Modellbasierte Entwicklung von Technologien für selbstorganisierende dezentrale Informations-systeme im Katastrophen-management

METRIK

Projekt Erdbebenfrühwarnung
im SoSe 2011

HUMBOLDT-UNIVERSITÄT ZU BERLIN

Entwicklung verteilter echtzeitfähiger Sensorsysteme

Joachim Fischer
Klaus Ahrens
Ingmar Eveslage

fischer|ahrens|eveslage@informatik.hu-berlin.de

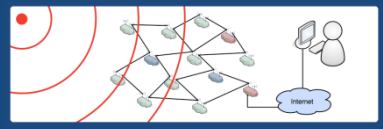
SAFER
Seismic eArly warning For EuRope

EDIM

SOSEWIN-extented

GFZ
Helmholtz-Zentrum
POTSDAM

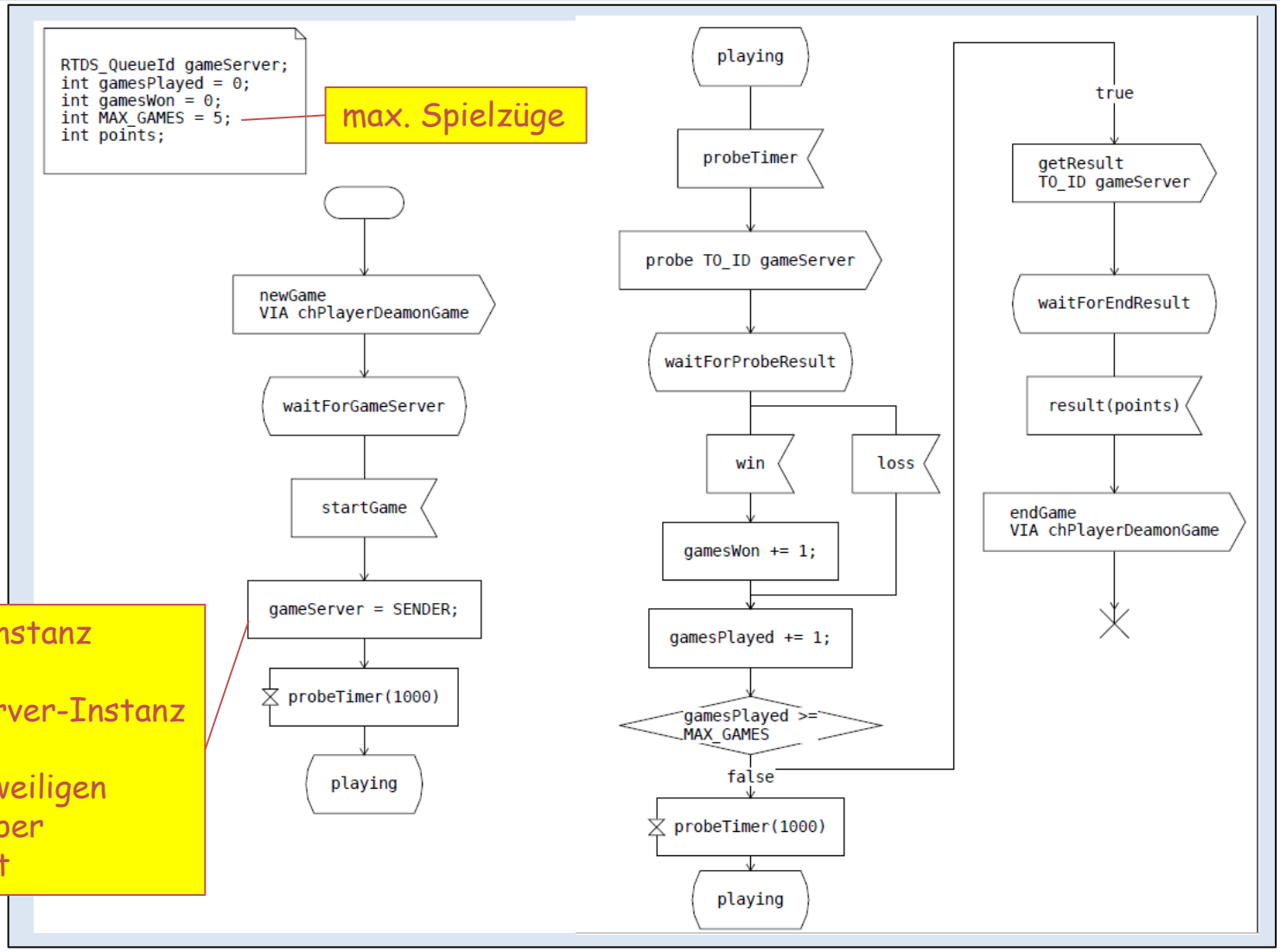
Systemanalyse

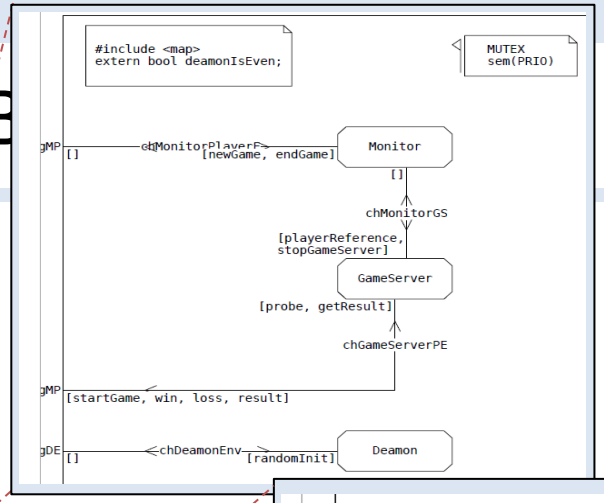


6. SDL

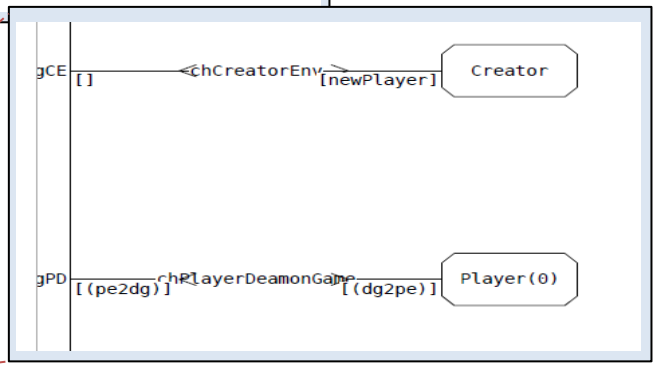
1. Grundphilosophie
2. ITU-Standard Z.100
3. Werkzeuge
4. SDL-Grundkonzepte
5. Musterbeispiel (in UML-Strukturen)
6. Umsetzung in SDL-RT

