

# ***Modul OMSI-2 im SoSe 2010***

## ***Objektorientierte Simulation mit ODEMx***

Prof. Dr. Joachim Fischer  
Dr. Klaus Ahrens  
Dipl.-Inf. Ingmar Eveslage  
Dipl.-Inf. Andreas Blunk

[fischer|ahrens|eveslage|blunk@informatik.hu-berlin.de](mailto:fischer|ahrens|eveslage|blunk@informatik.hu-berlin.de)

## 6. *SDL-Einführung*

1. Grundphilosophie
2. ITU-Standard Z.100
3. Werkzeuge
4. SDL-Grundkonzepte
5. Deamon-Game: Musterbeispiel (in UML-Strukturen)
6. DeamonGame:  
Struktur- und Verhaltensbeschreibung in SDL-RT
7. DeamonGame: Simulation
8. PragmaDev-Tutorial

# Konfiguration der SD-Ausgabe (MSC)

The screenshot displays the RTDS (Real Time Developer Studio) interface. The main window is titled "RTDS - Diagram 'Testlauf' (modified)". A yellow arrow points to the "SDL-RT debugger" window, which is currently in a "STOPPED" state. The debugger window contains several panes: "Process information" (a table with columns for Name, Prio, SDL id, RTOS id, Msg, SDL-RT state, and System state), "Watch variables", "Local variables", and "Timer info". A yellow callout box points to the "Process information" table with the text: "Standardmäßig werden alle System-Process-Instanzen und alle Nachrichten erfasst".

Below the debugger window is a "Real Time Developer Studio shell" with the following text:

```
>Free run off.  
>
```

The "MSC configuration" window is also visible, showing a tree of "Available agents" including DeamonGameSystem, DeamonGame, Monitor, GameServer, Deamon, PlayerEnsemble, Creator, Player, and RTDS\_Env. The "Traced agents" list includes GameServer and Monitor. The "Record message data" checkbox is checked.

The background shows a sequence diagram with participants like "Player (0:2e4420)", "For GameServer", and "playing".

# Einrichtung eines MSC-Trace-Diagramms

The screenshot displays the SDL Tracer application. The main window shows an MSC diagram with four lifelines: mySem (0x4b3ef00), playerA (0x4b3e7c8), RTDS\_Env (0x4b25898), and playerB (0x4b22268). The diagram shows a sequence of events starting at system time 1779. A semaphore is created, followed by playerA entering a 'playing' state, RTDS\_Env entering an 'RTDS\_Idle' state, and playerB entering a 'playing' state. The time progresses to 1822, 1823, and 1824.

A yellow arrow points to the 'SDL Tracer' window title bar. A callout box with a white background and black border contains the text: "All SDL events are dynamically displayed graphically."

The right-hand side of the interface shows the 'Debugger' window. It includes a 'Debugger Options' toolbar, a 'Process information' table, and a 'Real Time Developer Studio shell' terminal window. The terminal window displays the following output:

```
Real Time Developer Studio shell
>Executable loaded.
>Free run off.
>run
>Semaphore: mySem(0x4b3ef00) created by: Unknown at: 0x6f3 ticks
>Task playerA(0x4b3e7c8) prio:150 created by Unknown() at: 0x6fc ticks
>Task RTDS_Env(0x4b25898) prio:150 created by Unknown() at: 0x705 ticks
>Task playerB(0x4b22268) prio:200 created by Unknown() at: 0x70f ticks
>RTDS initialization done.
Flushing startup semaphore
>Task playerA(0x4b3e7c8) has changed to state playing at: 0x71e ticks
>Task RTDS_Env(0x4b25898) has changed to state RTDS_Idle at: 0x71f ticks
>Task playerB(0x4b22268) has changed to state playing at: 0x720 ticks
>
```

At the bottom of the debugger window, it shows 'Debugger state: RUNNING' and 'Active thread: 0x4b22268->playerB'.

# Eingabe von Umgebungsnachrichten

The image shows two windows from a software development tool. The left window, 'MSC Tracer', displays a sequence diagram with lifelines for 'mySem', 'playerA', 'RTDS\_Env', and 'playerB'. The right window, 'SDL-RT debugger', shows a 'Process information' table and a console output area.

**Handänderungen für aktiven Prozess möglich**

**Eingabe von Umgebungssignalen (Empfänger, Signaltyp, Parameter)**

The list of active task is updated as well as the list of semaphores and pending timers.

Name	Prio	Pid	Msg	SDL-RT state	System state
playerA	150	0x4b3e7c8		playing	pend
RTDS_Env	150	0x4b25898		RTDS_idle	pend
playerB	200	0x4b22268		playing	pend

```
>stop
>Debugger received a signal ! Dumping last reply:
>[Switching to task 0x4b39170 + tExcTask ] Program received signal SIGINT,
>refresh
>
```

# Start des Programms

The screenshot shows the MSC Tracer interface with a sequence diagram and the SDL-RT debugger. A dialog box titled "Send an SDL-RT message" is open, allowing the user to select a sending process, choose from available messages (mPing, mStart, mStop), and select a receiving process. A yellow arrow points to the debugger window, and a callout box highlights the "Available messages" table.

**Available messages:**

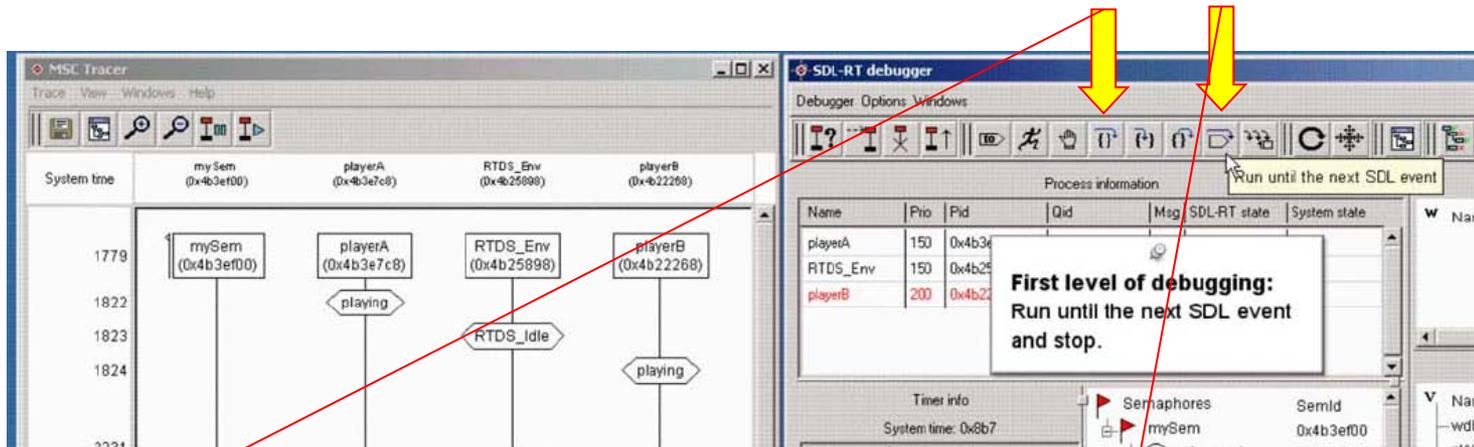
Name	#	Type
mPing	1	N/A
mStart	3	N/A
mStop	4	N/A

**Process information:**

Name	Prio	Pid	Qid	Mag	SDL-RT state	System state
playerA	150	0x4b3e7c8	0x4b3eb30	0	playing	pend
RTDS_Env	150	0x4b25898	0x4b3b118	0	RTDS_Idle	pend
playerB	200	0x4b22268	0x4b3ad60	0	playing	pend

**Debugger state:** STOPPED  
Active thread: 0x4b22268->playerB

# Debugger-Steuerung



1) übliche C-Debugger-Funktionalität (selbst erstellt oder generiert)

- flat step in the code
- step in
- step out

2) SDL-Debugger-Funktionalität

- until the next SDL event
- current Input queue empty
- until graphical symbol

Fragen:

- Was ist ein SDL-event ? Beispiele?
- Wodurch ist ein Zustand eines SDL-Systems charakterisiert?

# Wechsel: SDL/GR-Sicht → C/C++ -Textsicht

Debuggen im externen oder  
Generiertem C/C++ Code

The screenshot displays the SDL-RT debugger interface. The top-left window shows a sequence diagram for 'tennisMan' with participants 'mPing' and 'mPong'. The bottom-left window shows the source code for 'localCounter.cpp' with line 29 highlighted. The right window shows the 'SDL-RT debugger' with a 'Process information' table and a 'Semaphores' list.

Name	Prio	Pid	Qid	Mig	SDL-RT state	System state
playerA	150	0x4b3e7c8	0x4b3eb30	0	playing	ready
RTDS_Errv	150	0x4b25898	0x4b3b118	0	RTDS_idle	pend
playerB	200	0x4b22268	0x4b3ad50	0	playing	ready

Debugger state: STOPPED | Active thread: 0x4b3e7c8->playerA

It is possible to step  
in the external C or  
C++ code.

# Rückkehr: Text → Grafik

The image displays the RTDS (Real-Time Design Studio) interface. On the left, a graphical state transition diagram for a 'pingpong' system is shown. It includes states like 'playing' (red hexagon), 'mPing TO\_ID SENDER' (arrow), and 'mySem(FOREVER)'. On the right, the generated C code is visible, with line 155 highlighted: `RTDS_SEMAPHORE_NAME_GIVE("mySem", mySem);`. A yellow arrow points from this line to a callout box containing the text: "From anywhere in the generated C code, it is possible to go the graphical source code." The bottom of the screenshot shows the debugger's file stack and status, indicating the current file is `server.c` at line 155, column 0.

# Grafik → Generierter Code

The image displays the RTDS (Real-Time Design Studio) interface, which is used for modeling and generating code for real-time systems. The main window is titled "RTDS - Diagram 'tennisMan' (modified)".

**Diagram View (Left):** Shows a state machine diagram for a tennis game. The states are `playing`, `mPing`, and `mPong`. Transitions include `mPong` and `mPing`. The `playing` state contains logic for `globalCounter++`, `myCount->increment()`, and `mySem`. A callout box states: "It is also possible to go from the graphical source code to the generated C code." Below the diagram, there are sections for "Generated Source" and "Generated Code".

**Code View (Right):** Shows the generated C code for the diagram. The code is a switch statement that handles different signals and states. The `playing` state is highlighted in yellow. The code includes comments and function calls like `RTDS_SEMAPHORE_NAME_GIVE` and `RTDS_MSG_QUEUE_SEND_TO_ID`.

```
case mStop:
    RTDS_SDL_STATE_SET(sIdle);
    break;

/* State playing - signal mPing */
case mPing:
    RTDS_SEMAPHORE_NAME_TAKE("mySem", mySem, RTDS_SEMAPHORE_TIME_OUT_FO
    globalCounter++;
    myCount->increment();
    myRecord.score=myCount->getCount();
    RTDS_MSG_QUEUE_SEND_TO_ID(mPong, 0, NULL, SENDER);
    RTDS_SEMAPHORE_NAME_GIVE("mySem", mySem);
    RTDS_SDL_STATE_SET(playing);
    break;

/* State playing - signal mPong */
case mPong:
    RTDS_SEMAPHORE_NAME_TAKE("mySem", mySem, RTDS_SEMAPHORE_TIME_OUT_FO
    globalCounter++;
    myCount->increment();
    RTDS_SEMAPHORE_NAME_GIVE("mySem", mySem);
    RTDS_MSG_QUEUE_SEND_TO_ID(mPing, 0, NULL, SENDER);
    RTDS_SDL_STATE_SET(playing);
    break;

/* State playing - signal mStart */
case mStart:
    RTDS_MSG_QUEUE_SEND_TO_NAME(mPing, 0, NULL, "playerB", RTDS_process
    RTDS_SDL_STATE_SET(playing);
    break;

} /* End of switch(RTDS_currentContext->currentMessage->messageNumber)
break;
case sIdle:
    switch(RTDS_currentContext->currentMessage->messageNumber)
    {

/* State sIdle - signal mStart */
case mStart:
    RTDS_MSG_QUEUE_SEND_TO_NAME(mPing, 0, NULL, "playerB", RTDS_process
    RTDS_SDL_STATE_SET(sPlaying);
```

# Anzeige von Variablenänderungen

The screenshot displays the RTDS-Debugger interface. The main window shows a UML diagram with a code block containing the following code:

```
globalCounter++;  
myCount->increment();
```

The code block is highlighted in yellow, indicating a variable change. The debugger window below shows the following data:

**Process information**

Name	Prio	Pid	Qid	Msg	SDL-RT state	System state
playerA	150	0x4b3e7c8	0x4b3eb30	0	playing	ready
RTDS_Env	150	0x4b25898	0x4b3b118	0	RTDS_idle	pend
playerB	200	0x4b22268	0x4b3ad60	0	playing	ready

**Local variables**

Names	Values
myCount (localCounter*)	0x4b2ccb8
->count	0
myRecord	-
-_firstName	0x495cb20 "Jehn"
-_lastName	0x495cb25 "Smith"
-_position	National
-_score	0

A callout box points to the local variables window with the text: "Local variables are displayed automatically."

