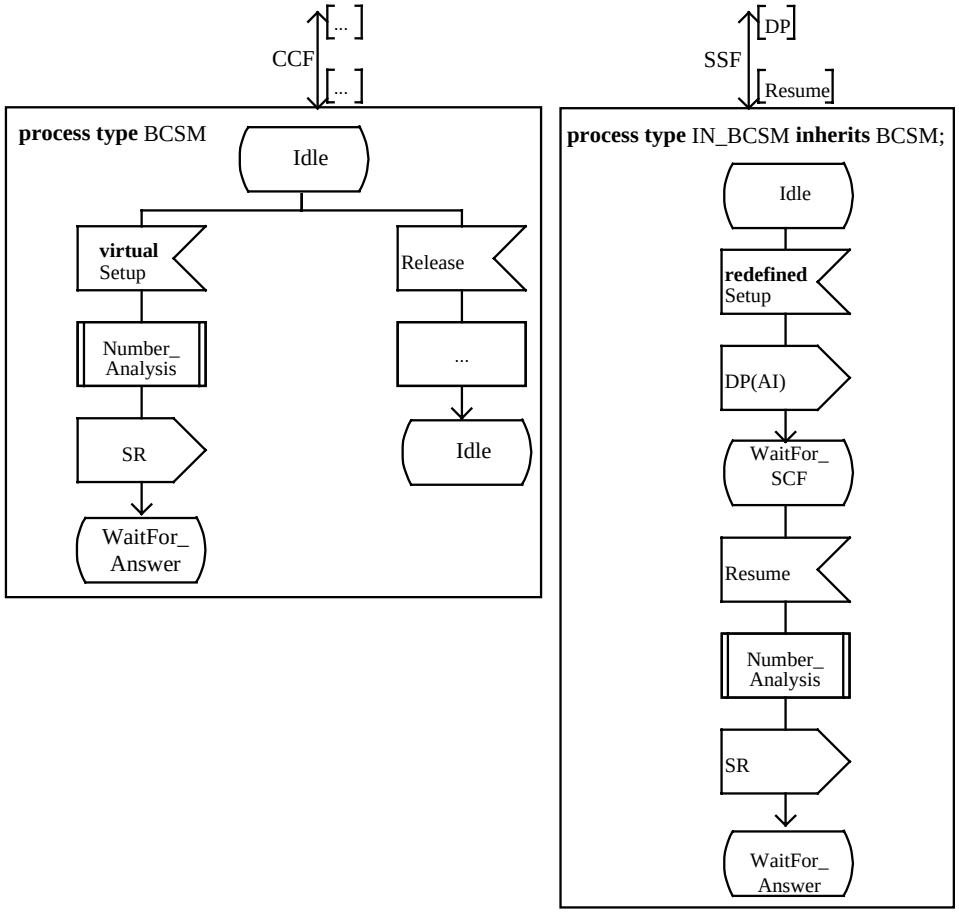

SF
...
...
process type CallControlFunction
Idle
DSS1
...
...
Setup
GetService_
Profile
WaitForSF


system type IntelligentNetwork inherits OurNetwork;
(...)
SM
(...)
SMGate
SCP:Service_
ControlPoint
SSP
ServiceControl_
Point
(...)
INAP
(...)
SCP
TE


use ISUP_ISDN_Definitions;
system OurNetwork
1(1)
(ISUPMessages)
TE:Transit_
Exchange
A
B
(ISUPMessages)
ISUP_1
ISUP_2
(ISUPMessages)
(ISUPMessages)
ISUP
LE_A:LocalExchange
DSS1
LocalExchange
ISUP
LE_B:LocalExchange
DSS1
(ISDNMessages)
(ISDNMessages)
A_DSS1
virtual
TransitExchange
B_DSS1
(ISDNMessages)
(ISDNMessages)


system type IntelligentNetwork
1(1)
(...)
SM
(...)
SMGate
SCP:Service_
ControlPoint
SSP
ServiceControl_
Point
(...)
redefined
TransitExchange
INAP
(...)
SCP
TE


SCP
(...)
(...)
(...)
redefined block type <<system type IntelligentNetwork>> TransitExchange
1(1)
redefined
BCSM
INAPRoute
(...)
BCSM
SSF
SSFRoute
SSF
(...)
(...)