Prototypinstanz der Prozessmenge Player

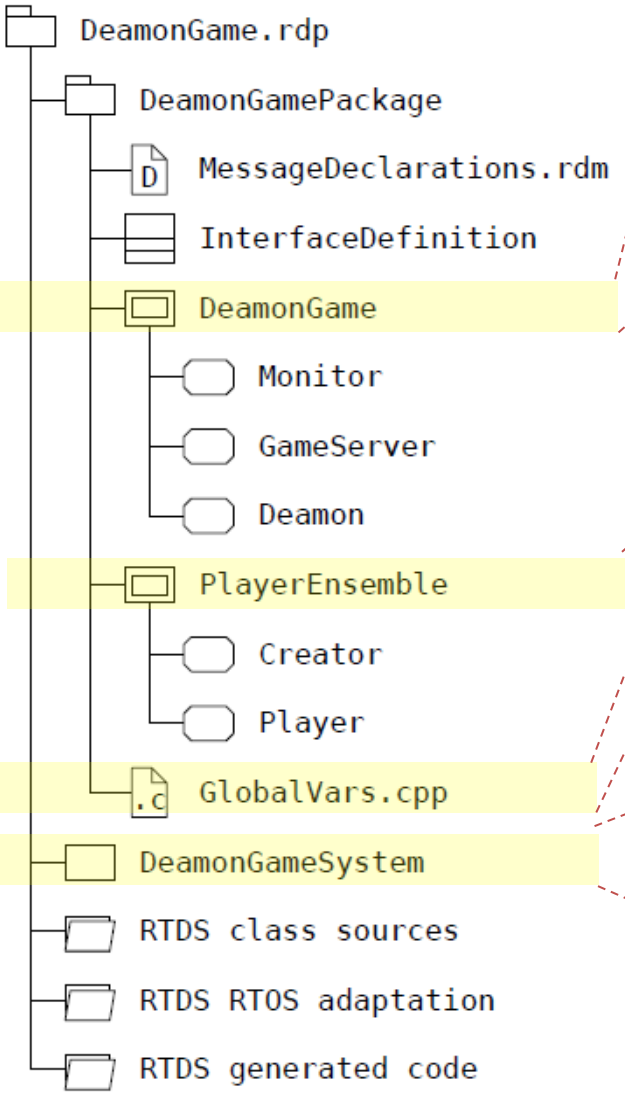




Blocktyp DeamonGame
Name = Dateiname

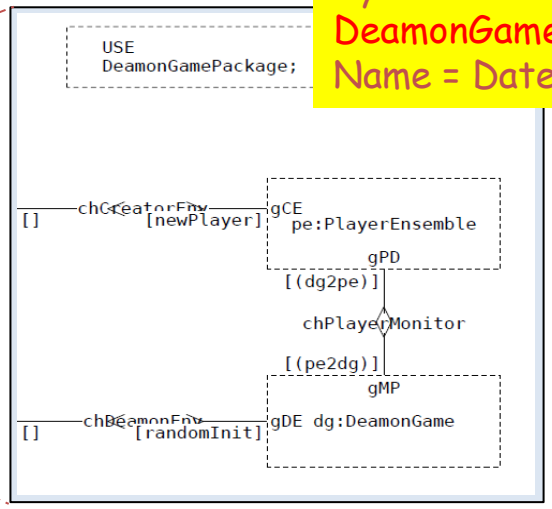


Blocktyp PlayerEnsemble
Name = Dateiname



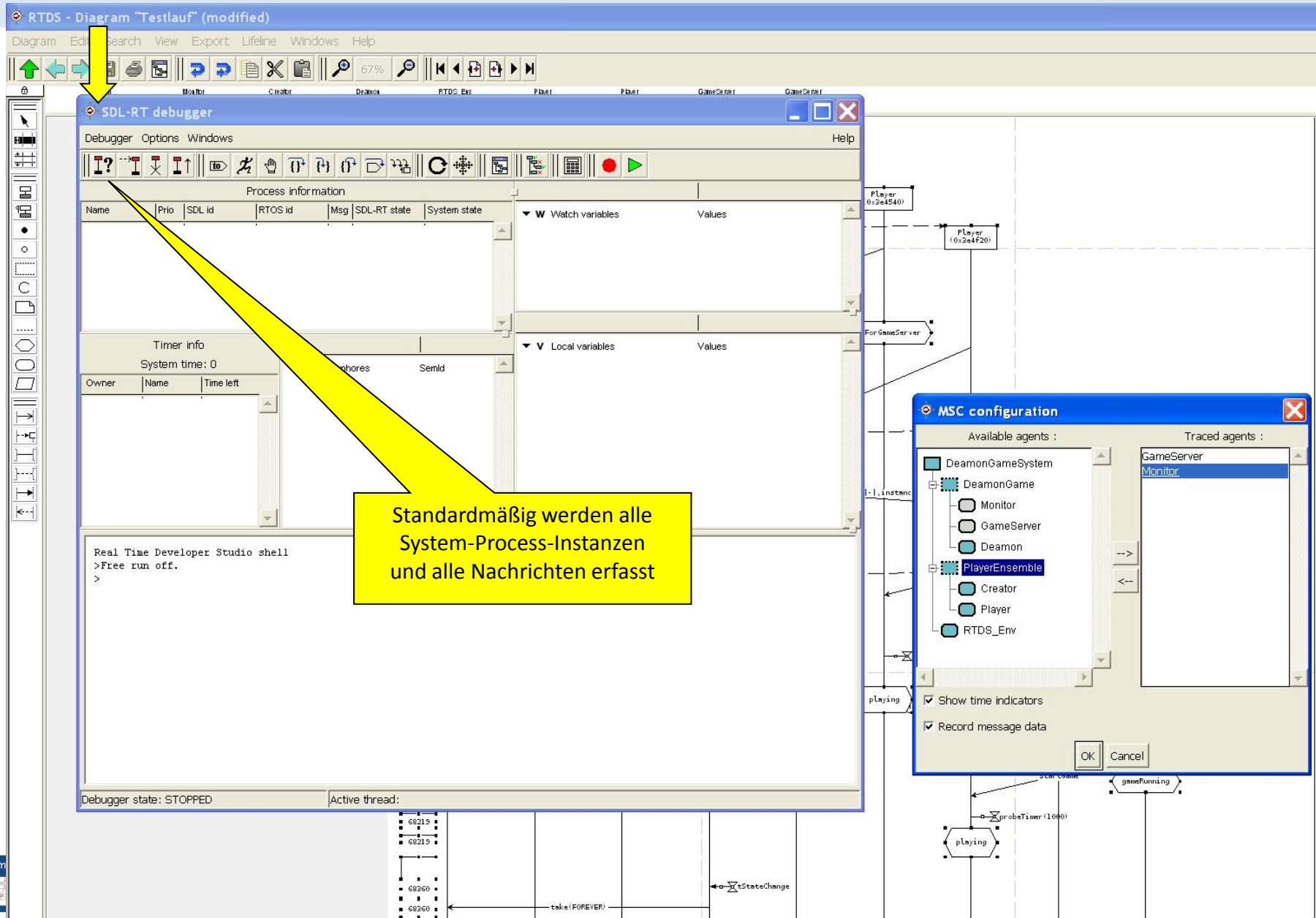

```
bool deamonIsEven = true;
```

C/C++ Erweiterung
globale Variable

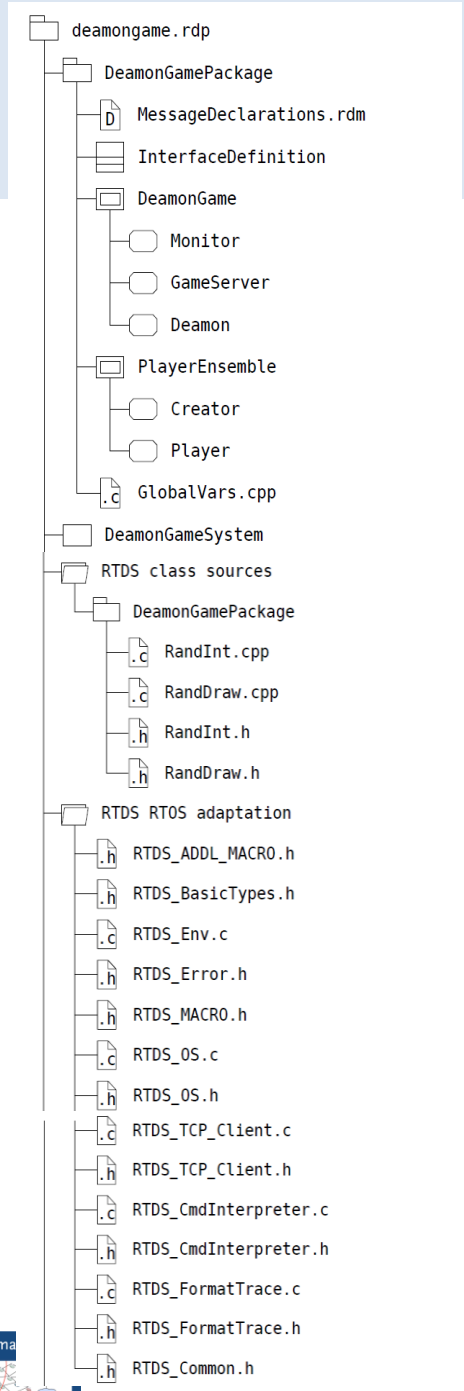


System
DeamonGameSystem
Name = Dateiname

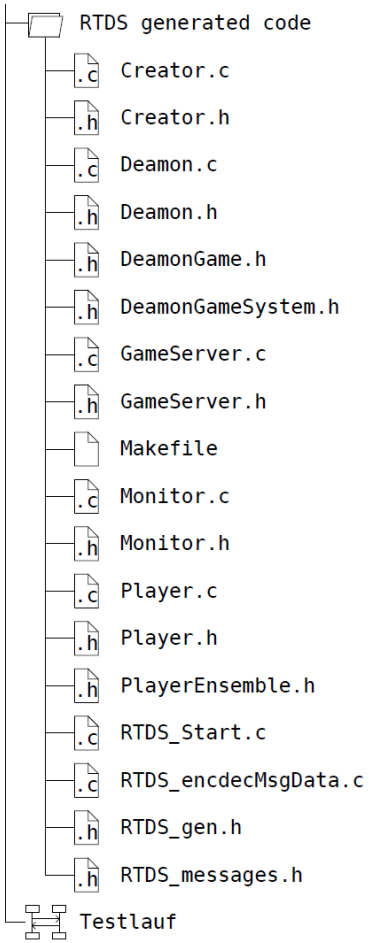
Konfiguration der SD-Ausgabe (MSC)



DeamonGame-Simulator



Quellen



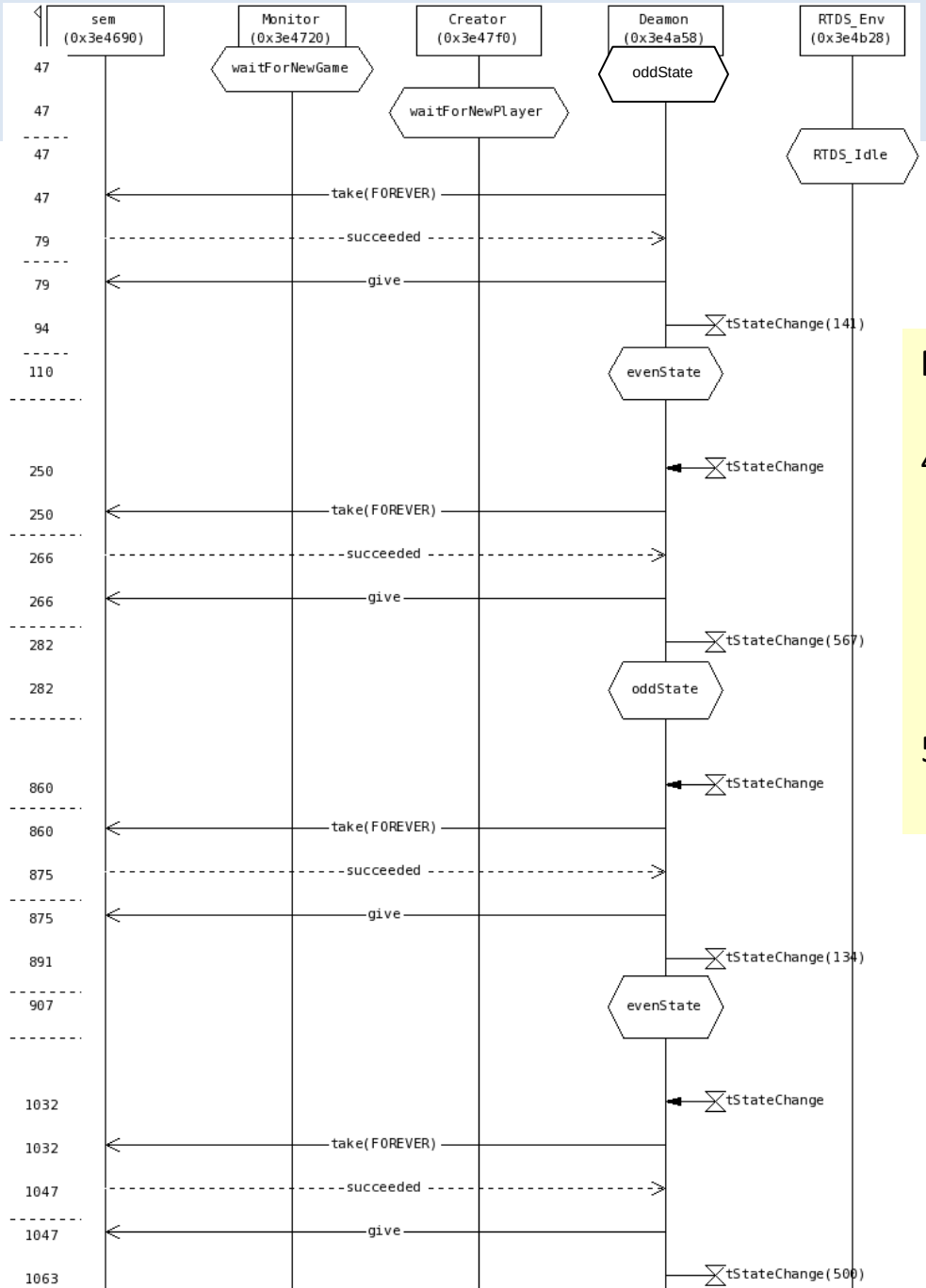
Run

1. Erzeugung des Simulators
   → neue Bedienoberfläche

2. Vorbereitung der Simulatorausgabe
   (SD-Ausgabe:Konfiguration)

3. Start des Simulators
   - alle statischen Prozessinstanzen werden erzeugt (Monitor,Creator,Deamon,Semaphore)
   - arbeiten Starttransitionen ab
   - verharren in Zuständen, wo sie auf Eingabenachrichten warten (bis auf Deamon)

4. Unterbrechung mit Pause



Noch keine Player- und GameServer-Instanzen

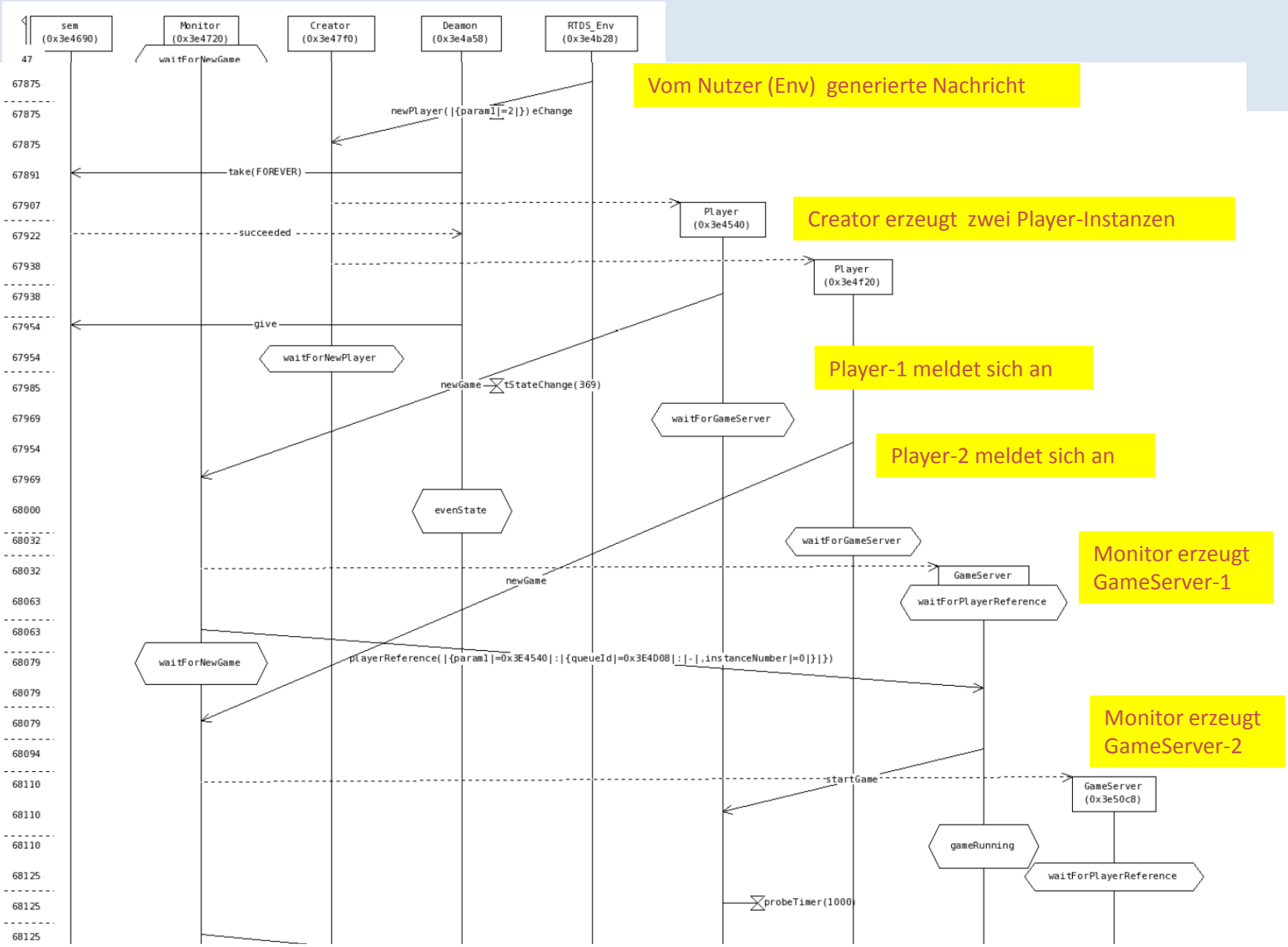
Monitor und Creator verharren

Deamon wechselt stochastisch seinen Grundzustand

Run (Fortsetzung)