**Tracing**

```
>key5dl3step  
>Message: mPong uniqueId: 1 received by: playerA(0x4b3e7c8) at: 0x8c2 ticks  
>key5dl3step  
\$SendHeader: mySem(0x4b3e7c8) SDL-Entnahme by: playerA at: 0x8c2 ticks
```

# Setzen von Beobachtungspunkten

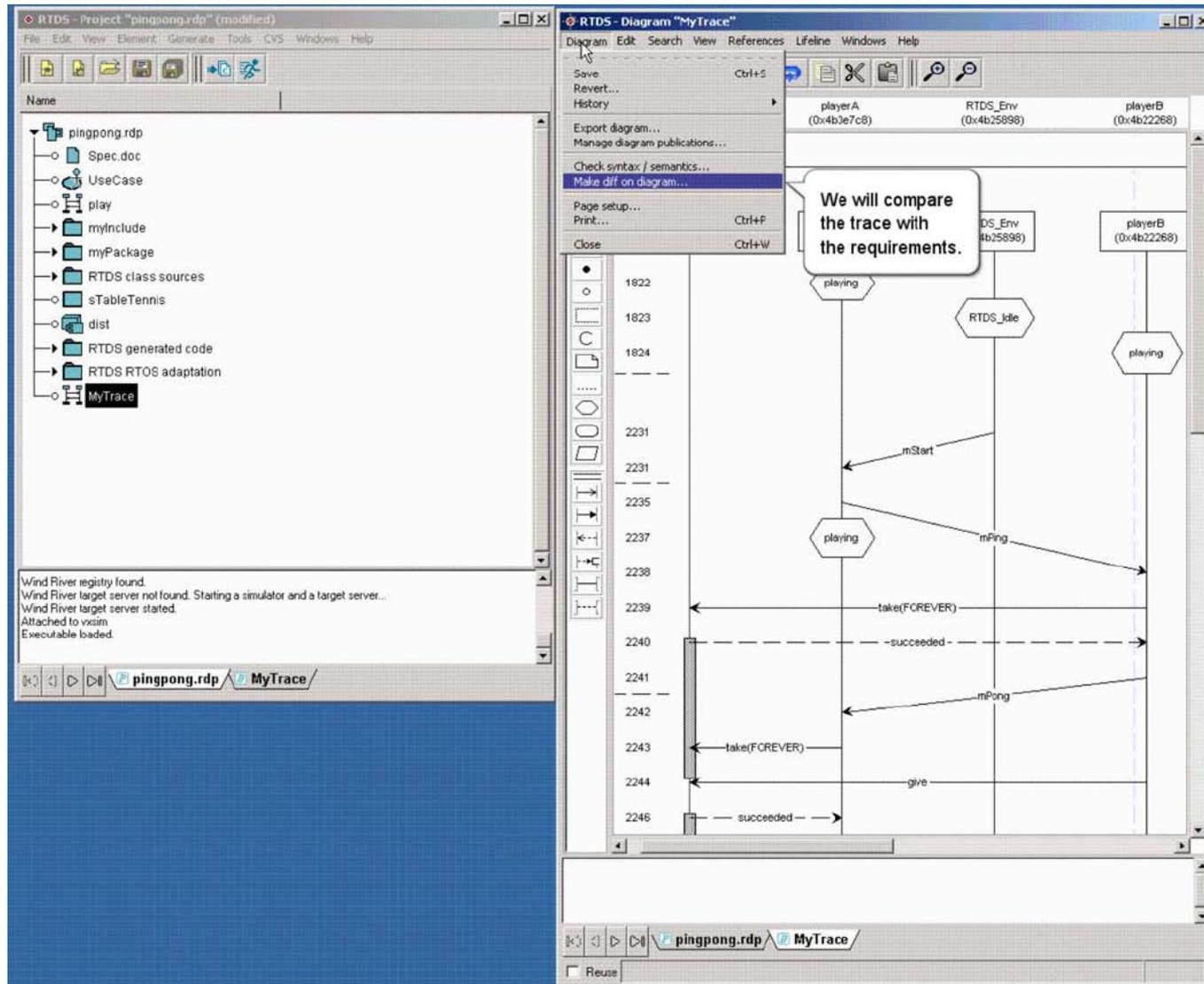
The screenshot shows the RTDS Diagram 'tennisMan' interface. The main diagram area displays a sequence diagram with several lifelines: 'EVER', 'SENDER', 'mPing', 'mySem(FOREVER)', 'mySem', and 'playing'. A red box highlights a state transition in the 'mySem(FOREVER)' lifeline, with a yellow callout box containing the text 'Markieren und aufnehmen'. The watch window and local variables panel are visible on the right, showing the current state of the system and the values of local variables. The local variables panel shows a tree structure for 'Names' with fields: 'myCount' (value: (localCounter \*) 0x4b2ccb8), 'count' (value: 0), 'myRecord' (value: -), 'firstName' (value: 0x495cb20 "John"), 'lastName' (value: 0x495cb25 "Smith"), 'position' (value: National), and 'score' (value: 0). The tracing window at the bottom shows the following log entries:

```
>keySdlStep  
>Message: mPing uniqueId: 1 received by: playerA(0x4b3e7c8) at: 0x8c2 ticks  
>keySdlStep  
>Semaphore: mySem(0x4b3ef00) take_attempt by: playerA at: 0x8c3 ticks
```

# Speichern eines MSC-Diagramms

The image displays two software windows. The left window, 'MSC Tracer', shows a sequence diagram with four lifelines: mySem (0x4b3ef00), playerA (0x4b3e7c8), RTDS\_Env (0x4b25898), and playerB (0x4b22268). The diagram includes messages like mStart(), mPing, take(FOREVER), and give, along with activation bars and 'playing' states. A 'Paused' indicator is at the bottom. A dialog box 'Trace not saved. Do you want to save it now?' is overlaid on the diagram. The right window, 'RTDS - Project "pingpong.rdp" (modified)', shows a file explorer with a tree view containing folders like 'pingpong.rdp', 'Spec.doc', 'UseCase', 'play', 'myInclude', 'myPackage', 'RTDS class sources', 'sTableTennis', 'dist', 'RTDS generated code', and 'RTDS RTOS adaptation'. A callout box points to the 'MyTrace' file with the text: 'The MSC Trace is now in the project file. Let's open it.' The bottom status bar of the RTDS window shows 'pingpong.rdp'.

# Vergleich: Anforderung - Simulationsergebnis

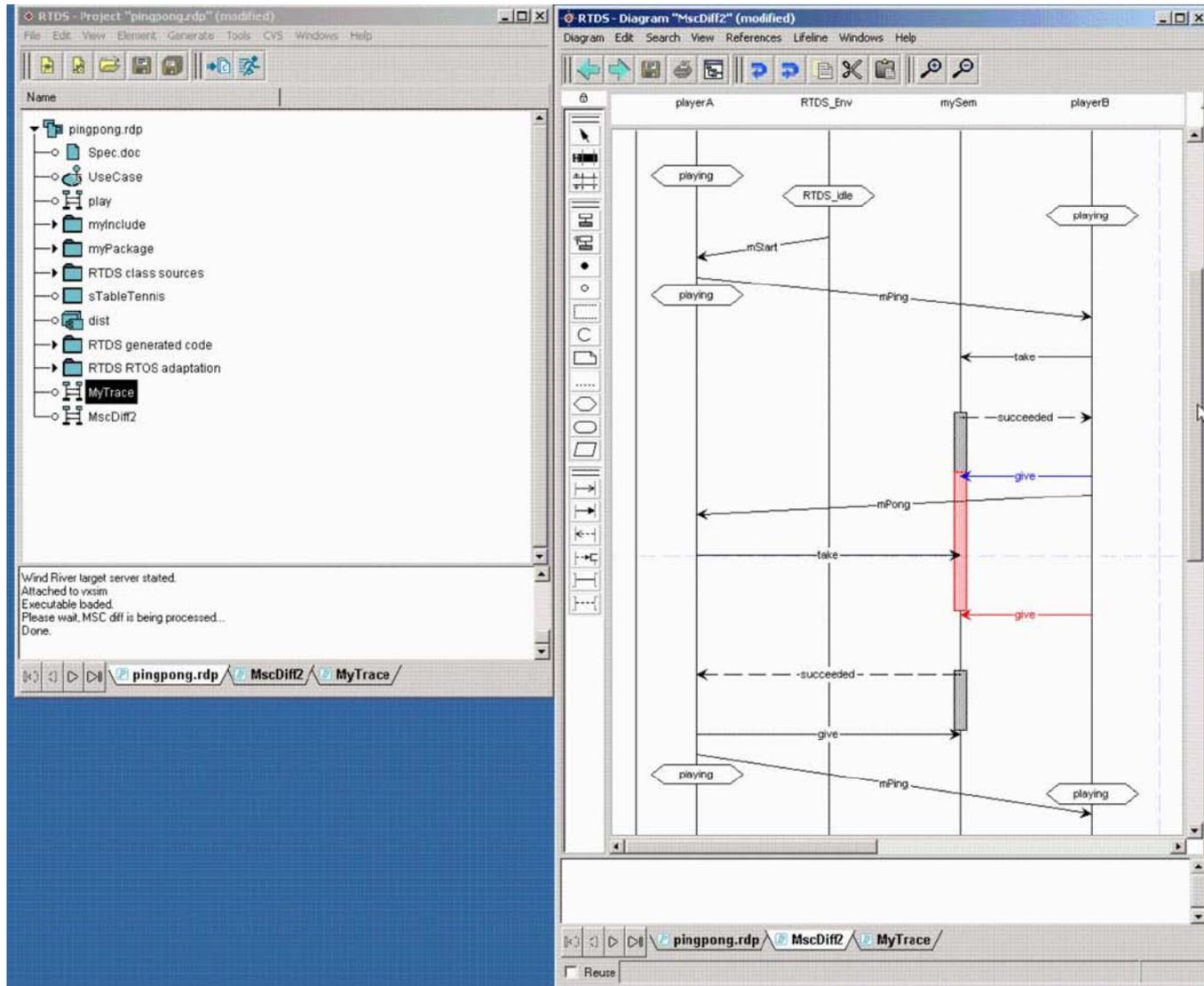


# Benutzung von MSC-Filtern

The screenshot displays the RTDS software interface. On the left, a project tree shows the structure of the 'pingpong.rdp' project. The main window shows an MSC diagram with participants: mySem (0x4b3ef00), playerA (0x4b3e7c8), RTDS\_Env (0x4b25898), and playerB (0x4b22268). The diagram includes messages like mStart, mPing, mPong, and give, along with states like playing and RTDS\_idle. A dialog box titled 'RTDS - Project "pingpong.rdp" (modified)' is open, showing options for selecting MSC diagrams to compare. A text box within the dialog states: 'Options will be set up because the trace is more precise than the requirements.' The dialog also has checkboxes for 'Filter activated', 'Messages', 'Timer', 'State', 'Semaphore takes', 'Comment Symbol', 'General name area', and 'Time constraint', along with radio buttons for 'Show & Diff' and 'No show & No diff'.

- Anzeige von Zuständen, aber kein Vergleich

# Erzeugung einer Diff-Datei



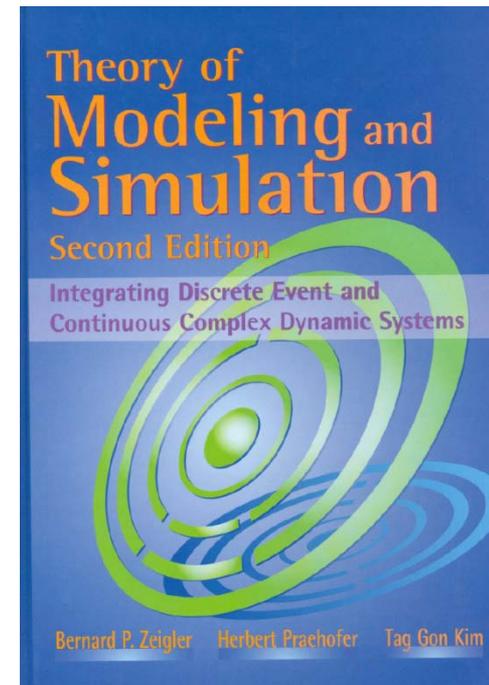
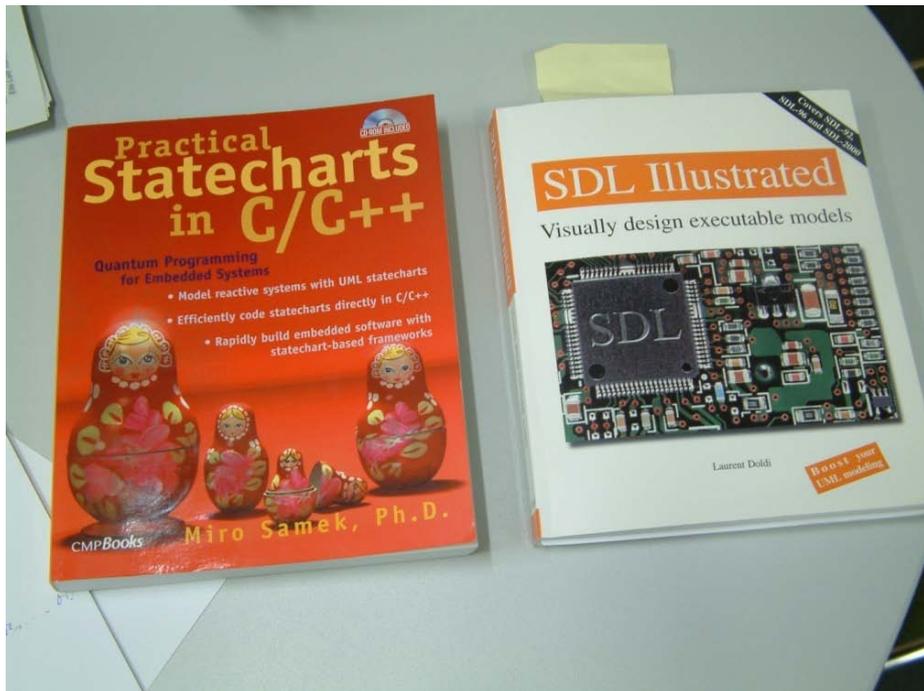
# Überdeckungsgraph

The screenshot shows a window titled "RTDS - Code coverage results 'Cod...". The window contains a menu bar (File, Edit, Windows, Help) and a toolbar with navigation and action icons. Below the toolbar is a table with two columns: "Agent/symbol" and "Hits". The table lists various code elements and their corresponding hit counts.