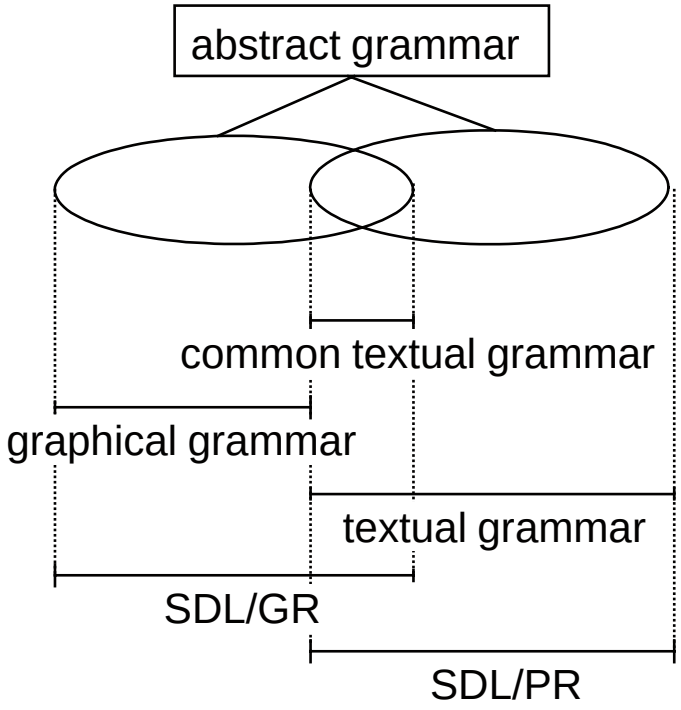


5.5 Die Sprachdefinition von SDL (CCITT Z.100)

Specification and Description Language (SDL)

- Z.100: Sprachbeschreibung (formale Grammatiken, informale Semantik, Beispiele)
- Annex A: ...
- Annex B: ...
- Annex C: ...
- Annex D: User Guidelines
- Annex E: ...
- Annex F: Formale Semantik
  - F.1: Einführung;
  - F.2: Statische Semantk;
  - F.3: Dynamische Semantik

Die SDL-Grammatiken (Z.100)



- **Abstrakte Grammatik**: abstrakte Syntax und statische Bedingungen für die Wohldefiniertheit für die abstrakte Syntax
- **Konkrete textuelle Grammatik**: konkrete textuelle Syntax und statische Bedingungen für die Wohldefiniertheit für diese Syntax und Relation zu der abstrakten Syntax
- **Konkrete graphische Grammatik**: konkrete graphische Syntax und statische Bedingungen für die Wohldefiniertheit für diese Syntax und Relation zu der abstrakten Syntax
- **Semantik**: definiert die Interpretation eines Objekts, insbesondere Bedingungen für die dynamische Wohldefiniertheit
- **Modell**: definiert eine Abbildung zwischen "shorthands" und dem "Basis-SDL"

Beispiel: Abstract grammar (2.6.2. *State*)

Abstract grammar

State-node	::	*State-name* *Save-signalset* *Input-node-set*
State-name	=	*Name*

*State-node*s within a process have different *State-names*.

For each *state-node*, all input *signal-identifiers* (in the valid input signal set) appear in either a *Save-signalset* or an *Input-node*.

The *Signal-identifier*s in the *Input-node*-set must be distinc

Metasprache zur Beschreibung der abstract Grammar:

Meta IV

Variable-definition :: *Variable-name Sort-reference-identifier*

Variable-definition ist ein zusammengesetztes Objekt bestehend aus den zwei Objekten *Variable-name* und *Sort-reference-identifier*

Sort-reference-identifier :: *Identifier*

definiert ein Objekt *Sort-reference-identifier* das syntaktisch nicht von dem Objekt *Identifier* unterschieden werden kann

Konstruktoren:

- * für eine eventuell leere Liste
- + für eine nicht-leere Liste
- | für eine Alternative
- [...] für eine Option

Metasprache zur Beschreibung der concrete textual grammar

Backus Naur Form (BNF):

<view expression> ::=
    VIEW (<identifier>,<expression>)

<block area> ::=
    <block reference>|<block diagram>
...

Beispiel: Concrete graphical grammar (2.6.2.)