4. Erzeugung einer Nachricht newPlayer(2) mit Creator als Empfänger-Prozessinstanz
   → 2 Spieler nehmen damit am Spiel teil

5. Unterbrechung mit Pause


sem (0x3e4690)
Monitor (0x3e4720)
Creator (0x3e47f0)
Deamon (0x3e4a58)
RTDS_Env (0x3e4b28)
47
waitForNewGame
67875
67875
67875
67891
67907
67922
67938
67938
67954
67954
67985
67969
67954
67969
68000
68032
68032
68063
68063
68079
68079
68079
68094
68110
68110
68110
68125
68125
68125
newPlayer(|{param1|=2|})eChange
take(FOREVER)
succeeded
give
waitForNewPlayer
newGame
tStateChange(369)
Player (0x3e4540)
Player (0x3e4f20)
waitForGameServer
evenState
waitForGameServer
newGame
GameServer
waitForPlayerReference
waitForNewGame
playerReference(|{param1|=0x3E4540|:|{queueId|=0x3E4D08|:|-|,instanceNumber|=0|}|})
startGame
GameServer (0x3e50c8)
gameRunning
waitForPlayerReference
probeTimer(1000)
Vom Nutzer (Env) generierte Nachricht
Creator erzeugt zwei Player-Instanzen
Player-1 meldet sich an
Player-2 meldet sich an
Monitor erzeugt GameServer-1
Monitor erzeugt GameServer-2

7. SDL-Konzepte (Präzisierung)

1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Ersetzungsmodell: Priorisierter Input
4. Nachrichtenadressierung
5. Dynamische Prozessgenerierung
6. Prozeduren
7. Lokale Objekte
8. Semaphore
9. Spezialisierung von Zustandsautomaten

Zustand und Zustandsübergang

Ann.: betrachten ein System zu einem beliebigen Zeitpunkt

- jede (existierende) Prozessinstanz
  - verharrt entweder in einem ihrer Grundzustände
    und wartet dabei auf einen Zustandsübergangsauslöser
    (z.B. auf die Ankunft eines bestimmten Signals)

  oder
  - führt einen Zustandsübergang aus (nicht unterbrechbar)

- im Zustandsgraphen eines Prozesses sind i. Allg.
  - pro Grundzustand alternative Varianten für die Auslösung eines Zustandsübergangs vorgesehen,
    wobei
  - die jeweils aktuelle Nachricht("älteste" im Puffer) in der Regel entscheidet, ob und welche der möglichen Alternativen zur Ausführung
    (d.h. auch ein weiteres Verharren im Zustand ist möglich)

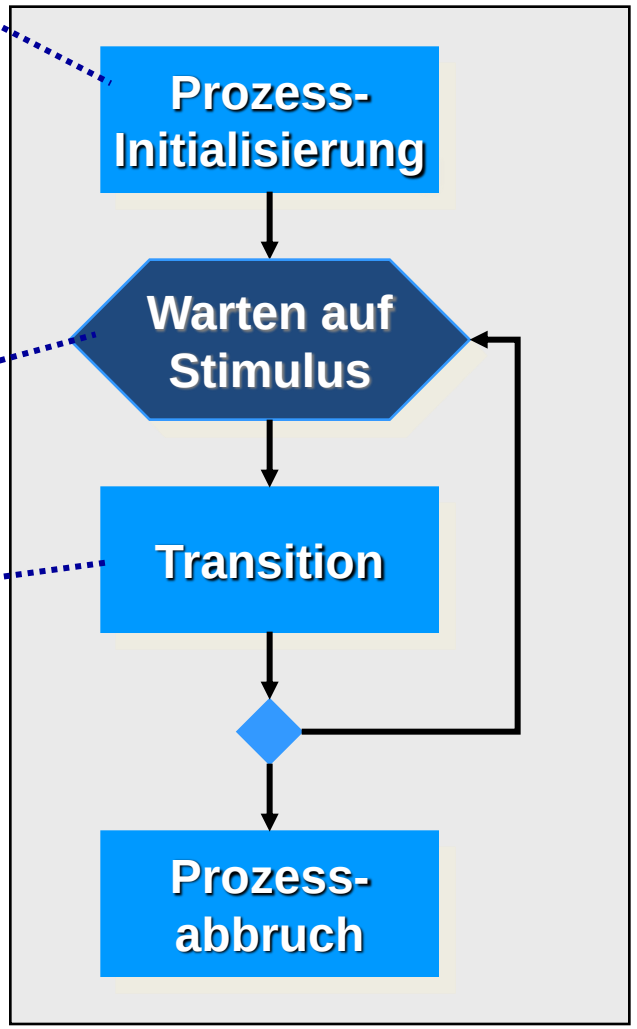
- die Auslösung eines Zustandsübergangs hat zur Folge:
  - die Konsumtion der Auslöser-Nachricht bei optionaler Übernahme der Parameter in lokale Variablen
  - Ausführung sequentieller Aktionen (Variablenänderungen, Nachrichtenausgaben, Prozessgenerierungen, Remote-Prozeduren-Rufe, Stop…)
  - Annahme eines neuen oder des gleichen Zustandes (vollzogener Zustandsübergang)

Prozess-Lebenszyklus (Schema)

mit Existenzbeginn
- Initialisierung lokaler Variablen
- Ausgabe von Nachrichten
- Aufruf von Prozeduren
- Erzeugung von Prozess-Instanzen
- Übergang in einen echten Zustand

Auswahl erfolgt nach
- Stimulus und
- Zustand

- Wertänderung lokaler Variablen,
- Ausgabe von Nachrichten,
- Aufruf von Prozeduren,
- Erzeugung von Prozess-Instanzen
- etc.



Vorschlag einer Namenskonvention

… für erstes Zeichen von Namen
in Abhängigkeit des Modellelementetyps:

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

Struktur / Architetur

- System - Environment
  (Blocksymbol wird als System benutzt)

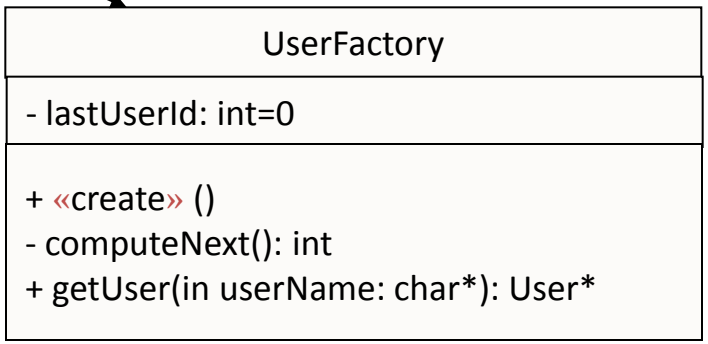
- Agent ist Element eines Systems
  - zwei Ausprägungen:
    - Block (ohne besondere Ausprägung im Zielcode),
    - Process (OS-Prozesse/Thread)
  - Mix von Prozessen und Blöcken auf einer Ebene des SDL-Programms ist erlaubt

- Prozessinstanzmengen-Kardinalität: Default (1,)

Klassenbeschreibung

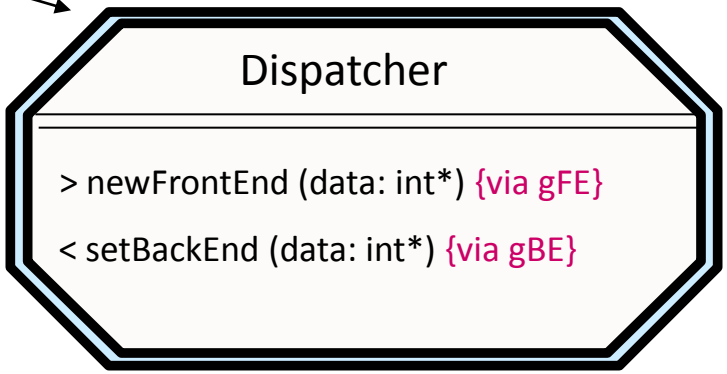
- 4 Formen
  - Interface-Klasse «interface»
  - System-Klasse «system»
  - Passive Klasse
  - Aktive Klasse : für Block-Typen, Process-Typen

In UML-Syntax



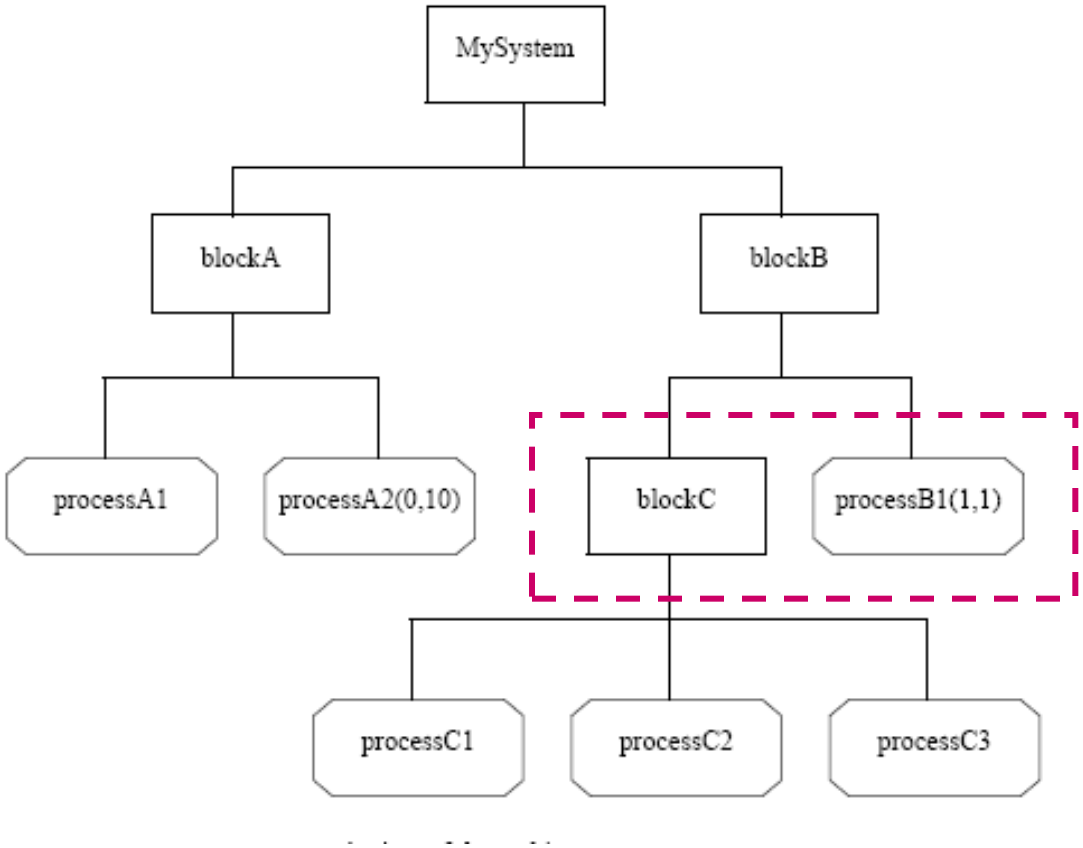
mehrere Konstruktoren (kein Return) möglich
nur ein Destruktor (<<delete>>)
In-, Out-, inOut-Parameter