Agent/symbol	Hits
Deamon	0 - 6
-	1
evenState	0 - 6
tStateChange	0 - 6
sem(FOREVER6	
daemonIsEven6	
sem	6
tStateChange(6	
oddState	6
random->setL0	
-	0
oddState	5
GameServer	0
Monitor	0 - 1
-	1
waitForNewGame	0
gameOver(player 0	
newGame	0
Player	0 - 255

verlangt unter *Options* des Projekt-Managers gewisse Einstellungen, da Erstellung ressourcenaufwändig ist

# OMSI- Literaturhinweise



# 7. *SDL-Konzepte* (Präzisierung, 1. Teil)

1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Nachrichtenadressierung
4. Timer

# ***Vorschlag einer Namenskonvention***

... für erstes Zeichen in Namen in Abhängigkeit des Modellelementetyps:

- b Block-Namen
- p Process-Namen
- t Timer-Namen
- s Semaphore-Namen
- g Gate-Namen

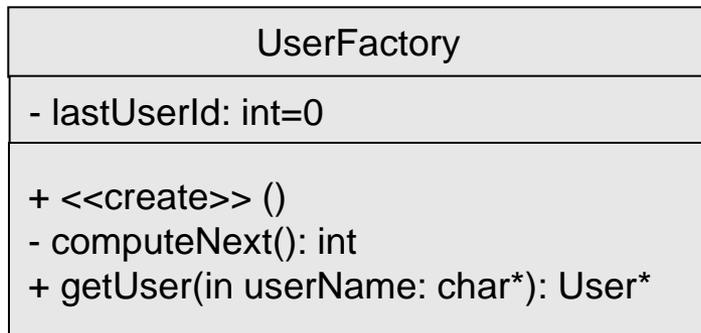
# Struktur / Architektur

- System - Environment  
(Blocksymbol wird als System benutzt)
- Agent ist Element eines Systems
  - zwei Ausprägungen:
    - Block (ohne besondere Ausprägung im Zielcode),
    - Process (OS-Prozesse/Thread)
  - Mix von Prozessen und Blöcken auf einer Ebene ist erlaubt
- Prozess-Kardinalität: Default (1, )

# Klassenbeschreibung

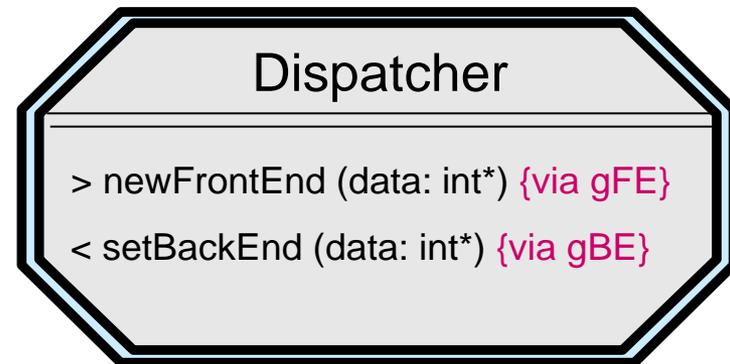
- 4 Formen
  - Interface-Klasse <<interface>>
  - System-Klasse <<system>>
  - Passive Klasse
  - Aktive Klasse : für Block-Typen, Process-Typen

In UML-Syntax



mehrere Konstruktoren (kein Return) möglich  
nur ein Destruktor (<<delete>>)  
In-, Out-, inOut-Parameter

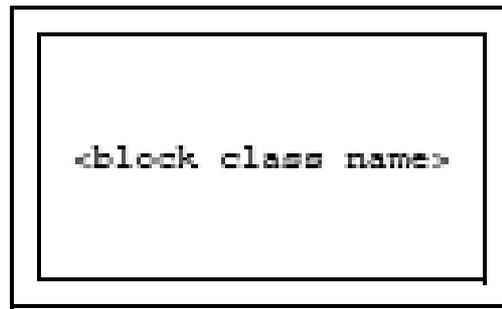
SDL-angepasste-Syntax



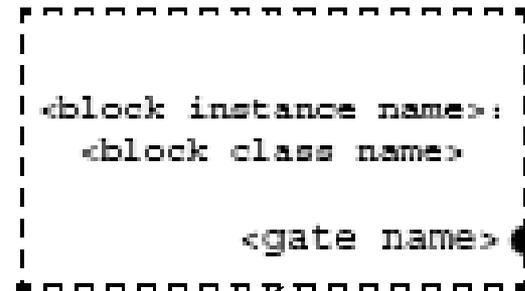
keine Attribute  
Operations → PseudoNachrichten (nurEinParameter)  
Property für Gate-Angabe

# SDL-RT: Block-Typen (Block-Klassen)

Einschränkungen gegenüber Z.100



keine Vererbung  
keine Virtualität  
keine Kontextparameter

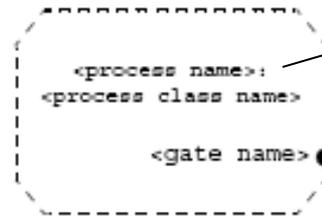


Instanzmengen sind nicht  
zulässig

Gates ohne Atleast-Typ-Angabe

# SDL-RT: Process-Typen (Process-Klassen)

Einschränkungen gegenüber Z.100



Instanzmengenbezeichner  
:  
Process-Typbezeichner

**Vererbung** eingeschränkt gegenüber Z.100

- Spezialisierung: Zustände und Transitionen wie Z.100
- aber keine Kontextparameter

Redefinition von

- Transitionen (alle sind implizit virtuell)
- Gates (Virtualität ist anzugeben)
- Datentypen

Abstrakte Process-Typen, falls

- abstrakte Trigger
- abstrakte Gates

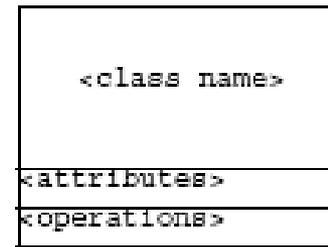
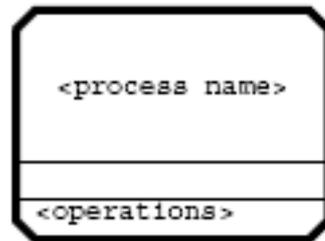
Präfix: ABSTRACT

Präfix: VIRTUAL

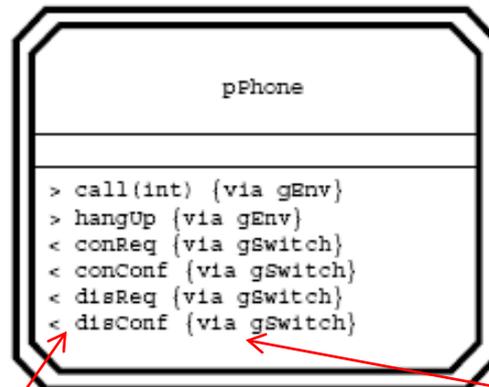
# UML-Klassendiagramme

- Aktive Klassen, passive Klassen

**vordefinierte Stereotypen**  
System, Block, Blockclass,  
Process, Processclass,



aktive Klassen haben keine Attribut-Kompartments (Attribute werden woanders definiert)  
Operationen sind ein- und ausgehende Nachrichten (asynchron) mit Gate-Angabe  
Leider: hier keine Angaben von Nachrichtenlisten erlaubt !!! (Grund unklar)

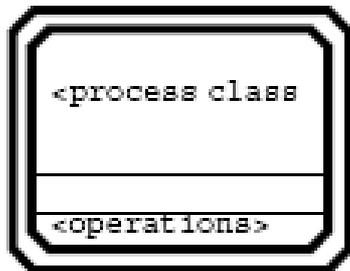


SDL-RT-Doko fehlerhaft:

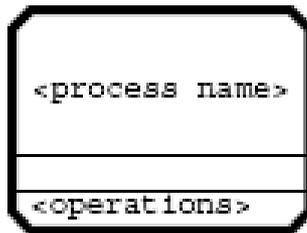
kein Leerzeichen

Doppelpunkt statt Leerzeichen

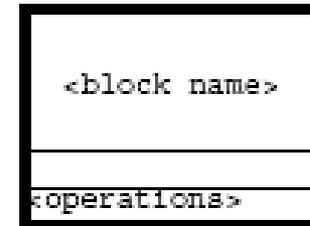
# Vordefinierte graphische Symbole von stereotypisierter Klassen



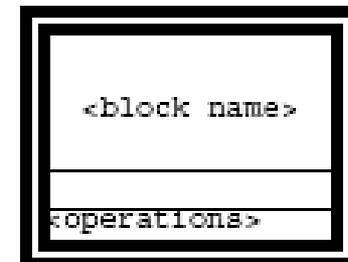
Class stereotyped as a class of process



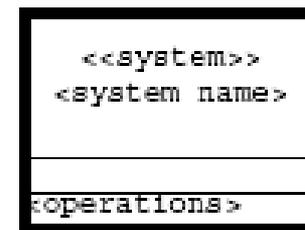
Class stereotyped as a process



Class stereotyped as a block



Class stereotyped as a class of block

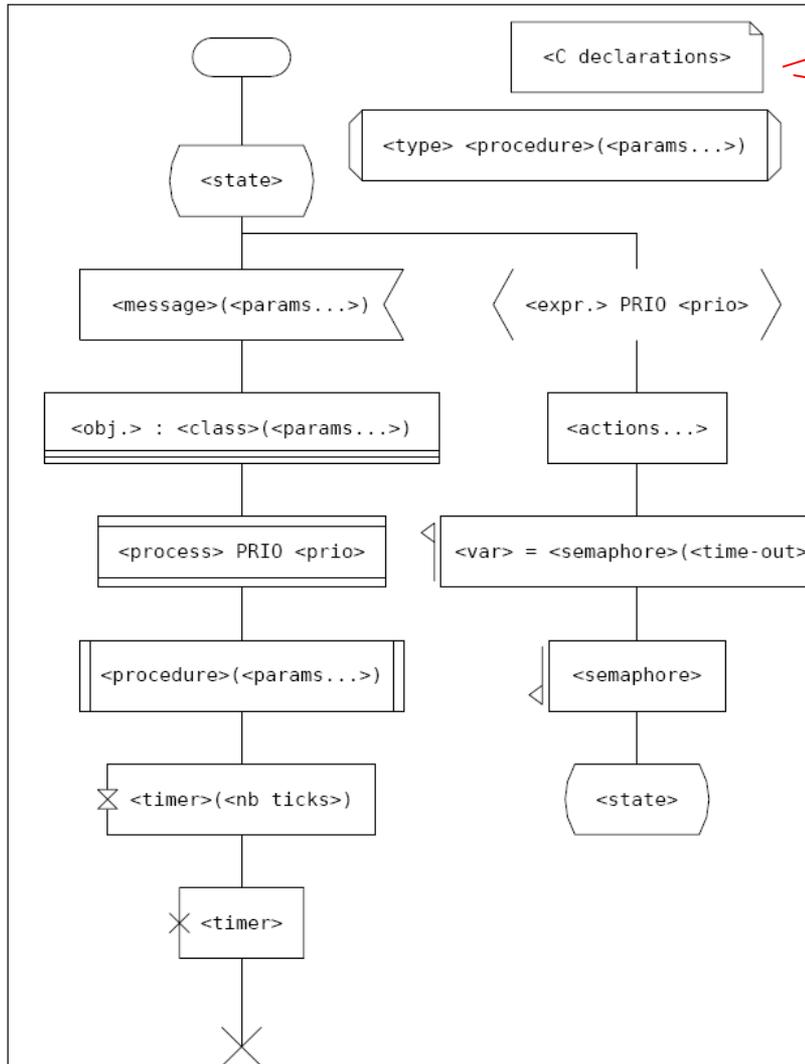


Class stereotyped as a system

# Datentypen

- zwei Möglichkeiten für Nutzung von C-Datentypen
  - externes Header-File (Projekt-Manager)  
`#include` dann auf Block- oder Process-Level möglich
  - SDL-Text-Box
    - Block-Level → daraus wird separate C-Header-Datei (globale Typen, **globale Variablen**)
    - Process-Level → für Process-lokale Variablen
    - Procedure-Level → für Prozedur-lokal Variablen