Concrete graphical grammar

<state area> ::=
<state symbol> contains <state list> is associated with
{<input association area>
|<priority input association area>
|<continuous signal association area>
| <save association area>}*

<state symbol> ::= 

<input association area> ::=
<solid association symbol> is connected to <input area>

<save association area> ::=
<solid association symbol> is connected to <save area>

The <input area> is defined in §2.6.4, <save area> in §2.6.5, <continous signal association area> in §4.11, <property input association area> in §4.10.2.

A <state area> represents one or more State-nodes.

The <solid association symbol> s originating from a <state symbol> may have a common originating path.

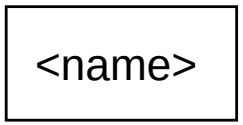
Metasprache zur Beschreibung der abstract grammar

Erweiterte BNF für graphische Syntax mit den Operatoren:

- contains
- is associated with
- is followed by
- is connected to

<block reference> ::=
<block symbol> contains <name>

<block symbol> ::=

 bedeutet: 

Beispiel: Semantics (2.6.2. *State*)

Semantics

A state represents a particular condition in which a process instance can consume a signal instance resulting in a transition.
If there are no retained signalinstances then the process waits in the state until a signal instance is received.

Beispiel: Model (2.6.2. *State*)

Model

When the <state list> of a certain <state> contains more than one <state name> s, a copy of the <state> is created for each such <state name>. Then the <state> is replaced by these copies.

5.6 Definition abstrakter Datentypen mit SDL

Allgemeine Einführung in die abstrakten Datentypen

nicht Beschreibung, **wie** ein Typ implementiert werden muß,
sondern, **was** das Resultat der Anwendung von Operatoren auf Daten eines Typs ist.

ADT = Sorten + Operatoren.

Abstrakte Datentypen gegenüber konkreten Datentypen

konkreter Datentyp = Implementierung
abstrakter Datentyp = Spezifikation

Die Signatur

Die Notation in den folgenden Beispielen ist kein SDL, lehnt sich aber an die in der Literatur der ADTen übliche Notation an, z.B. die von ACT ONE.

```
ADT Beispiel

SORTS bool,nat

OPNS true: ->bool
     false : ->bool
     not : bool->bool
     0 : ->nat
     1 : ->nat
     plus: nat,nat ->nat
     istNull : nat ->bool
```

Terme

Die Gesamtheit der Elemente einer Sorte kann durch wiederholte Anwendung von Operatoren des ADTs erzeugt werden. Eine solche Kombination von Operatoranwendungen bezeichnet man als *Term*.

Beispiele für Terme der booleschen Sorte sind

true,
false,
not(true),
not(false),
not(not(true)),...

Im obigen Beispiel haben wir zunächst die Terme

{0,1,true,false}

In dem Beispiel entsteht die folgende Menge:

{0,1,true,false,
plus(0,0),plus(1,0),plus(0,1),plus(1,1),
not(true),not(false),
istNull(0),istNull(1)}

bool-Terme:
true,false,not(true),not(false),istNull(0),.

nat-Terme: 0,1,plus(0,0),...

Gleichungen

Gleichungen zwischen Termen einer Sorte geben an, welche Terme das selbe Element der Sorte bezeichnen. Zum Beispiel die Gleichung

not(true) = false

bedeutet, daß die Terme not(true) und false das gleiche Element der sorte bool bezeichnen.

Die Menge der bool-Terme wird in Äquivalenzklassen durch die folgenden beiden Gleichungen eingeteilt

not(true) = false
not(false) = true

Die beiden zugehörigen Äquivalenzklassen sind dann

{true, not (false), not (not (true)), ... }
und
{false, not (true), not (not (false)), ... }

Desweiteren werden durch die Gleichungen die *Eigenschaften* der Operatoren definiert. Die einzigen Eigenschaften, die ein Operator also haben kann, sind, daß er bestimmte Gleichungen erfüllt. So sind zum Beispiel die Eigenschaften des *not*-Operators durch die obigen beiden Gleichungen gegeben.

```
ADT  Beispiel

SORTS  bool,nat

OPNS true :           ->bool
     false  :         ->bool
     not  :      bool ->bool
     0  :             ->nat
     1  :             ->nat
     plus : nat,nat   ->nat
     istNull  : nat   ->bool

EQNS not(true)= false
     not(false)  = true
     . . .
```