SDL-angepasste-Syntax

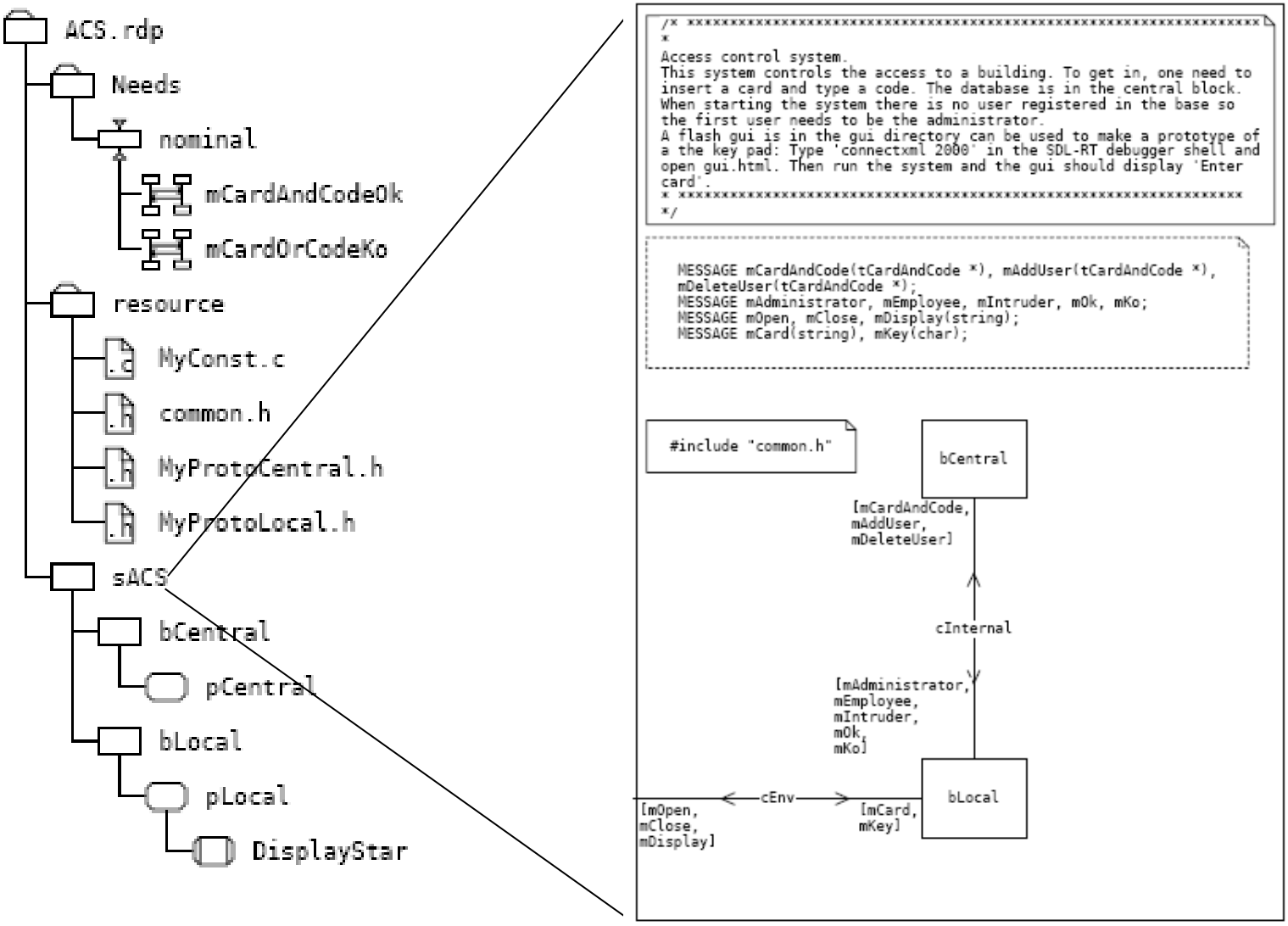


keine Attribute
Operations → PseudoNachrichten (nurEinParameter)
Property für Gate-Angabe

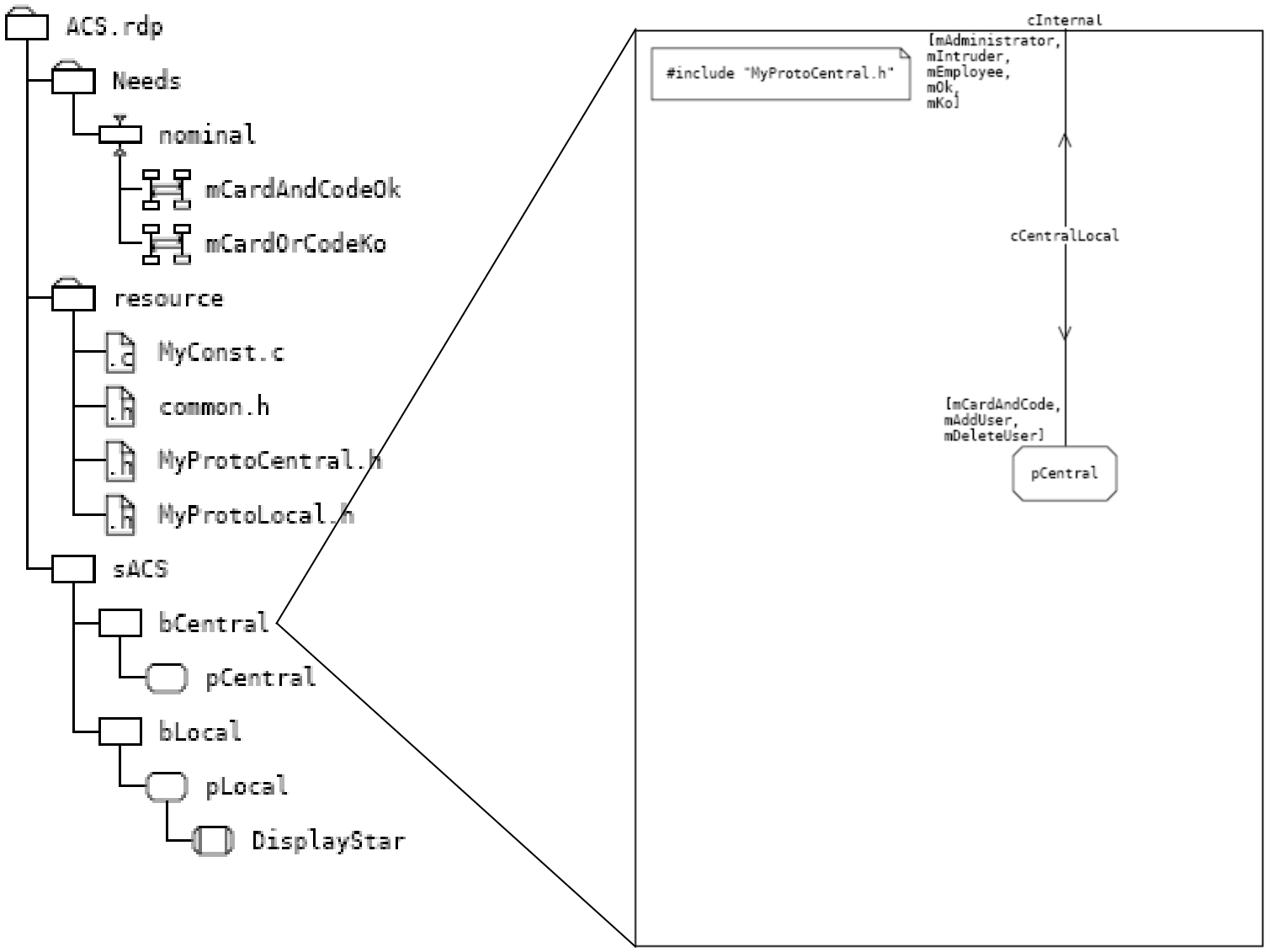
Beispiel-1: Systemarchitektur



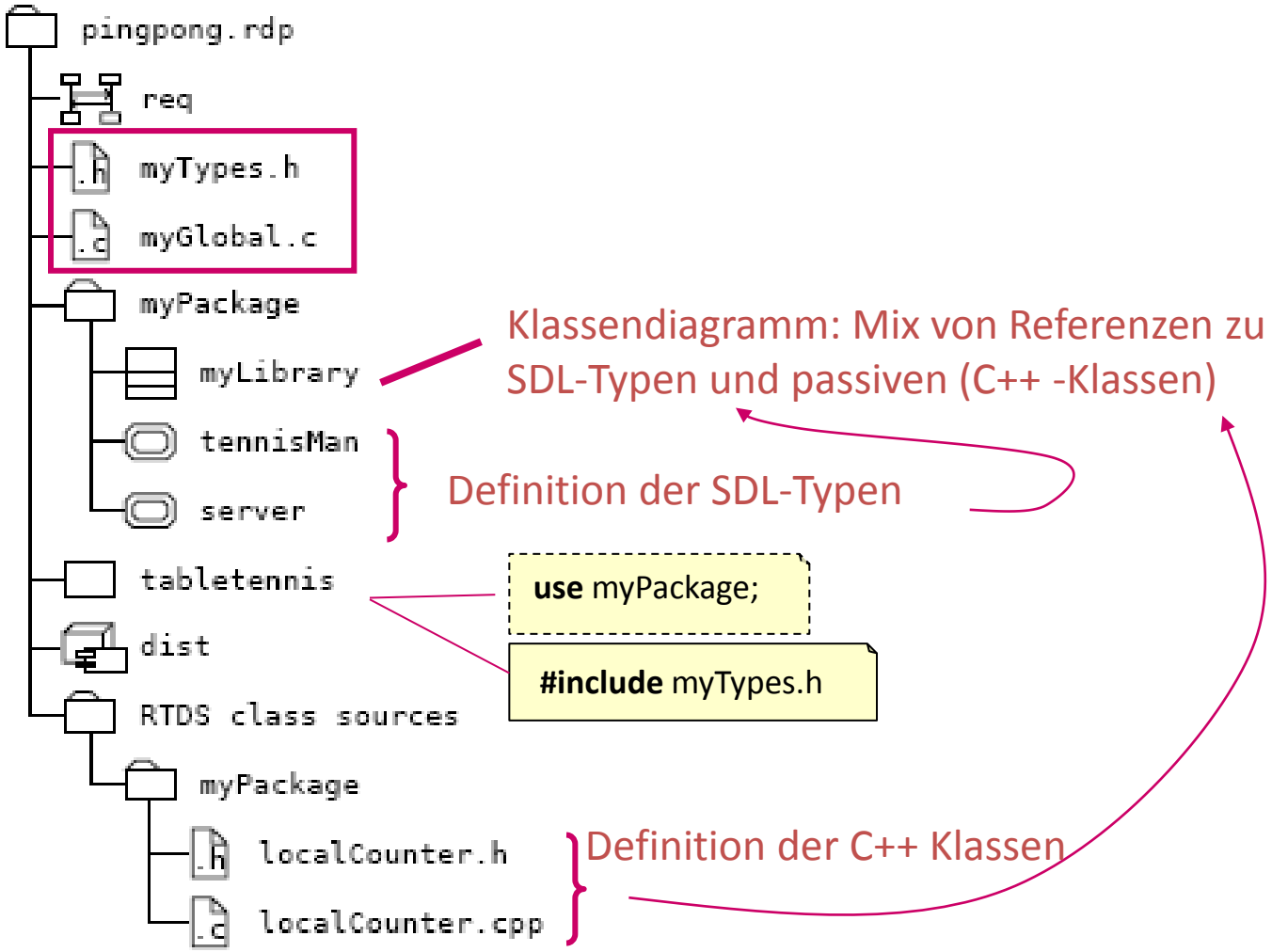
Beispiel-2: System- im SDL-RT-Projektbaum