# Process-lokale Deklarationen



gestrichelt: SDL-Deklaration

voll: C-Deklaration

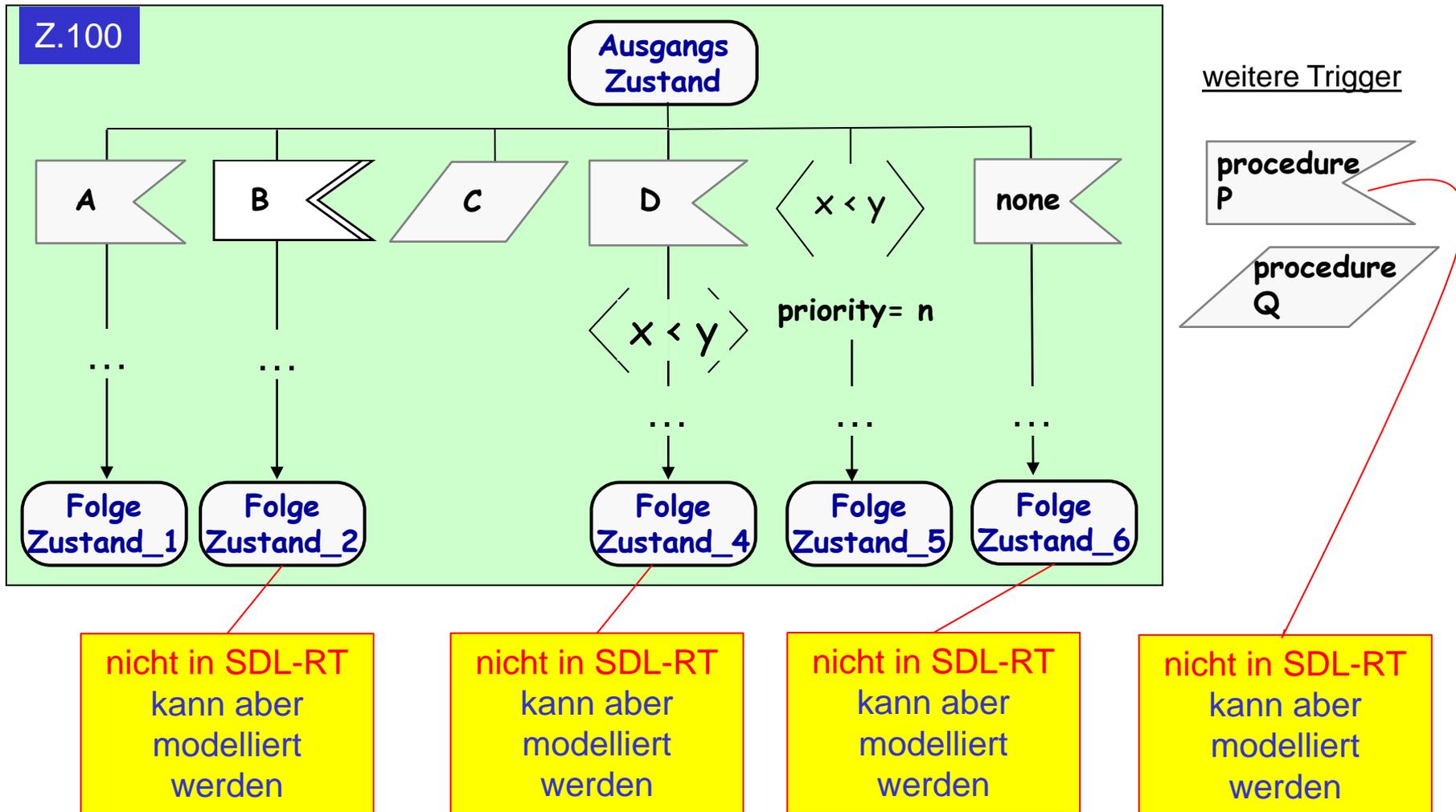
# SDL-Signale und Signallisten

- **Message** statt Signal
- MESSAGE msg1, msg2(int),  
msg3(char\*, struct MyType\*, double)
  - Parameter: gültige C-Typen (auch Zeiger)
- MESSAGE\_LIST myList= ((subList), msg1, msg3);
- implizite Deklaration von
  - Timern und
  - Timeout-Nachrichten

## 7. *SDL-Konzepte* (Präzisierung, 1. Teil)

1. Modellstruktur
2. Einfacher Zustandsautomat: Trigger-Arten
3. Nachrichtenadressierung
4. Timer

# Die Triggervarianten von SDL im Überblick



# Normale Signaleingabe: Input ohne Parameter

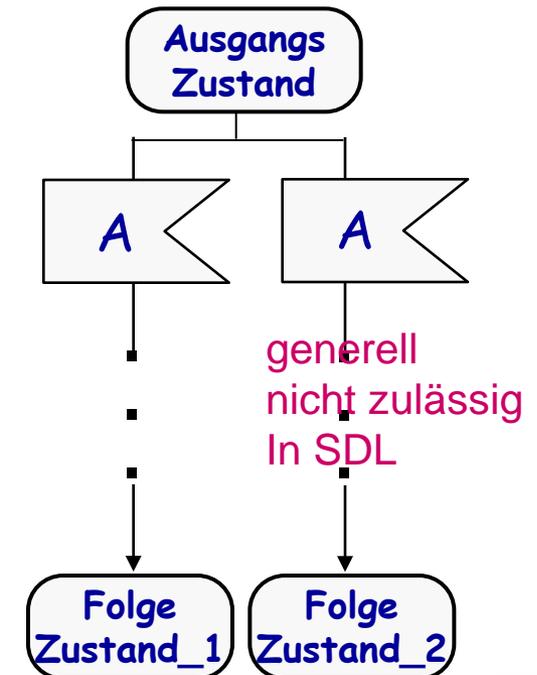


`state Ausgangszustand;`  
`input A;`

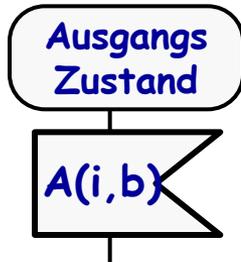
## Achtung:

ein Signal darf für je ein Ausgangszustand nicht mehr als einmal als Trigger vorgesehen werden

- der Übergang aus *Ausgangszustand* (in dem die Prozessinstanz verharrt), wird ausgeführt, sobald *A* zum ersten Signal im Puffer wird
- das Signal *A* wird konsumiert (d.h. Parameter werden evtl. übernommen, SENDER wird gesetzt, Signal wird im Puffer gestrichen)
- hat ein Prozess ein Signal unverändert weiterzugeben, sind zunächst alle Parameter zu kopieren



# Normale Signaleingabe: mit Parameterübernahme

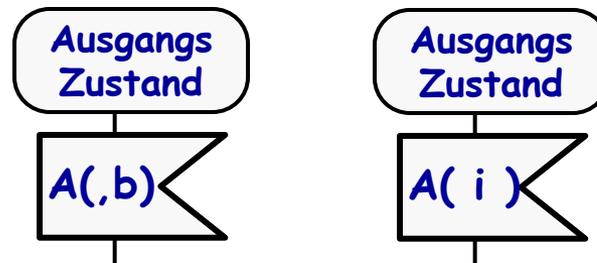


```
state Ausgangszustand;  
input A (i, b);
```

```
MESSAGE A (int, bool);  
/* beliebige Signatur */
```

```
int i,  
bool b;
```

- beide Parameter werden übernommen (d.h. in die Variablen *i* und *b* kopiert)
- Voraussetzung dafür ist, dass *i* und *b* lokale Variablen des Empfängerprozesses sind und zuweisungs-kompatible Typen repräsentieren
- **sender**-Variable wird gesetzt



**Achtung:**  
eine partielle Übernahme der Parameter ist möglich

- hat ein Prozess ein Signal unverändert weiterzugeben, sind zunächst alle Parameter zu kopieren

# Signalzurückstellung: Save



- das aktuelle Signal **A** wird zurückgestellt, das nachfolgende Signal wird damit zum neuen aktuellen Signal
- ist **A** das einzige Signal, verharret der Prozess im Ausgangszustand bis ein weiteres Signal eintrifft
- sobald ein neu eingetroffenes Signal zu einem Zustandsübergang geführt haben sollte, ist die Zurückstellung von **A** wieder aufgehoben
- weder die Parameter von **A** werden kopiert, noch wird **sender** aktualisiert

# Save- Semantik

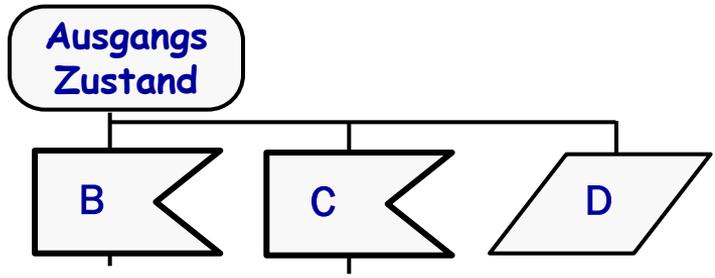
## Z.100

The saved signals are retained in the input port in the order of their arrival.

The effect of the save is valid only for the state to which the save is attached.

In the following state, signal instances that have been "saved" are treated as normal signals

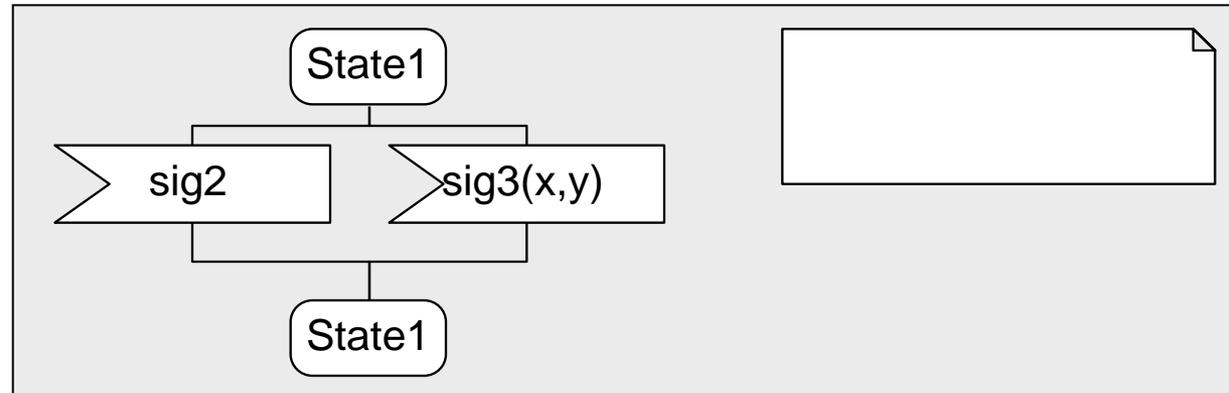
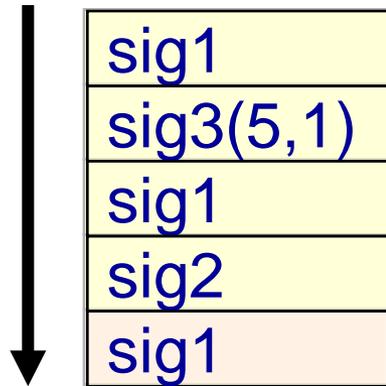
# Signalverwerfung: implizites Discard



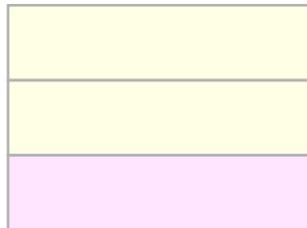
```
MESSAGE A (int, bool);  
/* beliebige Signatur */
```

- Sollte für einen Zustand, indem sich der Prozess befindet, **kein Trigger** für das aktuelle Signal vorgesehen sein, wird es (ohne Kopie seiner expliziten und impliziten Parameter) **verworfen**
- o.B.d.A.: sei *A* dieses Signal für *Ausgangszustand*, wird dieses (und nur dieses) *A* verworfen

# Trigger-Beispiel-1



nachher (im Zustand *State1*)

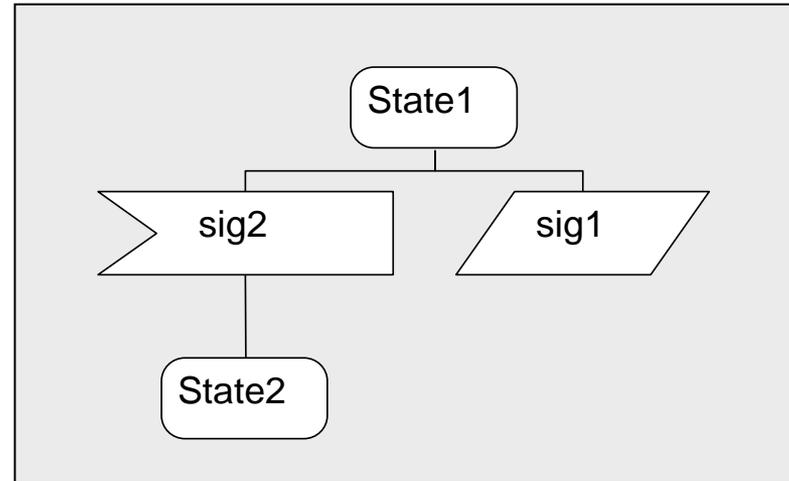
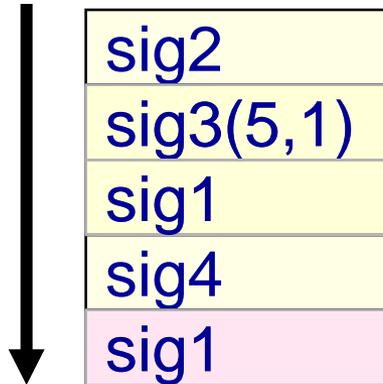


1. sig1 → discard
2. sig2 → input
3. sig1 → discard
4. sig3 → input mit Parameterübergabe
5. sig1 → discard

**sender**= PId-Wert des Senders von **sig3**

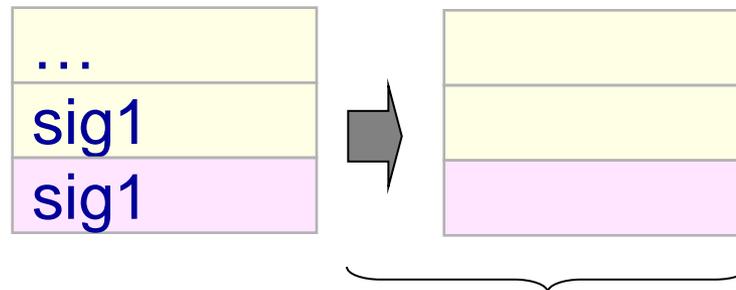
# Trigger-Beispiel-2

vorher



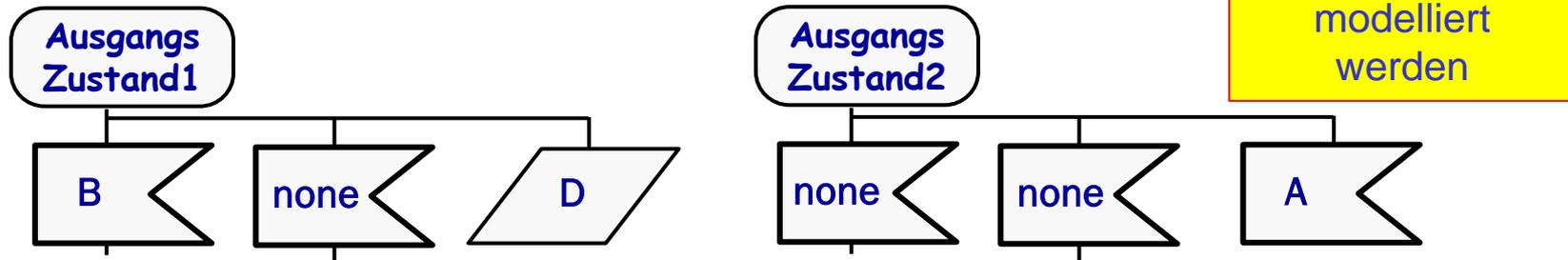
1. sig1 → save
2. sig4 → discard
3. sig1 → save
4. sig3 → discard
5. sig2 → input

nachher (im Zustand *State2*)