Beispiel für die Konstruktion einer Sorte mit **Hilfe von Gleichungen**

Durch *plus* werden folgende Terme erzeugt:

0 <- Terme ohne *plus*
1
plus(0,0) <- Terme mit einem *plus*
plus(0,1)
plus(1,0)
plus(1,1)
plus(plus(0,0),0) <- Terme mit zwei *plus*
plus(plus(0,0),1)
plus(plus(0,1),0)
plus(plus(0,1),1)
plus(plus(1,0),0)
plus(plus(1,0),1)
plus(plus(1,1),0)
plus(plus(1,1),1)
plus(0,plus(0,0))
plus(0,plus(0,1))

plus(0,plus(1,0))
plus(0,plus(1,1))
plus(1,plus(0,0))
plus(1,plus(0,1))
plus(1,plus(1,0))
plus(1,plus(1,1))
...
...

<- Terme mit drei *plus*

. . .

Symmetrieeigenschaft der Addition

```
FORALL x,y: nat
   plus(x,y) = plus(y,x)
```

Mit dieser Gleichung wird die Menge der unterschiedlichen Terme folgendermaßen reduziert:

0 <- Terme ohne *plus*

1

plus(0,0) <- Terme mit einem *plus*

plus(0,1)

plus(1,1)

plus(plus(0,0),0) <- Terme mit zwei *plus*

plus(plus(0,0),1)

plus(plus(0,1),0)

plus(plus(0,1),1)

plus(plus(1,1),0)

plus(plus(1,1),1)

. . . <- Terme mit drei *plus*

. . . . . .

Addition von Null

FORALL x: nat
    plus(x,0) = x

Damit reduzieren sich Termmengen wie folgt:

0	<- Terme
ohne *plus*	
1	
plus(1,1)	<- Terme mit
einem *plus*	
plus(plus(1,1),1)	<- Terme mit
2 *plus*	
plus(plus(plus(1,1),1),1)	<- Terme mit
3 *plus*	
plus(plus(1,1),plus(1,1))	
. . .	<- Terme mit
4 *plus*	
. . .	. . .

Es wird nun langsam eine mögliche Zuordnung der Terme zu den natürlichen Zahlen deutlich:

0 = *null*
1 = *eins*
plus(1,1) = *zwei*
plus(plus(1,1),1) = *drei*
plus(plus(plus(1,1),1),1) = *vier*
plus(plus(plus(plus(1,1),1),1),1) = *fünf*
... ...

Die "Additionseigenschaft" der Addition

FORALL x,y: nat
plus(x,plus(y,1)) = plus(plus(x,y),1)

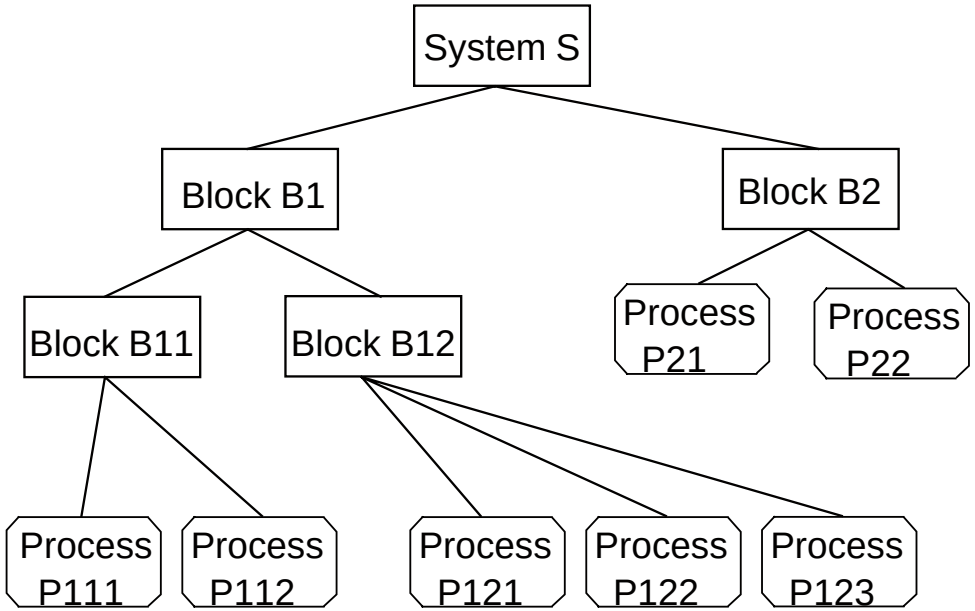
Jetzt gilt: "*zwei + drei = fünf*"

plus(plus(1,1),plus(plus(1,1),1)) =
plus(plus(plus(plus(1,1),1),1),1)