Beispiel-2: System- im SDL-RT-Projektbaum

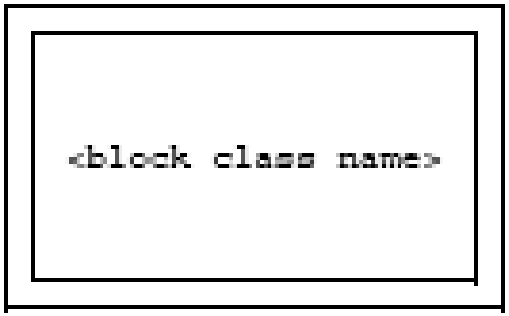


Beispiel-3: System im Projektbaum

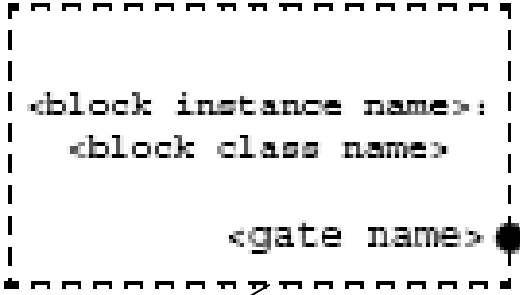


SDL-RT: Block-Typen (Block-Klassen)

Einschränkungen gegenüber Z.100



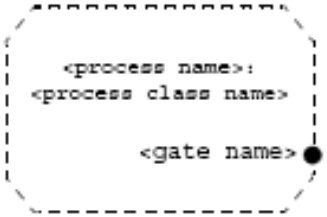
keine Vererbung
keine Virtualität
keine Kontextparameter



Instanzmengen sind nicht zulässig

Gates ohne Atleast-Typ-Angabe

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



Einschränkungen für Vererbung gegenüber Z.100
- Spezialisierung von Transitionen wie Z.100,
- aber keine Kontextparameter

mit erlaubter Redefinition von
- Transitionen (alle sind implizit virtuell)
- Gates (gewünschte Virtualität ist anzugeben)
- Datentypen

abstrakte Process-Typen, falls
- abstrakte Trigger
- abstrakte Gates

Präfix: ABSTRACT

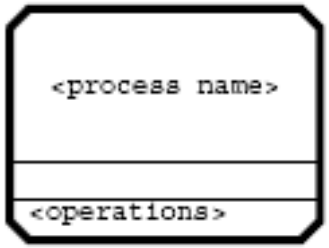
Präfix: VIRTUAL
Wirkung noch nicht überprüft

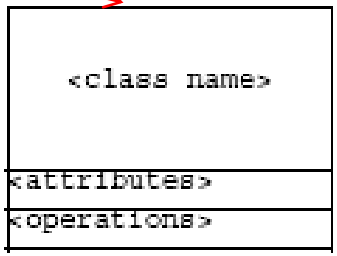
Präfix: VIRTUAL

UML-Klassendiagramme

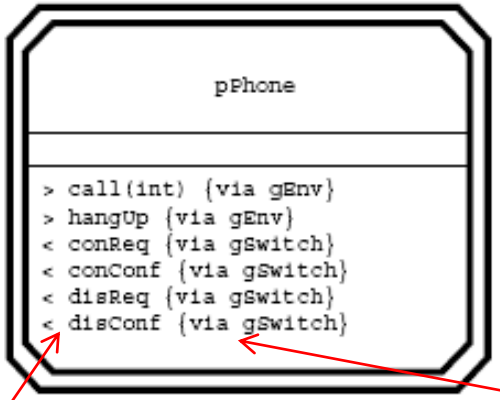
- aktive Klassen, passive Klassen

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





aktive Klassen haben keine Attribut-Compartments (Attribute werden woanders definiert ?)
Operationen sind ein- und ausgehende Nachrichten (asynchron) mit Gate-Angabe
leider: keine Angaben von Nachrichtenlisten möglich

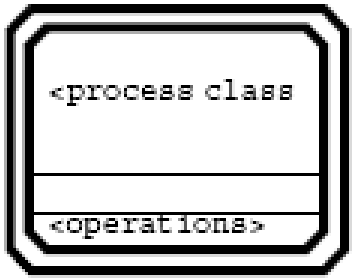


SDL-RT-Doko fehlerhaft !

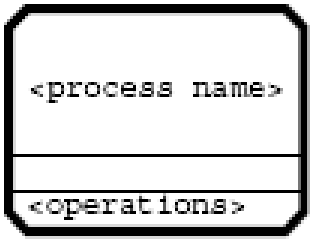
kein Leerzeichen

Doppelpunkt statt Leerzeichen

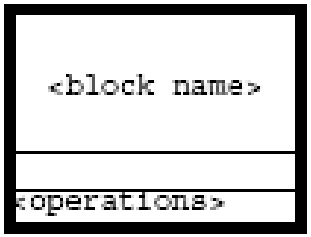
Vordefinierte graphische Symbole von stereotypisierter Klassen



Class stereotyped as a class of process

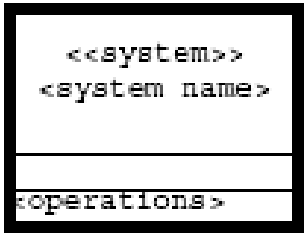


Class stereotyped as a process



Class stereotyped as a block

Class stereotyped as a class of block



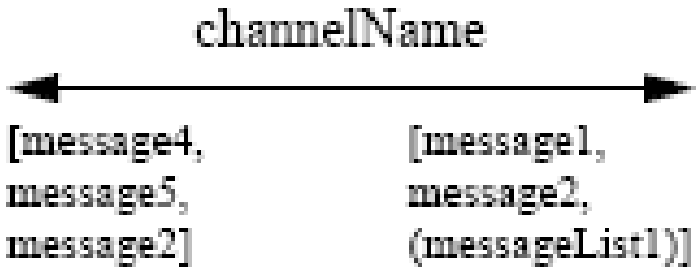
Class stereotyped as a system

Kommunikation

- Channel:= Zusammenfassung von Channel und Signalroute

Z.100: zeitbehaftet

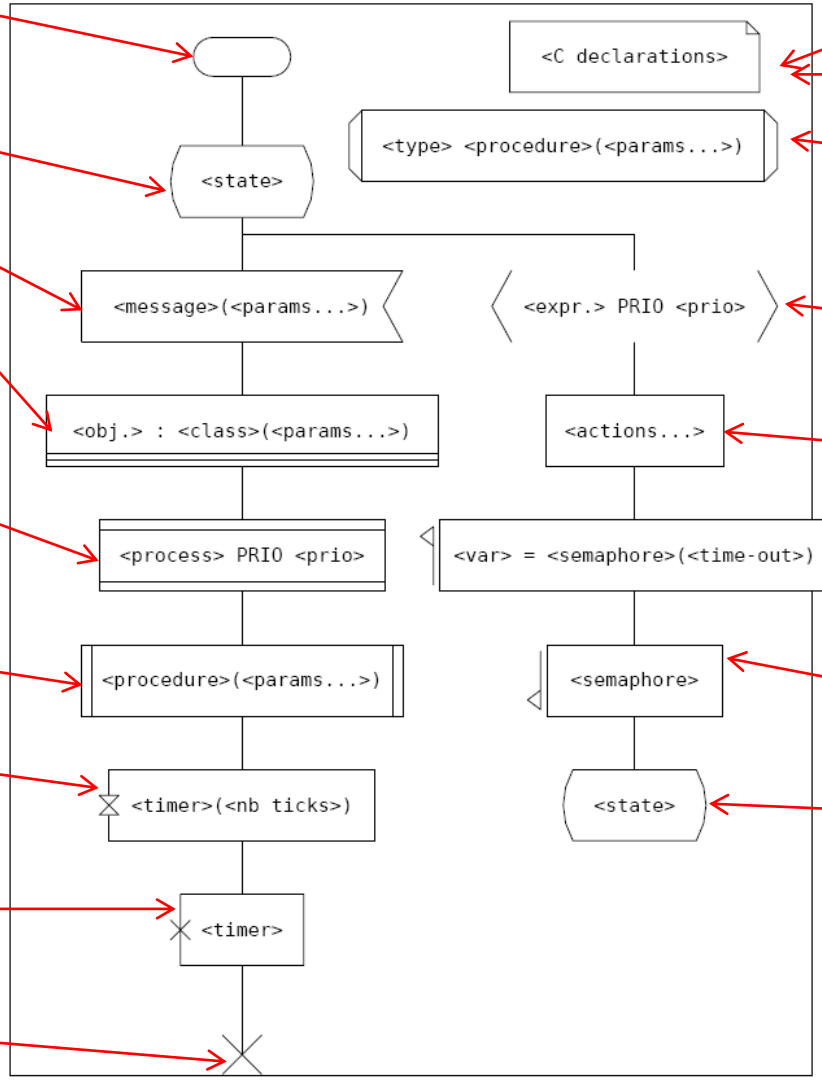
Z.100: zeitlos



Datentypen

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

Process-lokale Deklarationen



Startzustand

Zustand

Trigger: Signal

Objekterzeugung

Prozessinstanz-erzeugung

Prozedurruf

Timer-Start

Timer-Stop

Process-Stop

gestrichelt: SDL-Deklaration

voll: C-Deklaration

Prozedur-Deklaration

Trigger: Continuous-Signal

Task: Aktionssequenz

Semaphore-Ein mit Rückkehrcode

Semaphore-Aus

neuer Zustand

SDL-Signale und Signallisten

Änderung gegenüber Z.100:

Message statt Signal

- MESSAGE msg1, msg2(int),
  msg3(char*, struct MyType*, double)

  Parameter: gültige C-Typen (auch Zeiger)

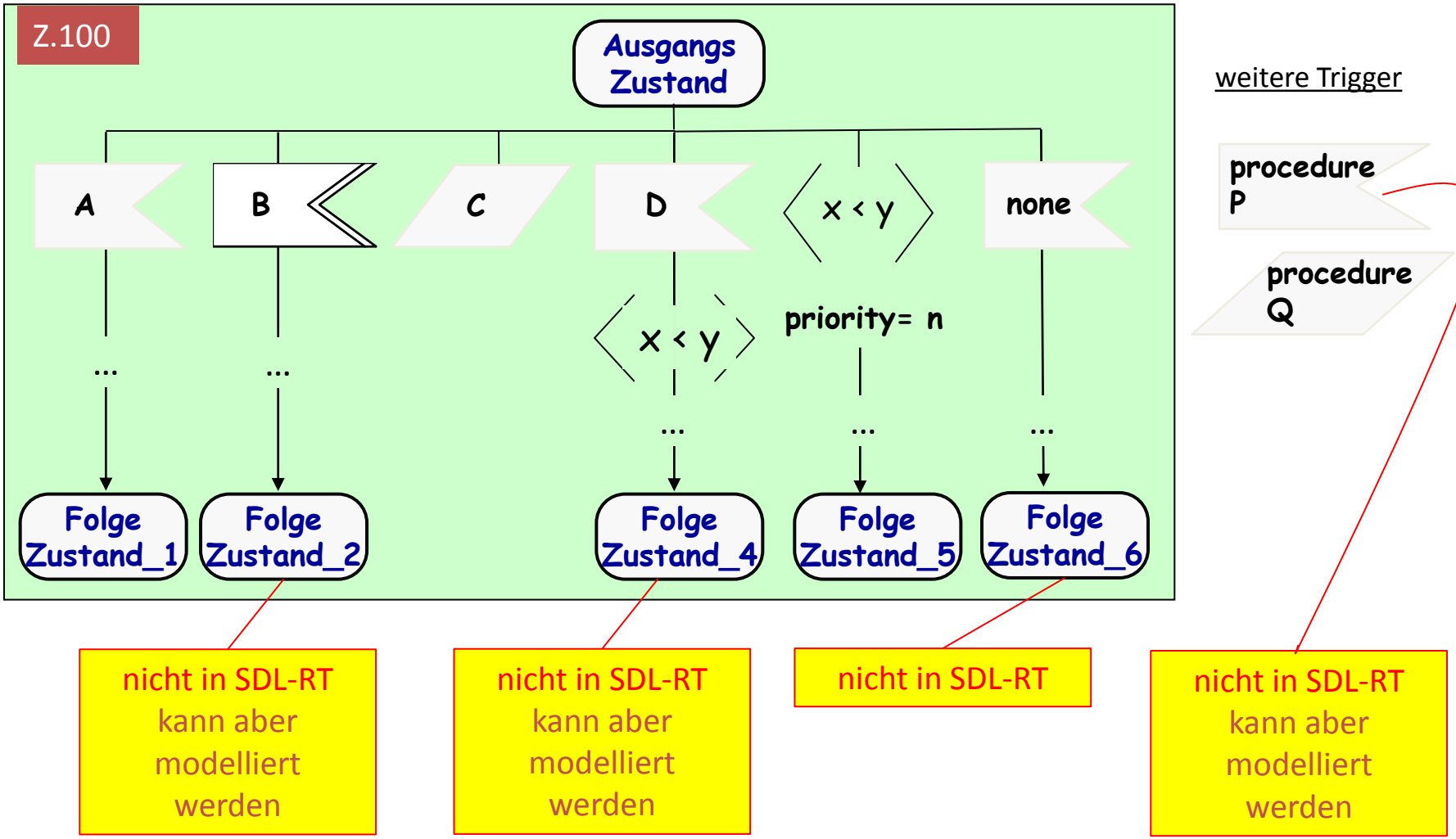
- MESSAGE_LIST myList= ((subList), msg1, msg3);