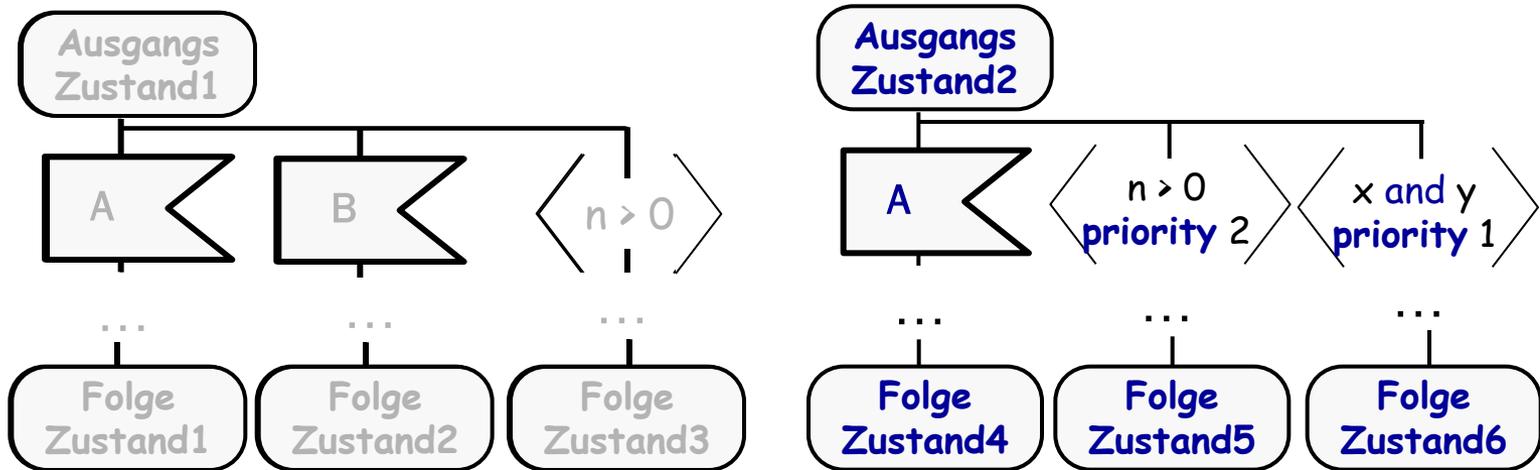
falls keine weiteren Übergänge in *State2*

# Spontaner Zustandsübergang



- ein spontaner Übergang benötigt **keine** bestimmte Belegung des Signalpuffers – er kann auch erfolgen, wenn der Puffer leer ist
- ob und wann der spontane Übergang vollzogen wird, ist **nichtdeterministisch** bestimmt. Mit Cinderella (Werkzeug) lässt sich eine Wahrscheinlichkeit einstellen.
- der **sender**-Wert nach einem spontanen Übergang ist **null**
- der spontane Übergang wird benötigt, um Störeinflüsse auf das Verhalten eines Systems nachbilden zu können
- mehrere spontane Übergänge für einen Zustand sind zulässig, **maximal** kann aber nur **einer** bei einem Zustandsübergang zur Realisierung kommen

# Trigger: Continuous-Signal



der Übergang aus *Ausgangszustand1* in *FolgeZustand3* wird ausgeführt

- falls  $n > 0$  ist ( $n$  als lokale Variable des Prozesses) und
- sich weder  $A$  noch  $B$  im Puffer befindet

→ normale Input-Trigger sind höher priorisiert als Continuous-Trigger

der Übergang aus *Ausgangszustand2* in *FolgeZustand6* wird ausgeführt

- wenn sich  $A$  **nicht** im Puffer befindet und  $(x \text{ and } y)$  den Wert **true** liefert
- der Übergang wird **unabhängig** vom  $n$ -Wert ausgeführt

→ niedrigster Prioritätswert bedeutet höchste Priorität

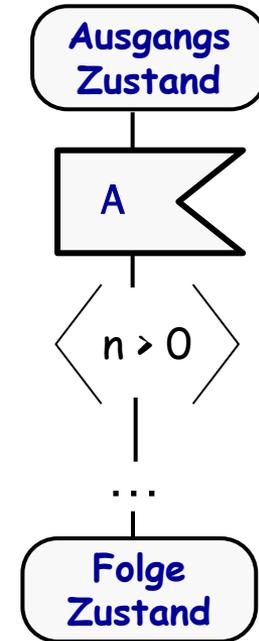
# Trigger: Input mit Bedingung

der normale Input-Trigger kann gesteuert werden

- der Übergang kann nur vollzogen werden, wenn **A** das aktuelle Signal im Puffer **und** die zusätzliche Bedingung erfüllt ist
- da die Bedingung nur von lokalen Variablen abhängig ist, besteht eine reale **Deadlock**-Gefahr für den Prozess

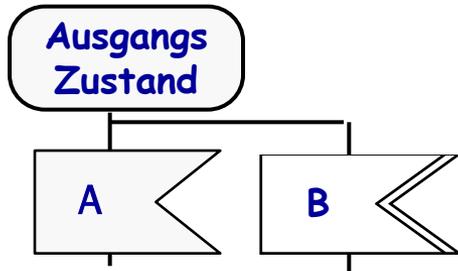
→ **Lösung**: alternative Transitionen über

- normale Inputs
- priorisierte Input



**Achtung:** Eine Bedingung kann nicht über explizite oder implizite Signalparameter gebildet werden – wenn doch, dann über die Werte des zuletzt ausgeführten Zustandsübergangs

# Priorisierter Zustandsübergang



- der Übergang, ausgelöst durch ein Signal **B** im Ausgangszustand, ist gegenüber allen anderen **priorisiert**, unabhängig von der Position von **B** im Puffer  
**A** behält seine Position im Puffer bei
- der normale Input bei Konsumtion von **A** kommt im Ausgangszustand nur dann zum Ausführung, wenn sich **keine** Signale **B** im Puffer befinden

**Achtung:** SDL kennt keine Signalpriorisierung,  
dafür aber Trigger-Priorisierung

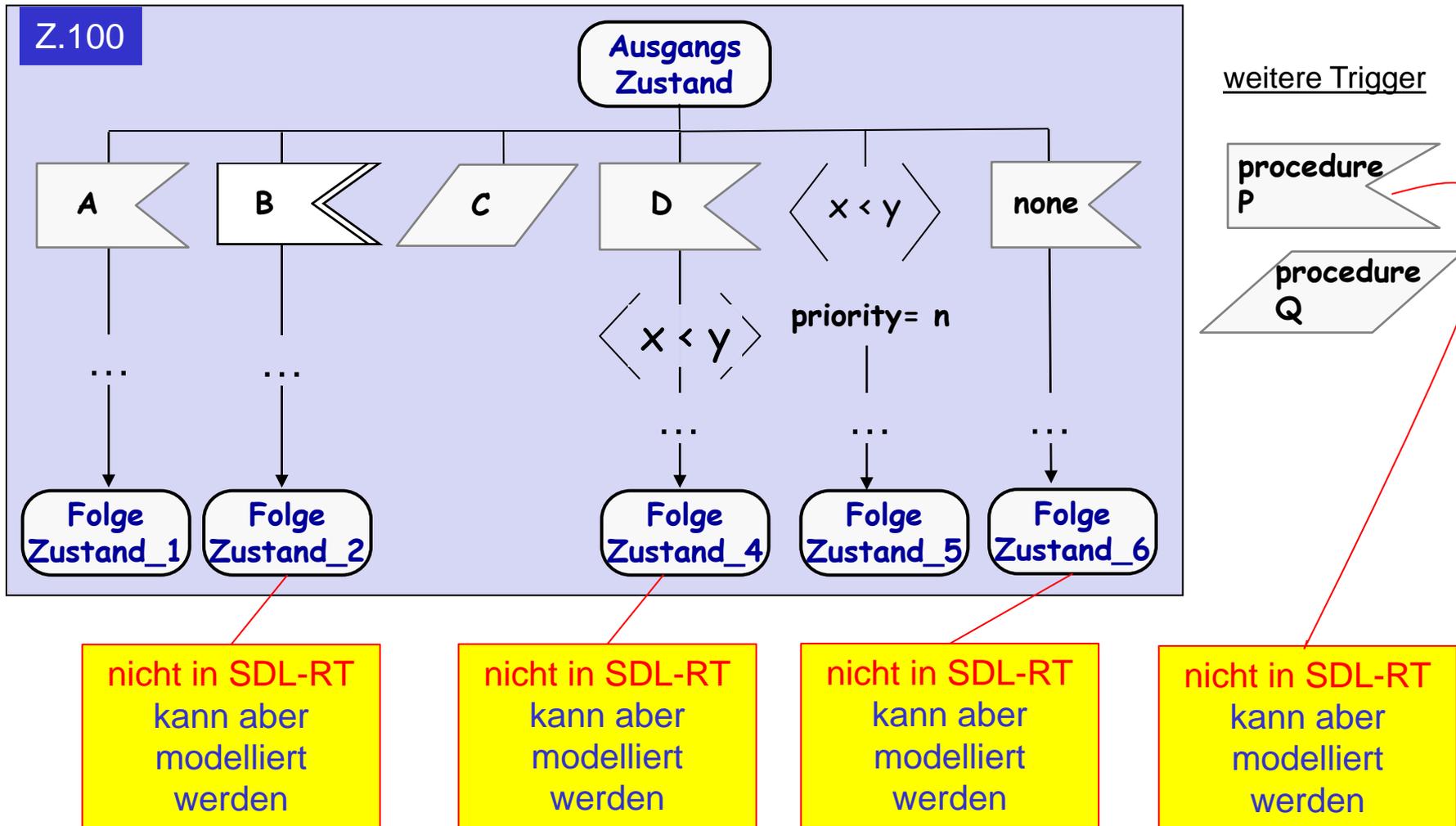
Rangfolge:

- (1) Priorisierter Input
- (2) Normaler Input
- (3) Input mit Bedingung
- (4) Continuous-Signal (mit Prioritätsklassen)

bei Gleichheit: erfolgt eine nichtdeterminierte Auswahl

} überlagert  
durch  
spontane  
Trigger

# Die Triggervarianten von SDL im Überblick

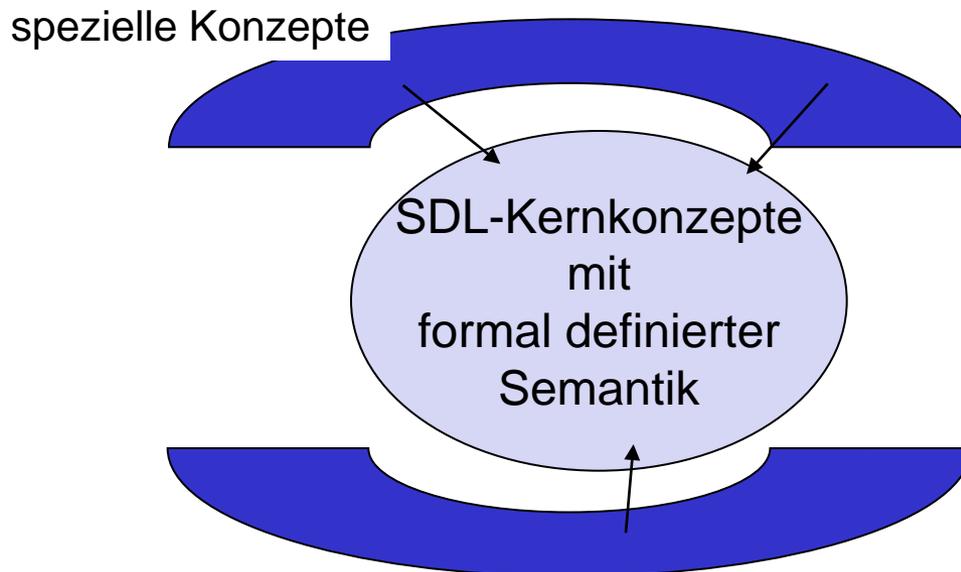


nicht in SDL-RT  
kann aber  
modelliert  
werden

# Semantik-Definition der Trigger

Beherrschung der semantischen Komplexität

- Mehrzahl neuer Konzepte in **SDL-92/96** wird durch Transformation auf Kernkonzepte definiert



## Beispiel:

- Priority Input,
- Continuous Signal Input,
- Input mit Vorbedingung
- (und RemoteProcedure-Input/Save)

werden auf

- normalen Input,
- Save,
- Discard
- (und gewöhnliche Prozeduren)

zurückgeführt

## 7. *SDL-Konzepte* (Präzisierung, 1. Teil)

1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Nachrichtenadressierung
4. Timer