Eine vollständige Definition des Datentyps Beispiel inklusive der Gleichungen für den Operator istNull könnte dann wie folgt aussehen:

```
ADT Beispiel
SORTS    bool,nat
OPNS     true  :       ->bool
         false :       ->bool
         not   :   bool->bool
         0     :       ->nat
         1     :       ->nat
         plus  : nat,nat->nat
         istNull   :  nat->bool
EQNS     not(true)  =  false
         not(false) =  true
         FORALL x,y: nat
           plus(x,y) = plus(y,x)
           plus(x,0) = x
           plus(x,plus(y,1)) =
               plus(plus(x,y),1)
           istNull(0) = true
           istNull(plus(x,1)) =
               false
```

Hierarchische Definition abstrakter Datentypen



Es seien definiert in

S die Sorten: *Sor1*,*Sor2*,
die Operatoren: *Op1*,*Op2*,*Op3* und
die Gleichungen: *Gl1*,*Gl2*

B1 die Sorte: *SorA*,*SorB*,*SorC*,
keine Operatoren und
die Gleichung: *GlA*

B11 keine Sorten,
die Operatoren: *OpA*,*OpB* und
die Gleichung: *GlB*

dann ist dadurch implizit der ADT mit
den Sorten: Sor1,Sor2,SorA,SorB,SorC,
den Operatoren:
Op1,*Op2*,*Op3*,*OpA*,*OpB* und
den Gleichungen: *Gl1*,*Gl2*,*GlA*,*GlB*
definiert

Bei der hierarchischen Spezifikation des ADT gibt es gewisse Konsistenzregeln zu beachten, damit der ADT insgesamt wohldefiniert ist. Insbesondere ist es nicht erlaubt,

- mit einer Gleichung Terme äquivalent zu machen, die auf einer höheren Hierarchiestufe nicht äquivalent sind,

und

- neue Werte zu einer Sorte hinzuzufügen, die auf einer höheren Hierarchiestufe definiert ist.

Spezifikation von Signatur und Gleichungen mit partiellen Datentypdefinitionen

```
NEWTYPE sortenname
...
ENDNEWTYPE;
```


Beispiel

```
NEWTYPE bool
     LITERALS true, false
     OPERATORS not: bool -> bool
     AXIOMS
          not(true)  ==  false;
          not(false)  ==  true
ENDNEWTYPE bool;
```

```
NEWTYPE nat
     LITERALS 0,1
     OPERATORS
       plus:  nat,nat  ->  nat;
       istNull:  nat  ->  bool
     AXIOMS
     FOR ALL x,y IN nat
     (plus(x,y)  ==  plus(y,x);
     plus(x,0)  ==  x;
     plus(x,plus(y,1)) ==
         plus(plus(x,y),1);
     istNull(0)  ==  true;
     istNull(plus(x,1))  ==
         false)
ENDNEWTYPE nat;
```


"Axiome"

Term == true

Umbenennung von Sorten

```
SYNTYPE ganze_Zahlen = Integer;
SYNTYPE
Fenster = Integer CONSTANTS 0:4;
```

Konstanten

```
SYNONYM max_Laenge Integer = 4096;
```

Bedingte Gleichungen
Bedingung ==> Gleichung

```
FOR ALL  x,z  IN Real
   (z/=0 == true ==> (x/z)*z == x)
FOR ALL x,z IN Real
   (z=0 == true ==> (x/z)*z == ...)
FOR ALL x,z IN Real
   ((x/z)*z == IF z/=0 THEN x ELSE ...
FI)
```

Fehler

```
FOR ALL x IN Real
   ((x/0) == Error!)
```

Error! sollte stets benutzt werden, um auszudrücken:
"Anwendung des Operators auf diese Werte ist nicht erlaubt, und wenn es trotzdem gemacht wird, ist das zukünftige Verhalten der Spezifikation undefiniert."
Vererbung

Durch Vererbung können
- alle Werte,
- einige oder alle Operatoren
- alle Gleichungen

einer Vater-Sorte von einer Sohn-Sorte übernommen werden.