- implizite Timer-Deklaration

7. SDL-Konzepte (Präzisierung)

1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Ersetzungsmodell: Priorisierter Input
4. Nachrichtenadressierung
5. Dynamische Prozessgenerierung
6. Prozeduren
7. Lokale Objekte
8. Semaphore
9. Spezialisierung von Zustandsautomaten

Die Triggervarianten von SDL im Überblick

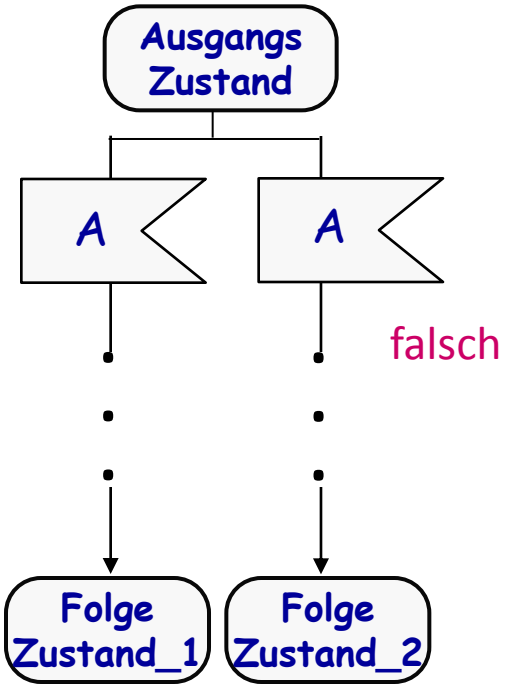


Normale Signaleingabe: Input ohne Parameter

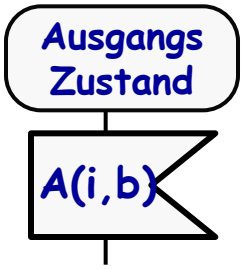

```
state Ausgangszustand;
input A;
```

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. im Puffer gestrichen)
- die Variable ***sender*** wird mit dem PId-Wert des Sender-Prozesses aktualisiert
- evtl. nutzerdefinierte Parameter von *A* werden ignoriert und gehen verloren



Normale Signaleingabe: mit Parameterübernahme

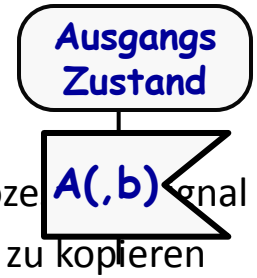


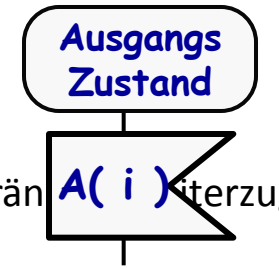

```
state Ausgangszustand;
input A (i, b);
```

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

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

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

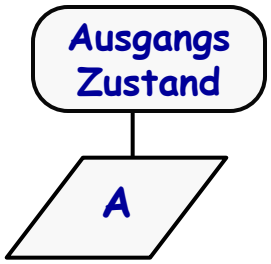




- hat ein Proze gnal unverän iterzugebe Parameter zu kopieren

Achtung:
eine partielle Übernahme der Parameter ist möglich

Signalzurückstellung: Save




```
state Ausgangszustand;
save A;
```

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

- das aktuelle Signal *A* wird zurückgestellt, das nachfolgende Signal wird damit zum neuen aktuellen Signal
- ist *A* das einzige Signal, verharrt 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

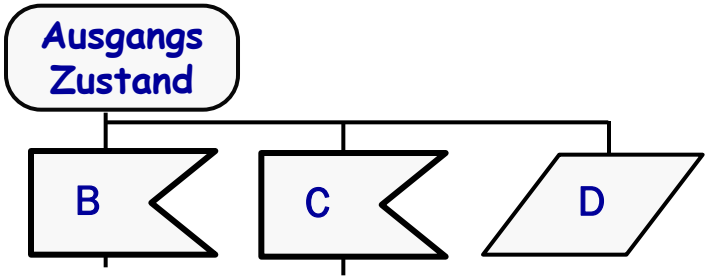
Save- Semantik

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

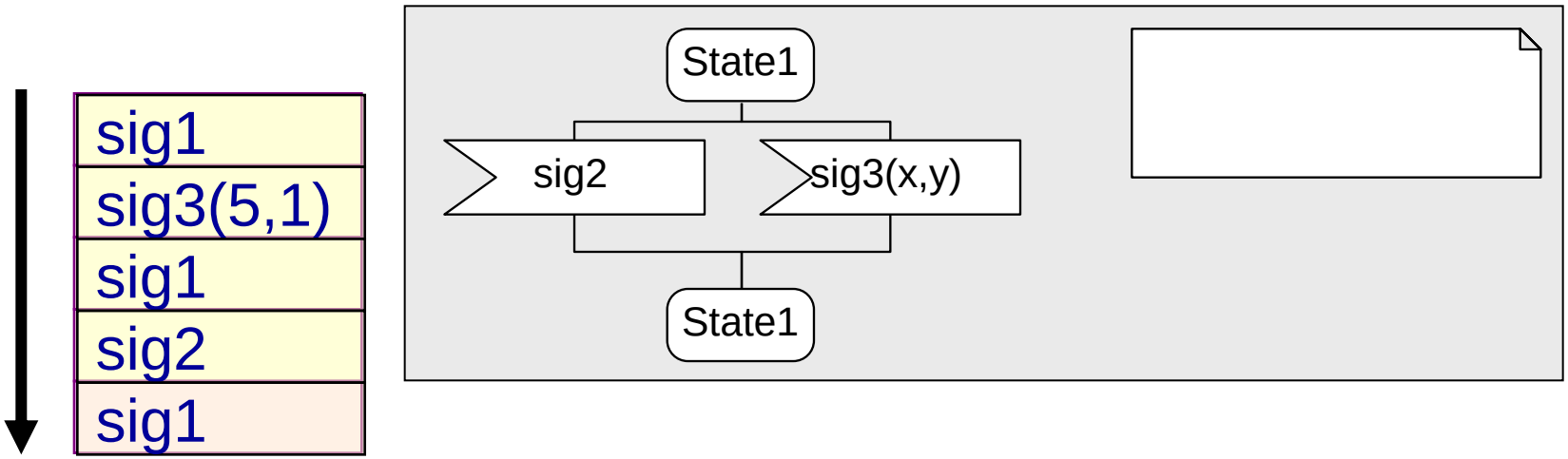



```
MESSAGE A (int, bool);
/* beliebige Signatur */
```

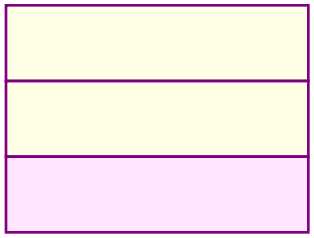
- Sollte für einen Zustand, indem sich der Prozess befindet, **kein Trigger** für das aktuelle Signal vorgesehen sein,
  wird es (ohne Kopie seiner expliziten und impliziten Parameter)
  verworfen

- o.B.d.A.: sei *A* dieses Signal für *Ausgangszustand*, wird dieses (und nur dieses) *A* verworfen

Trigger-Beispiel-1



nachher (im Zustand *State1*)

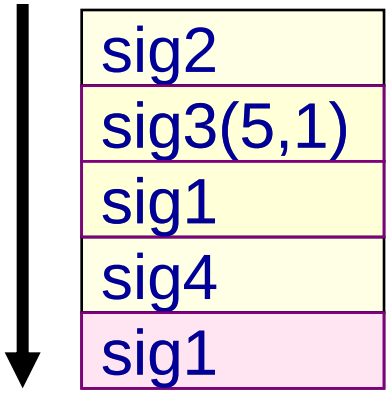


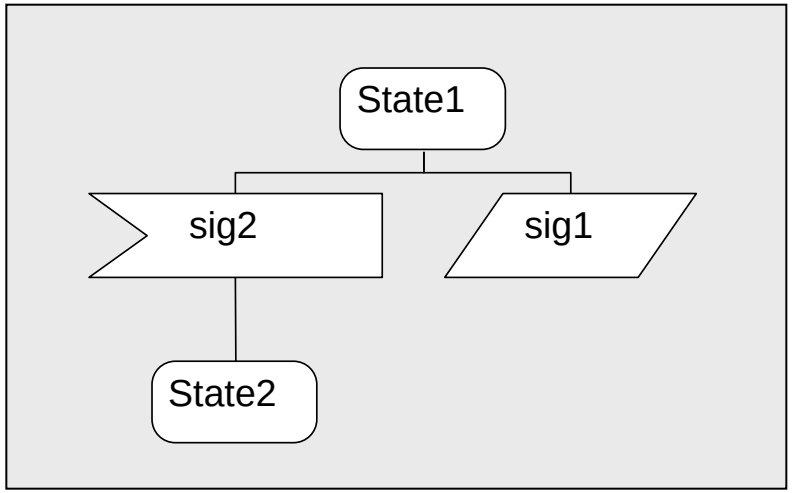
1. sig1 → discard
2. sig2 → input
3. sig1 → discard
4. sig3 → input mit Parameterübergabe
5. sig1 → discard

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

Trigger-Beispiel-2

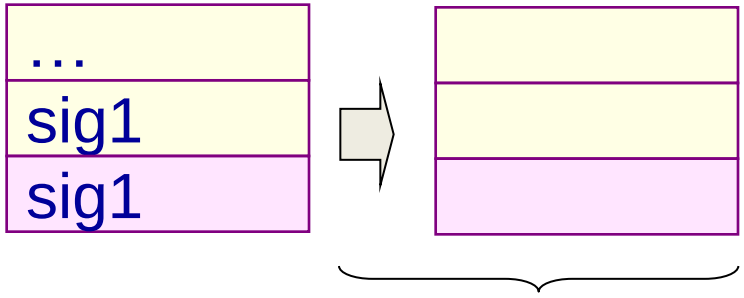
vorher





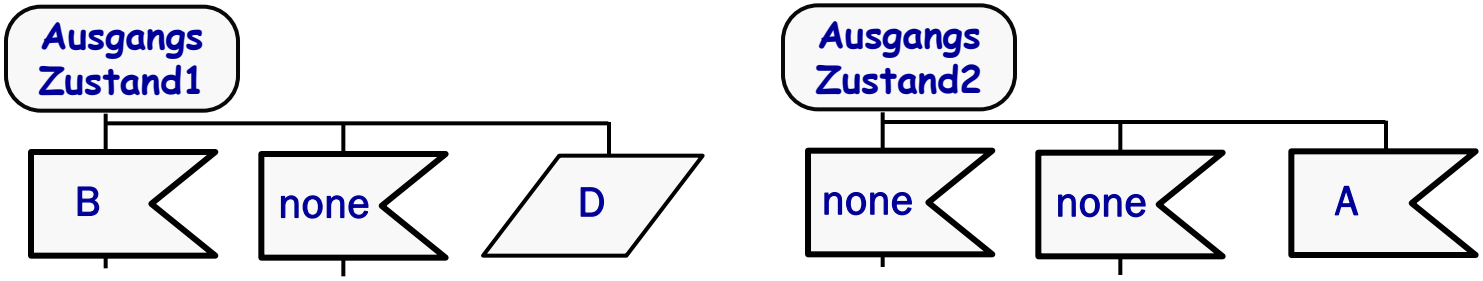
1. sig1 → save
2. sig4 → discard
3. sig1 → save
4. sig3 → discard
5. sig2 → input

nachher (im Zustand *State2*)