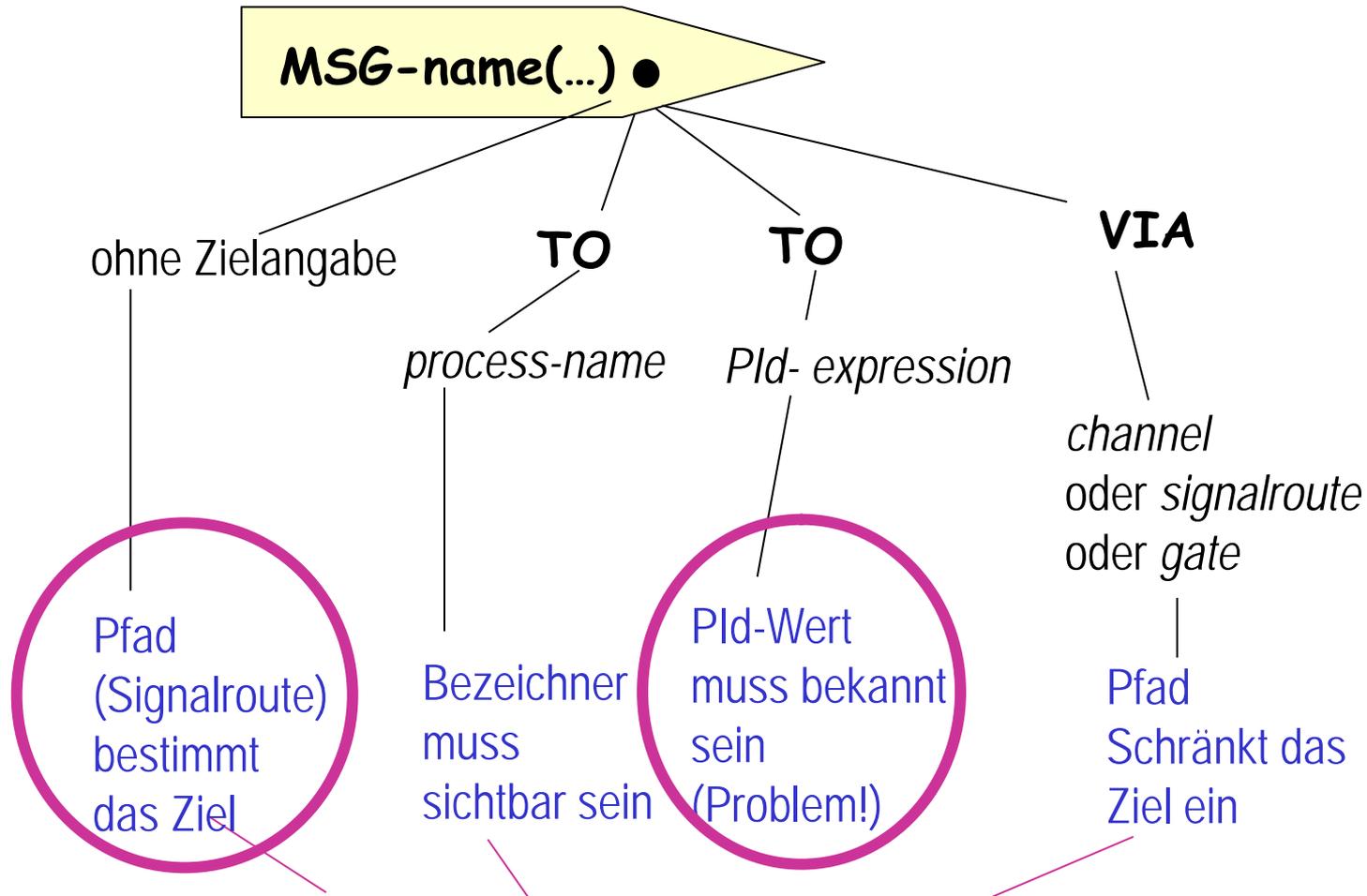
# Zwei Adressierungsarten

(1) **OUTPUT** *msg-name (param-List)* **TO\_X** *receiver*

(2) **OUTPUT** *msg-name (param-List)* **VIA** *medium*

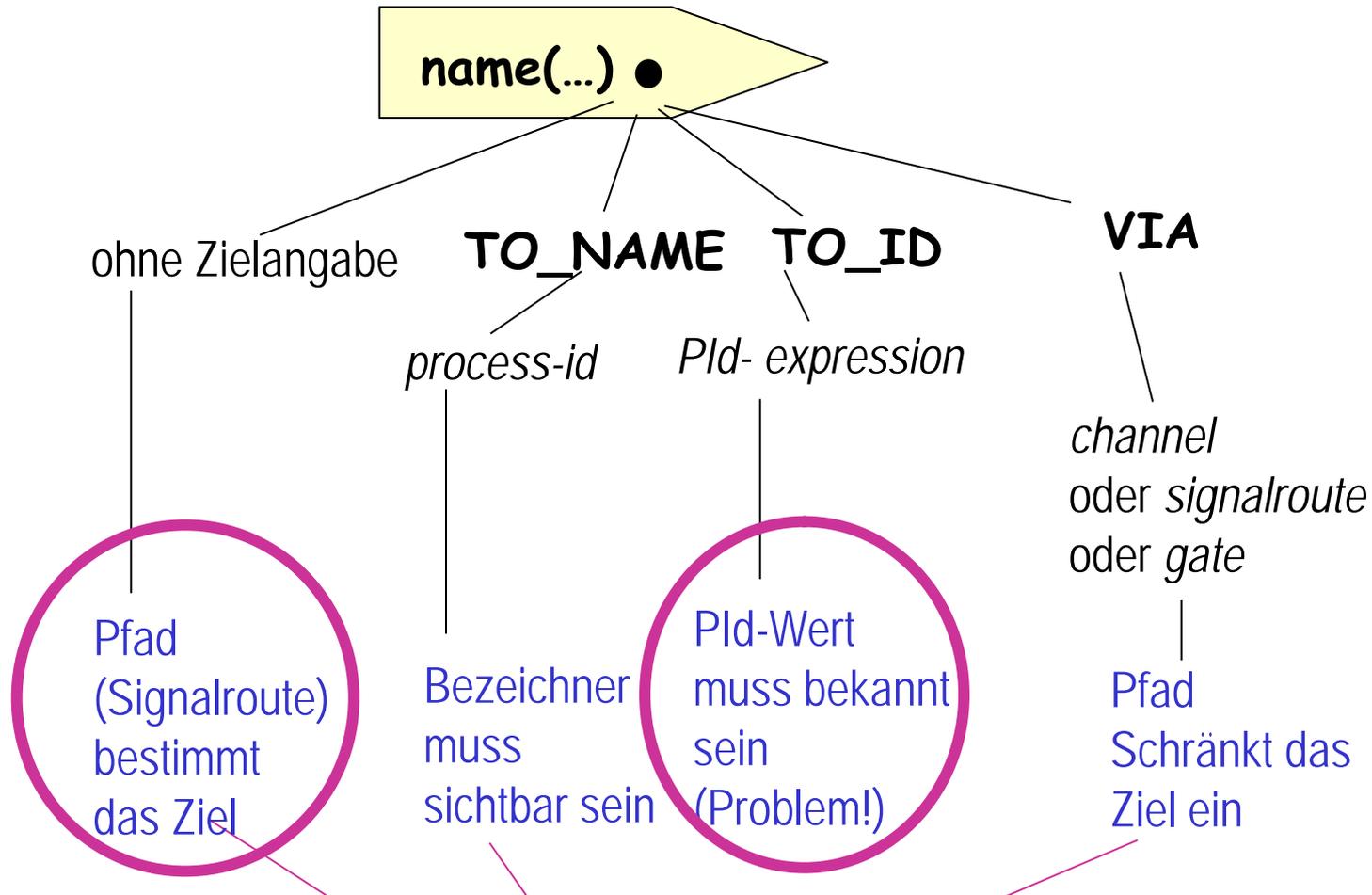
- Parameter: Werte, Referenzen  
bei Referenzen ist wichtig: Welcher Prozess legt Lebenslauf fest.  
häufig:
  - Sender besitzt Datum nicht mehr, nachdem gesendet worden ist,
  - Empfänger gibt Speicherplatz frei (daraus entstehen Probleme)
- **TO\_X**:
  - **ID**: PARENT, SELF, OFFSPRING, SENDER, oder RTDS\_QueueId
  - **NAME**: erster generierter Prozess einer Menge  
(nicht benutzen, wenn Name nicht 1-deutig)
  - **ENV**
- **medium**:
  - Name von Channel oder Gate (verbunden mit aktuellem Prozess)  
**keine Mehrdeutigkeit zulässig!**

# SDL-Standard: Nachrichten-Adressierungsarten



bei Mehrdeutigkeiten der Empfänger-Instanz erfolgt eine nichtdeterminierte Auswahl

# SDL-RT: Nachrichten-Adressierungsarten



bei Mehrdeutigkeiten der Empfänger-Instanz erfolgt eine nichtdeterminierte Auswahl

# SDL-RT: Zeiger als Parameter

```
MESSAGE
  ConReq(unsigned char *),
  ConConf,
  DisReq(myStruct *);
```

```
long          myDataLength;
unsigned char  *myData;
myStruct      *pData;
```

```
ConReq
(myDataLen,
myData)
```

```
ConConf
```

```
DisReq
(pData)
```

**unsigned char\*** man kann optional auf die Länge zugreifen  
(ist als erster Parameter zu übergeben)

# SDL/RT- Adressierungsarten

```
MESSAGE  
  ConReq(unsigned char *),  
  ConConf,  
  DisReq(myStruct *);
```

```
long          myDataLength;  
unsigned char *myData;  
myStruct      *pData;
```

```
ConReq  
(256, myData)  
TO_ID PARENT
```

```
ConConf TO_ID  
aCalculatedReceiver
```

```
DisReq  
(pData) TO_ID  
PARENT
```

Adressierungsart **TO\_NAME**: wird rudimentär unterstützt

Spezifische Adressierungsklauseln:

- TO\_ID (PID-Ausdruck)
- TO\_NAME
- TO\_ENV

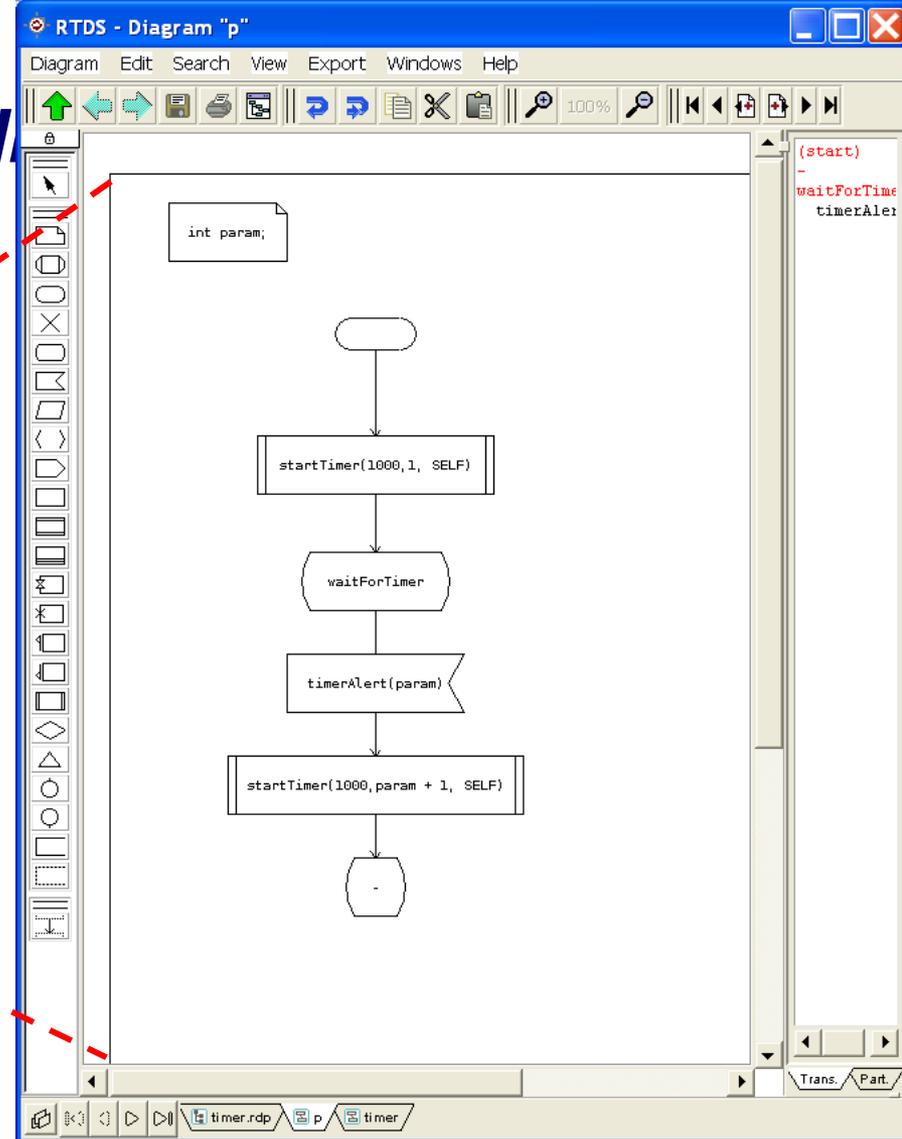
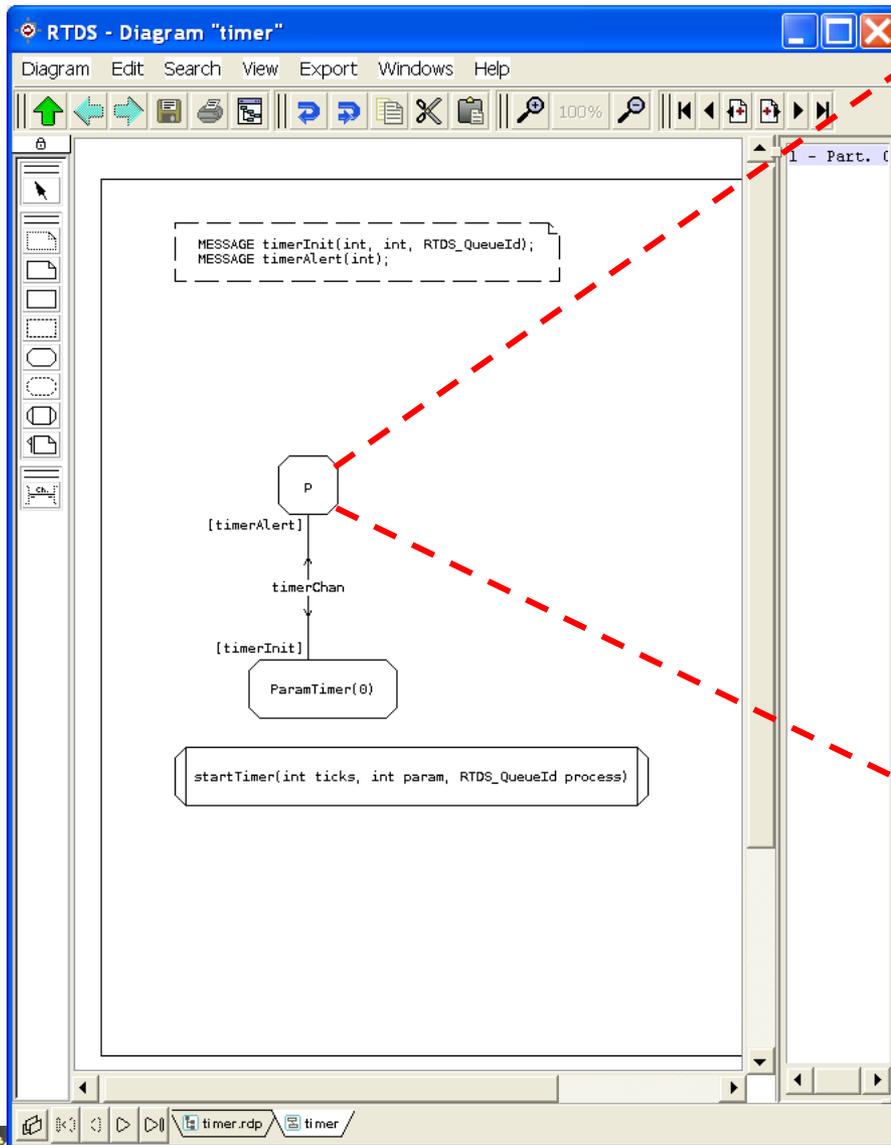
## 7. *SDL-Konzepte* (Präzisierung, 1. Teil)