Mit dem Schlüsselwort ADDING können dann weitere Literale, Operatoren und Gleichungen zu der Sohn-Sorte hinzugefügt werden.

Die vererbten Operatoren können umbenannt werden, um Mißverständnisse zu vermeiden.

```
NEWTYPE Sor1
   LITERALS Li1,Li2
   OPERATORS Op1: ...
   OPERATORS Op2: ...
   OPERATORS Op3: ...
   AXIOMS ...
ENDNEWTYPE;
NEWTYPE Sor2
   INHERITS Sor1
   OPERATORS (Op1,OpA=Op2)
   ADDING
   LITERALS LiA
   OPERATORS OpB: ...
   AXIOMS ...
ENDNEWTYPE Sor2;
```

Der Fall, daß alle Operatoren übernommen werden sollen, kann mit

```
. . .
INHERITS. . .
OPERATORS ALL
. . .
```

spezifiziert werden.
Generatordefinition

die "Variationen eines Themas"

```
NEWTYPE Int_Menge
LITERALS leere_Int_Menge

OPERATORS add: Int_Menge,Integer ->
                 Int_Menge
   ist_in: Int_Menge,Integer ->
                 Boolean

AXIOMS

   ist_in(leere_Int_Menge,x) == false;
   ist_in(add(m,x)y) == (x=y) or
                 (ist_in(m,y));
   x=y ==> add(add(m,x),y) ==
                 add(m,x);
   x/=y ==> add(add(m,x),y) ==
                 add(add(m,y),x);

ENDNEWTYPE;
```

```
NEWTYPE Real_Menge

LITERALS leere_Real_Menge

OPERATORS add: Real_Menge,Real ->
Real_Menge
   ist_in: Real_Menge,Real -> Boolean

AXIOMS

   ist_in(leere_Real_Menge,x) ==
                    false;
   ist_in(add(m,x)y) == (x=y) or
                    (ist_in(m,y));
   x=y ==> add(add(m,x),y) ==
                    add(m,x);
   x/=y ==> add(add(m,x),y) ==
                    add(add(m,y),x);

ENDNEWTYPE;
```



```
GENERATOR Menge
   (TYPE Element, LITERAL leere_Menge)

LITERALS leere_Menge

OPERATORS add: Menge,Element -> Menge
   ist_in: Menge,Element -> Boolean

AXIOMS
   ist_in(leere_Menge,x) == false;
   ist_in(add(m,x)y) == (x=y) or
                        (ist_in(m,y));
   x=y ==> add(add(m,x),y) ==
                        add(m,x);
   x/=y ==> add(add(m,x),y) ==
                        add(add(m,y),x);

ENDGENERATOR;
```

```
NEWTYPE Int_Menge
Menge(Integer,leere_Int_Menge)
ENDNEWTYPE;

NEWTYPE Real_Menge
Menge(Real,leere_Real_Menge)
ENDNEWTYPE;

GENERATOR Menge
(TYPE Element,LITERAL leere_Menge,
OPERATOR Groesse, CONSTANT
Max_Groesse)

LITERALS leere_Menge

OPERATORS Groesse: Menge -> Integer;
   . . .
```


Verbund- und Feld-Typen

```
NEWTYPE AdressenTyp
   STRUCT
   Name,
   Vorname,
   Strasse: Charstring;
   Nummer: Integer;
   Stadt: Charstring;
ENDNEWTYPE;

Adresse:=(.Hogrefe,D.,Am
       Waldrand.,29,Gr. Grönau.);
Ort := Adresse!Stadt;

SYNTYPE Zaehler = Integer
CONSTANTS 1:10;
NEWTYPE KarteiTyp
       Array(Zaehler,AdressenTyp)
ENDNEWTYPE;

Kartei(8) := Adresse;
```

Vordefinierte Sorten

Boolean
Character
Charstring
Integer
Natural
Real
PId
Duration
Time
String
Array
Powerset