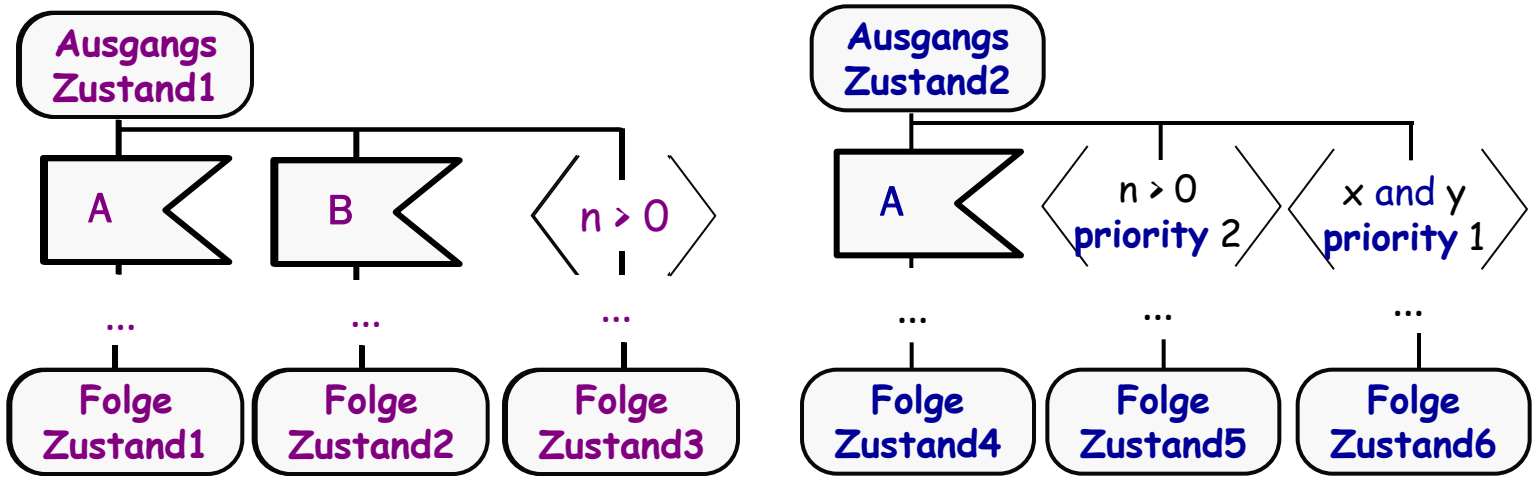
falls keine weiteren Übergänge in *State2*

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 (In Cinderella 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* and *y*) den Wert true liefert
- der Übergang wird **unabhängig** vom *n*-Wert ausgeführt

➔ niedrigster Prioritätswert bedeutet höchste Priorität

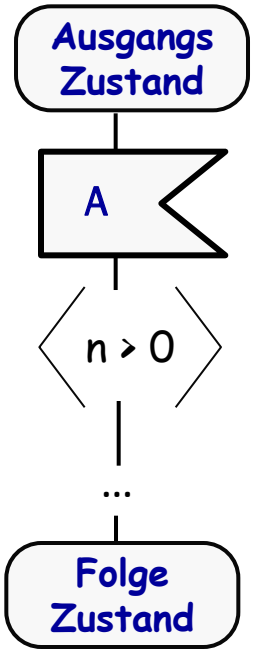
Trigger: Input mit Bedingung

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

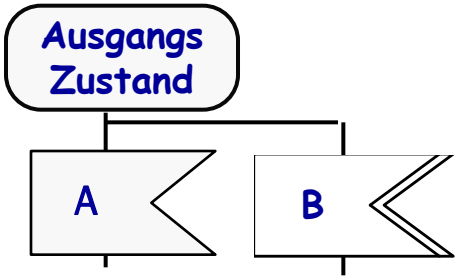
➔ **Lösung**: alternative Transitionen über

- 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

7. SDL-Konzepte (Präzisierung)

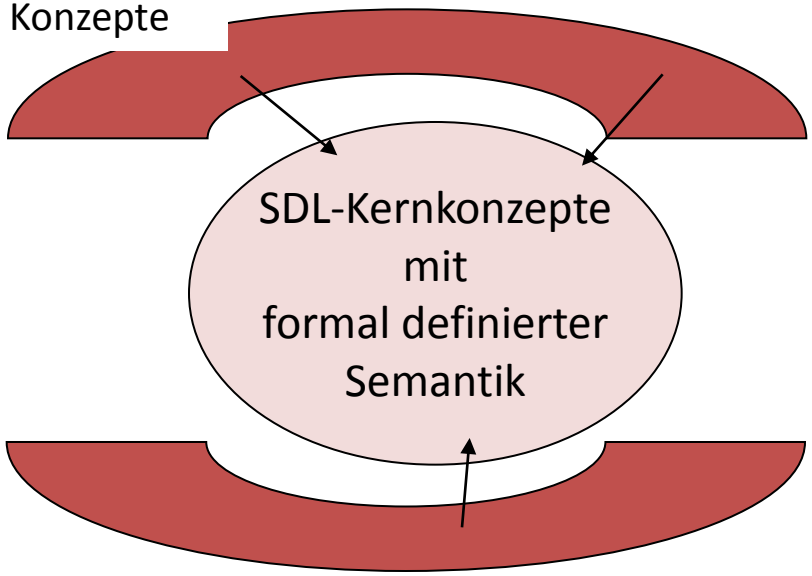
1. Modellstruktur
2. Einfacher Zustandsautomat: Triggerarten
3. Ersetzungsmodell: Priorisierter Input
4. Nachrichtenadressierung
5. Dynamische Prozessgenerierung
6. Prozeduren
7. Lokale Objekte
8. Semaphore
9. Spezialisierung von Zustandsautomaten

Semantik-Definition der Trigger

Beherrschung der semantischen Komplexität

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

spezielle Konzepte



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

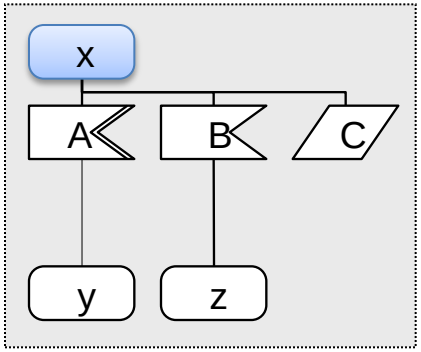
werden auf

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

zurückgeführt

Ersetzungsmodell für priorisierten Input (1)

Annahme (o.B.d.A.)



zu betrachtender Zustand x

Prinzip

– Aufteilung der Signal-Triggermenge (A, B, C) für den Zustand x:
  (1) für priorisierter Trigger: { A } und
  (2) den Rest: { B, C }

– Ersetzung des Zustandes x durch zwei (neue implizite) Zustände s1 und s2:

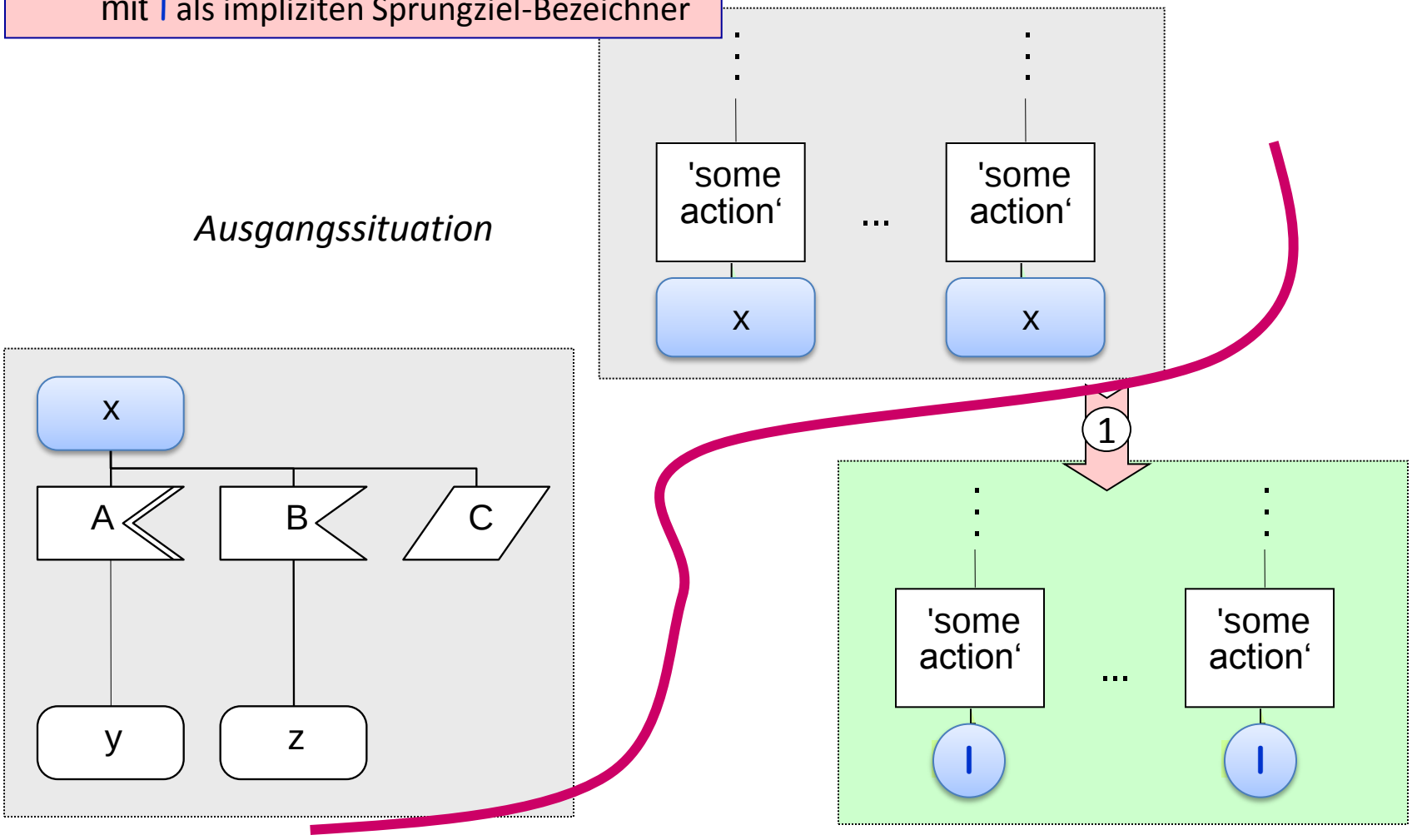
  s1: behandelt { A } als normale Inputs nach FCFS und stellt alle anderen ankommenden Signale mit SAVE zurück

  s2: behandelt den Rest { B,C } nach FCFS

in SDL/RT müssen priorisierte Trigger per Hand transformiert werden !