1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Nachrichtenadressierung
4. Timer

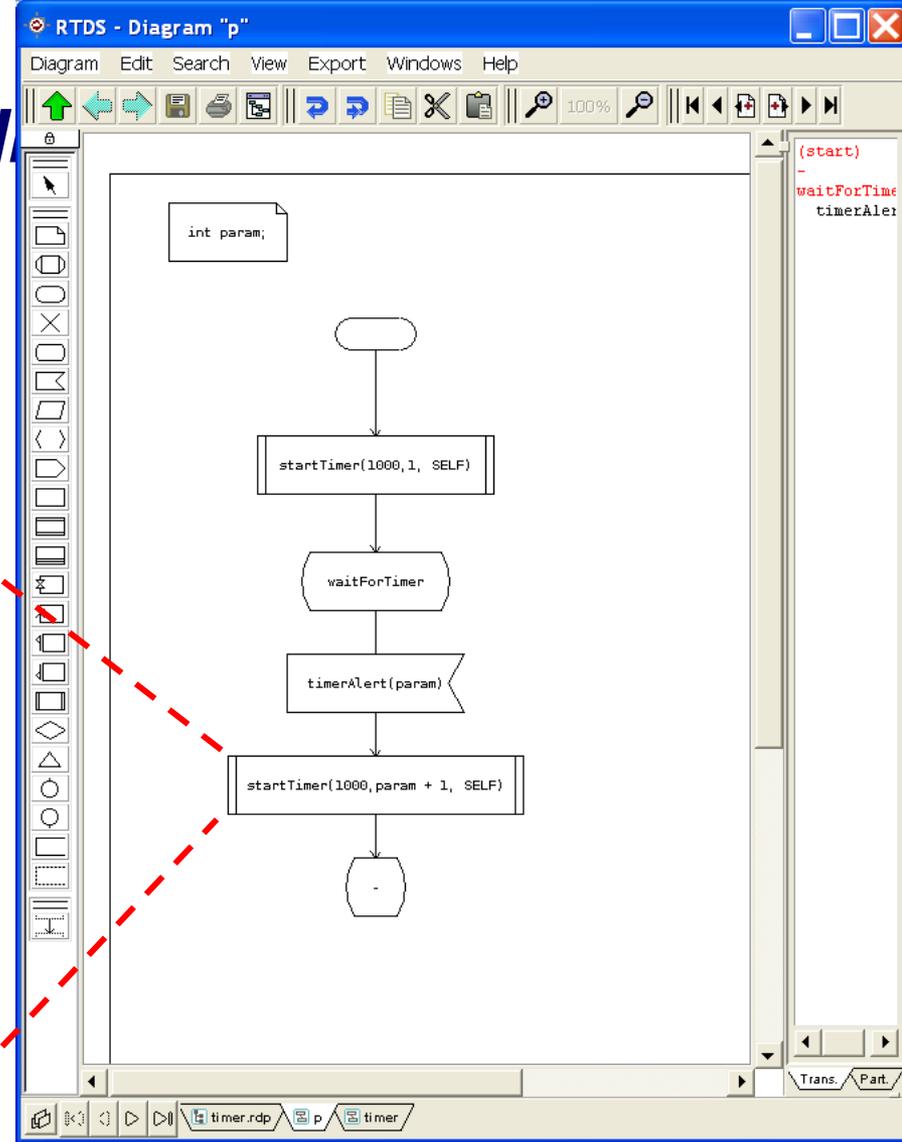
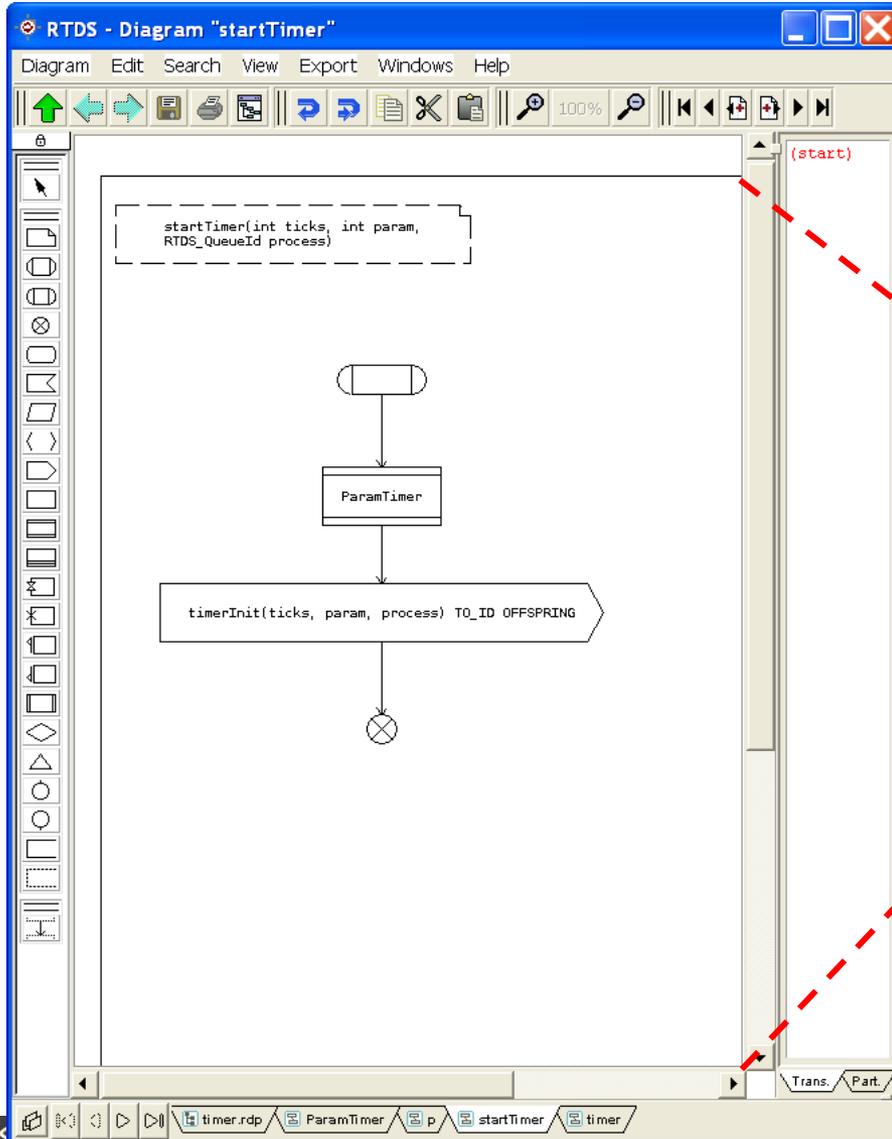
# Timer-Konzept

- Grundkonzept zur Verhinderung von Deadlocks und Livelocks in realen verteilten Systemen
  - Starten: **start/restart** bei Angabe der Timeoutzeit
  - Stoppen: **stop**
  - Timeout: **Timeout-Nachrichten** unterscheidbar durch Namen des auslösenden Timers
  - Jeder Prozess kann überbeliebig viele Timer verfügen
- Z.100-SDL erlaubt Parametrisierung/Indizierung von Timern zur Identifikation (SDL-RT nicht)
- In SDL/RT bleibt nur ein nutzereigenes Ersetzungsmodell

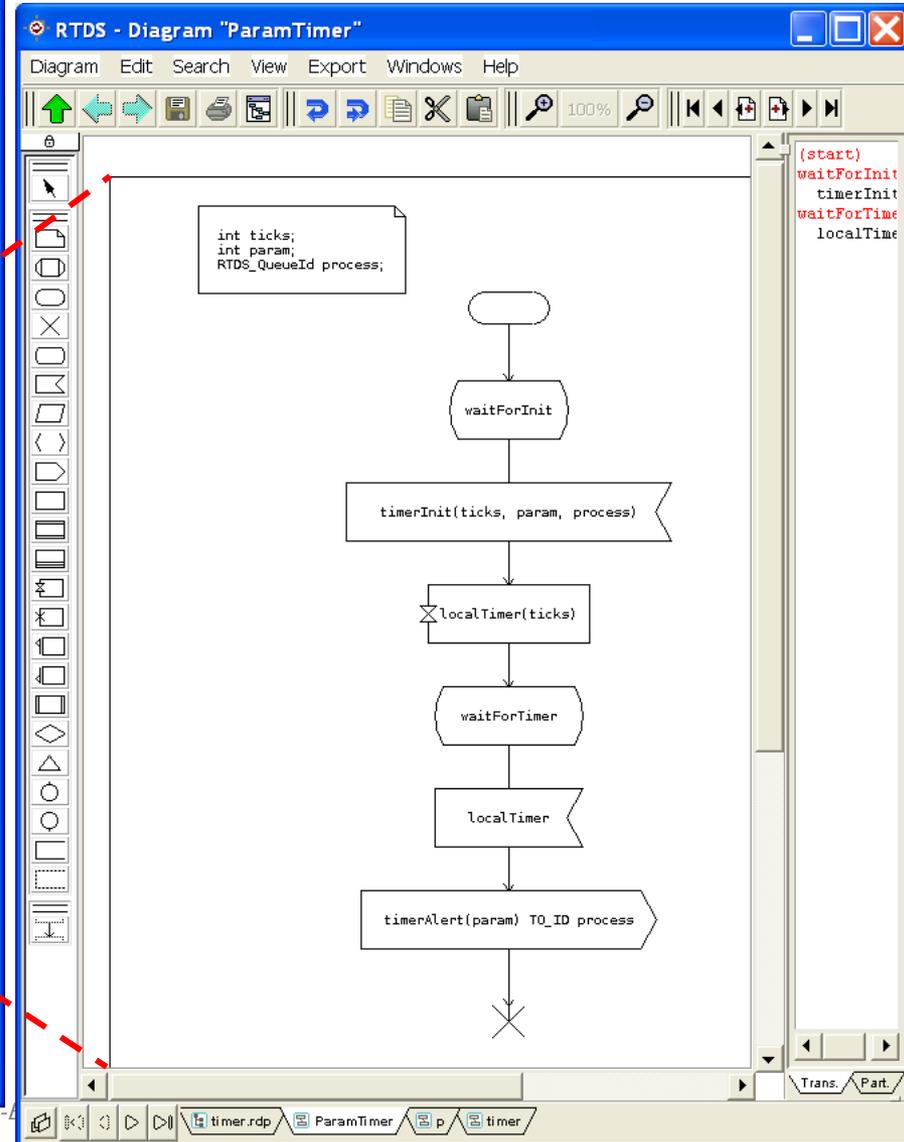
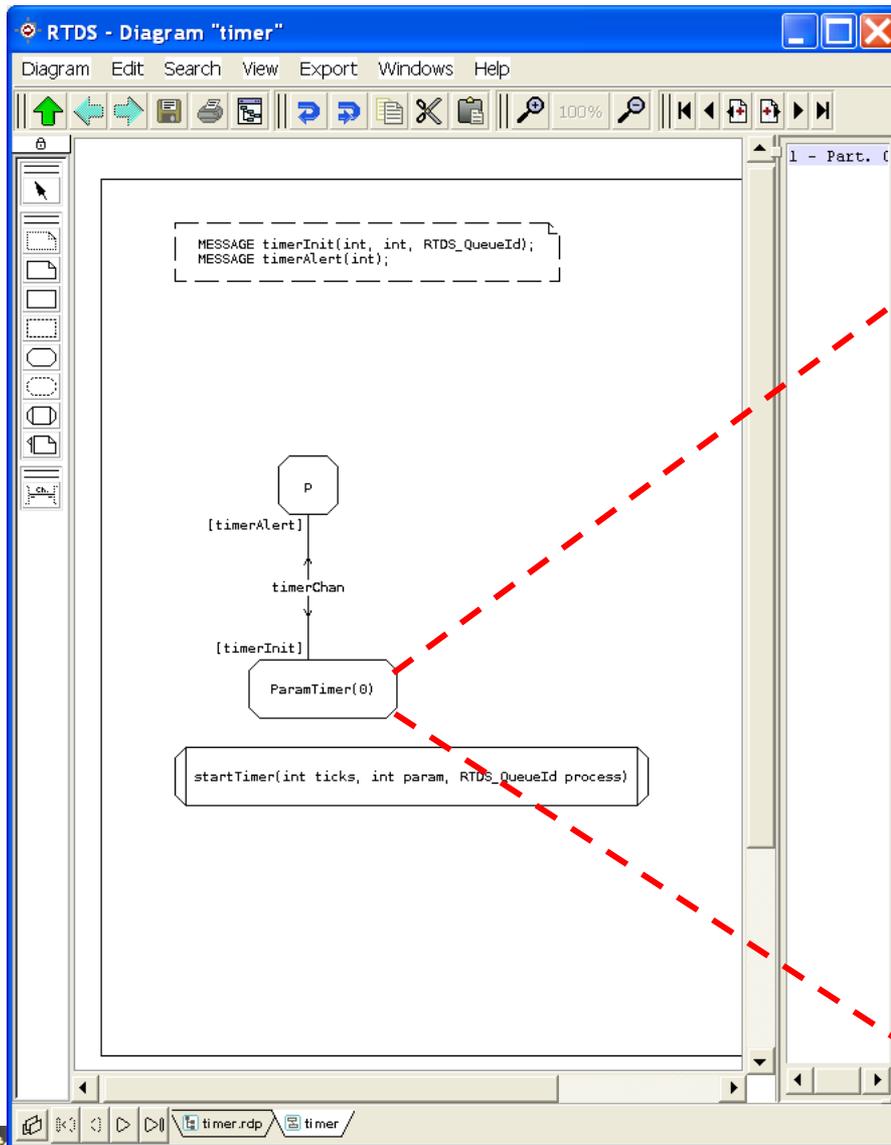
# Timer als eigenständ



# Timer als eigenständ



# Timer als eigenständiger Prozess



## 8. *Protokollentwicklung in SDL*

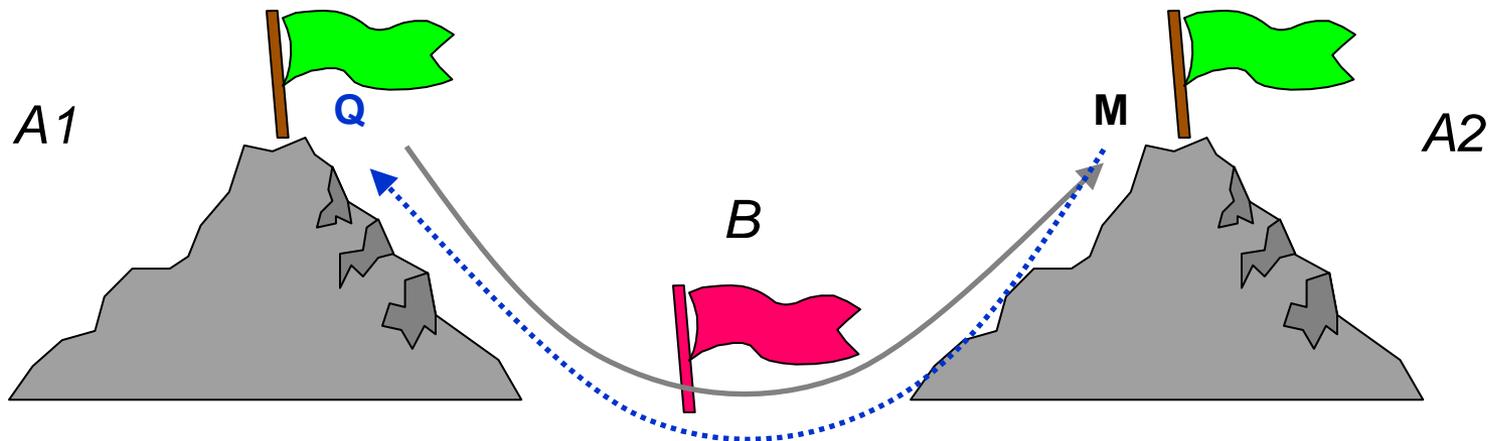
- OSI-Schichtenmodell (Konzept)
- InRes-Protokoll (Pseudo-Rechnernetzprotokoll)
- Umsetzung in SDL/RT
- weiterer Ausbau

# Protokolle im Altertum

Übertragung von Nachrichten über unsichere „Kanäle“  
bei Quittieren dieser Nachrichten

## Ziel

Absprache eines gemeinsamen Angriffstermins der verteilten grünen Armee



**Fall:** A1 erhält tatsächlich eine Bestätigung Q für Nachricht M von A2  
Wie sicher können sich A1 und A2 sein, dass Termin gilt ?

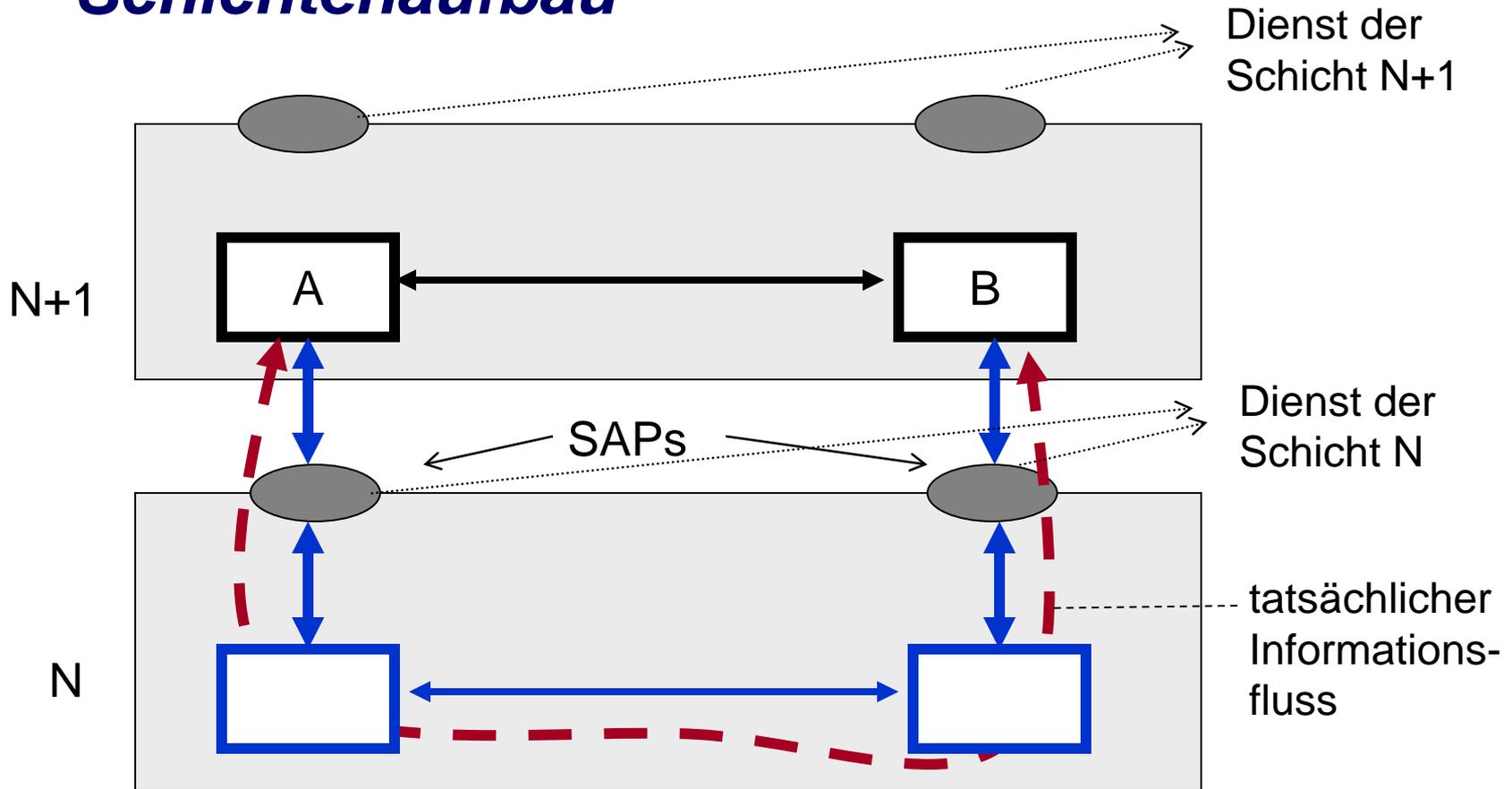
# Protokoll (Protocol)

- Regeln zur Kommunikation zwischen Partnerinstanzen innerhalb einer Kommunikationsebene
  - Wer muss wann aus welchem Grund eine Information an seinen Partner schicken
  - Vereinbarungen von Struktur und Codierungen der auszutauschenden Informationen
- Schichten-Struktur zur Beherrschung komplexer Übertragungsdienste für verbundene Rechner (als Knoten in einem Rechnernetz)
  - in realen Systemen gibt es nur eine einzige physisch vorhandene horizontale Verbindung, die anderen sind abstrakt / virtuell
- ISO-Standard zum Aufbau von Kommunikationsprotokollen: **Open System Interconnection**
  - Konzepte unabhängig von ihrer Implementierung)

# Dienst (Service)

- Eine Partnerinstanz einer Schicht N (*layer N*) bietet der darüber liegenden Schicht (*layer N+1*) eine Dienstleistung (*Service*)
- Kommandos zur Dienstinanspruchnahme heißen Dienstelemente/Dienstprimitive (*Service Primitives*, SP)
- Schnittstelle einer Schicht bzgl. eines Dienstes heißt Dienstzugangspunkt (*Service Access Point*, SAP)
- es besteht eine vertikale Instanzbeziehung zwischen den Dienst-Beteiligten unterschiedlicher Schichten (Dienstverbinder-Dienstnutzer)

# Schichtenaufbau

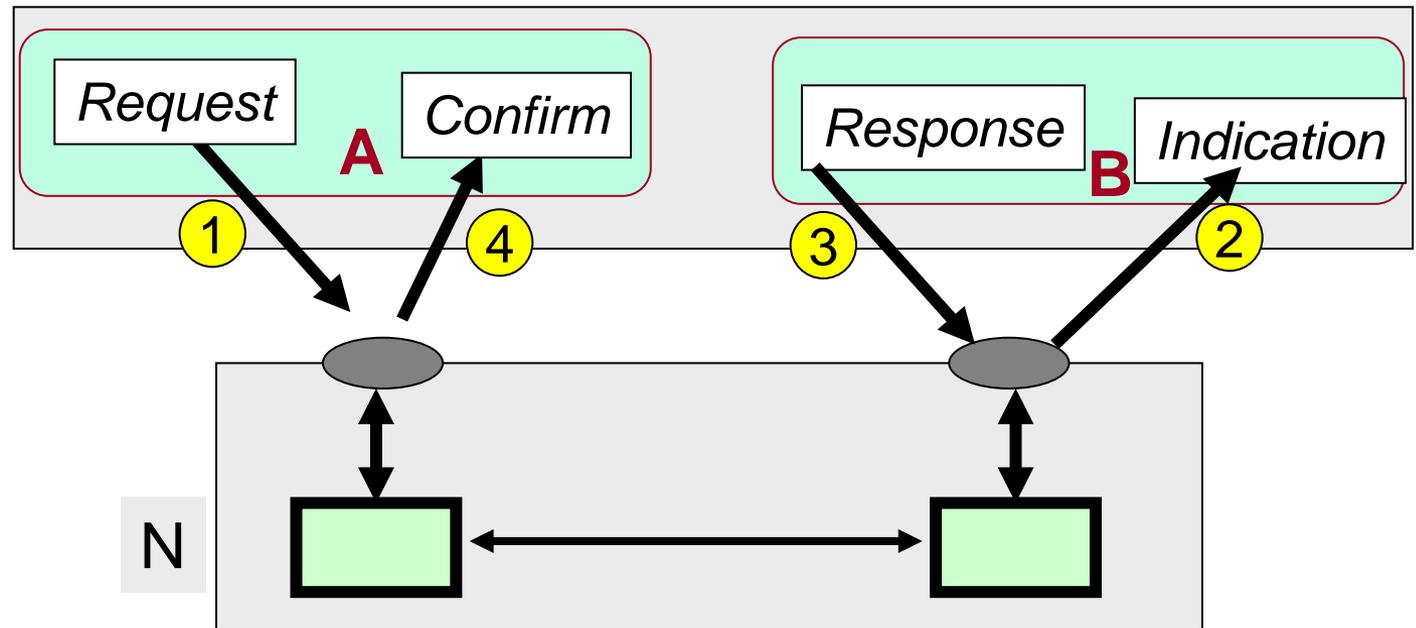


- A und B sind Kommunikationspartner
- Sie haben komplexe Informationen auszutauschen **ohne** physische Verbindung – sie vereinbaren ein Protokoll
- Sie sind Nutzer von Diensten der Schicht N, die physische Übertragung elementarer Informationen unterstützt

# Identifikation von Dienstprimitiv-Klassen

**vier** Grundtypen von Aktionen bei **einer** gerichteten Kommunikation zwischen **zwei** Partnern

**Beispiel:** A ist Auslöser, sein Anliegen wird von B bestätigt



# Identifikation von Dienstprimitiv-Klassen

**Beispiel:** B ist Auslöser, sein Anliegen soll von A nicht bestätigt werden