Ersetzungsmodell für priorisierten Input (2)

1. Schritt: jedes **nextstate** x; wird zu **join** l;
mit l als impliziten Sprungziel-Bezeichner



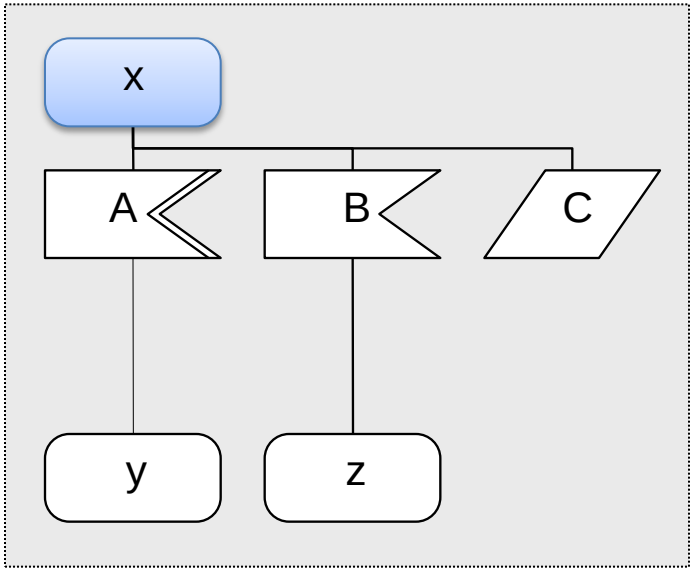
Ersetzungsmodell für priorisierten Trigger (3)

2. Schritt:
a) Einführung impliziter Variablen und Signale

2a

```
int n = 0;
int newn;
```

MESSAGE Xcont (**int**);



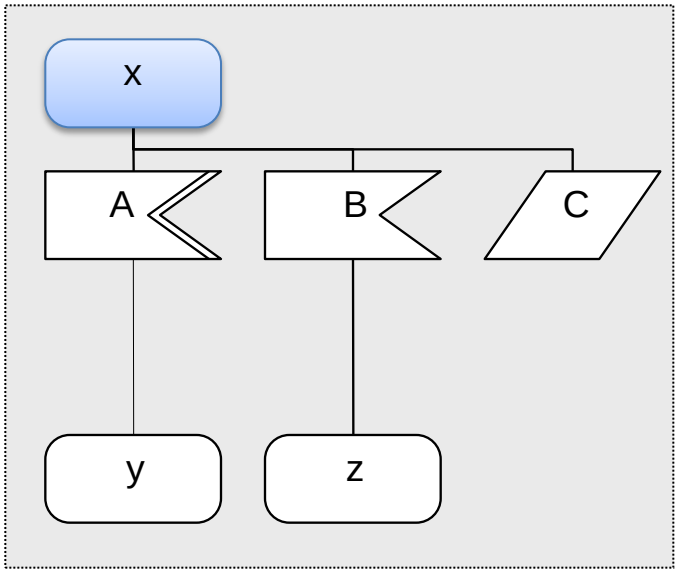
b) Einführung zweier impliziter Zustände

Behandlung der priorisierten Signale

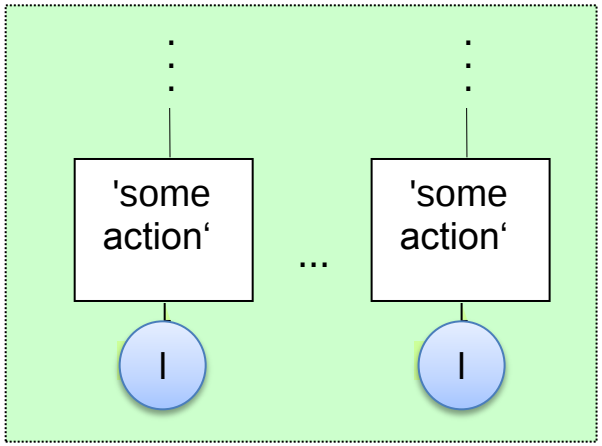
Behandlung der restlichen Signale

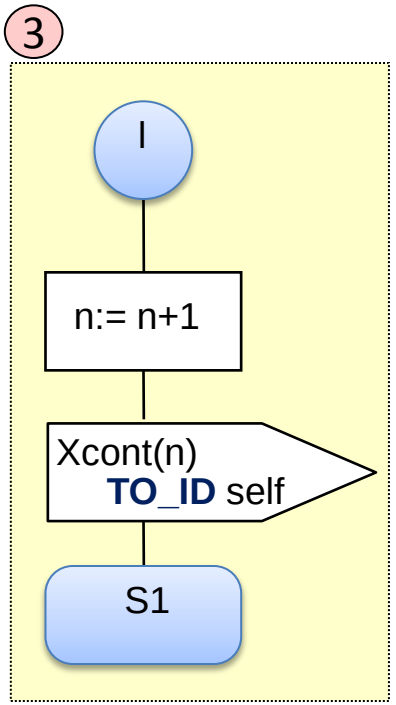
Ersetzungsmodell für priorisierten Input (4)

3.Schritt:
Übergang von der generierten Sprungmarke *I* zum impliziten Zustand *S1*



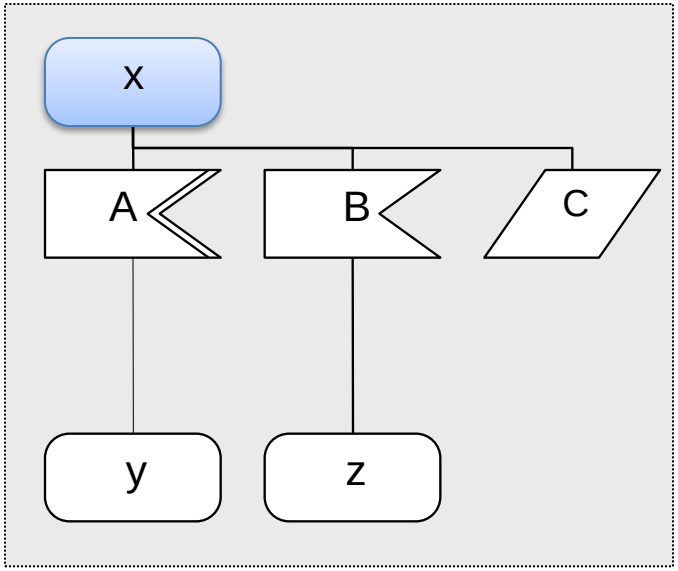

```
int n = 0;
int newn;
```

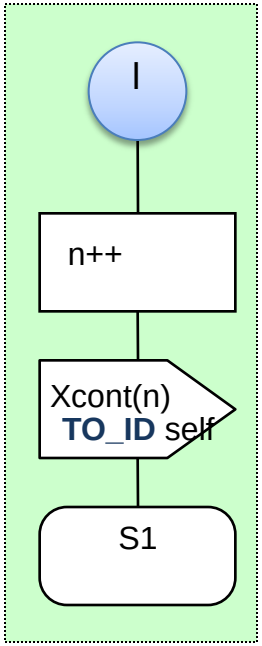


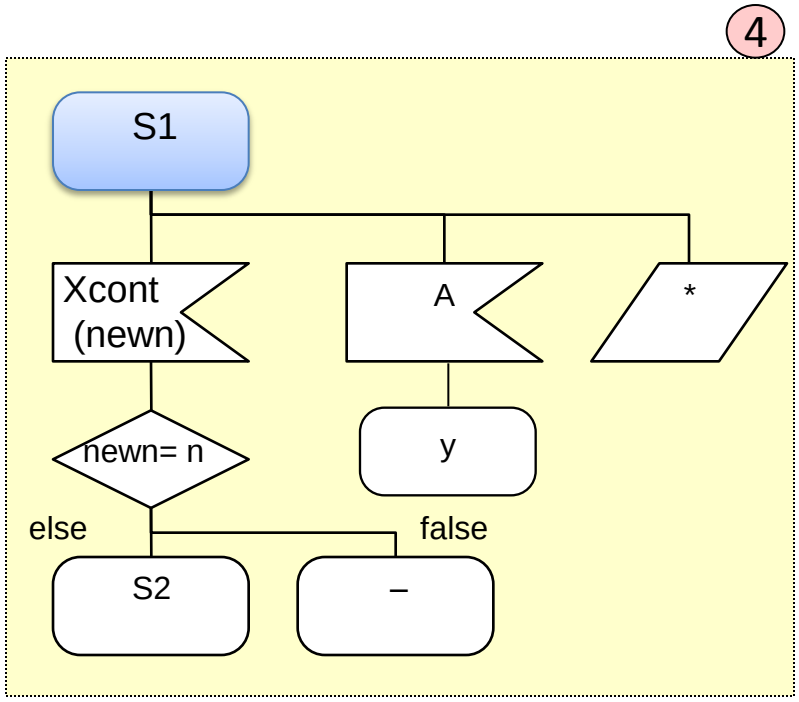


Ersetzungsmodell für priorisierten Input (5)

4.Schritt: Definition der Übergänge für *S1*



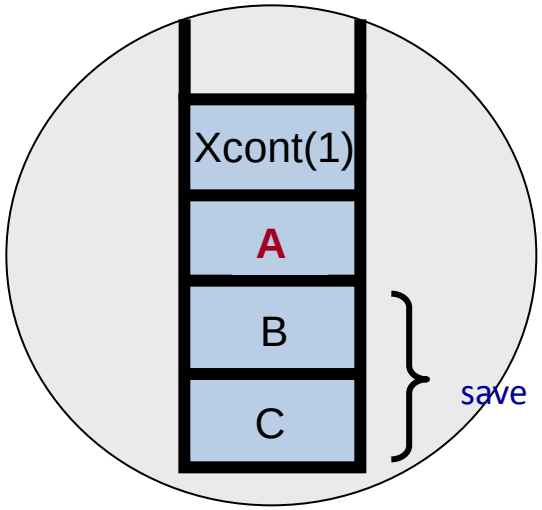


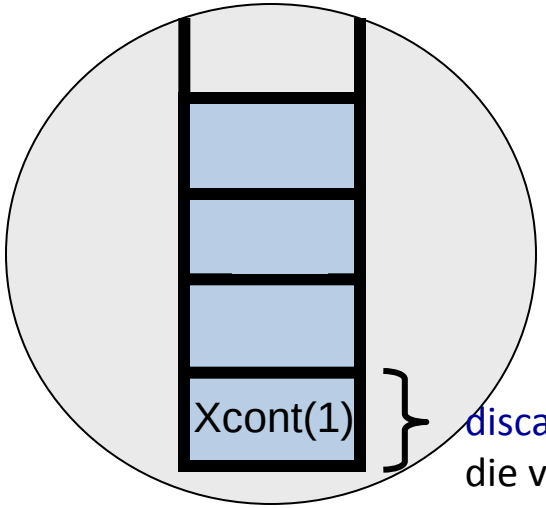



```
int n = 0;
int newn;
```

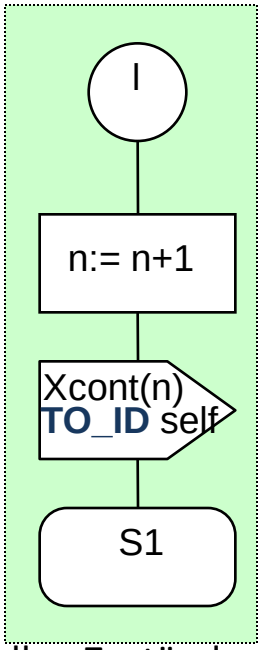
Zur Plausibilität des 4.Transformationsschrittes (1)

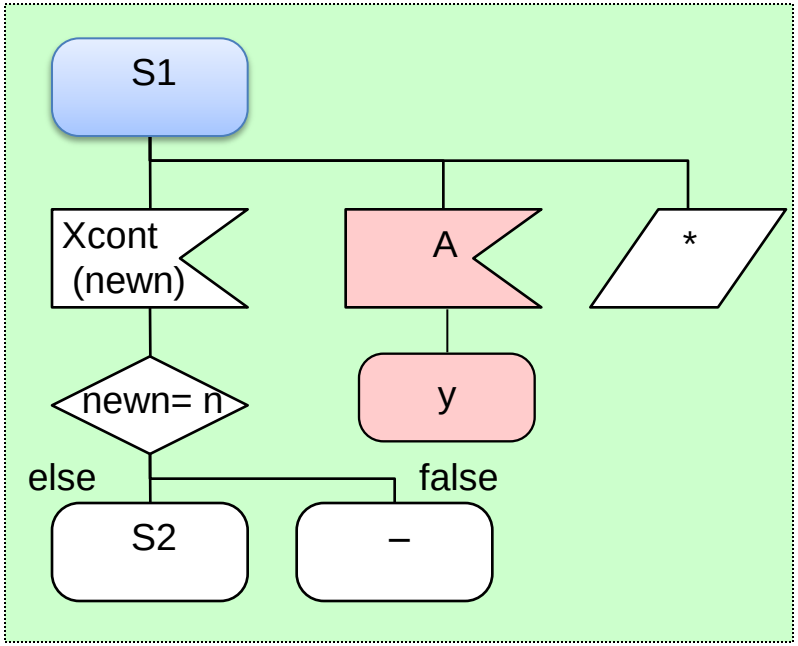
1.Szenario zur Bearbeitung der Nachrichten im Zustand S1: **A sei vorhanden**





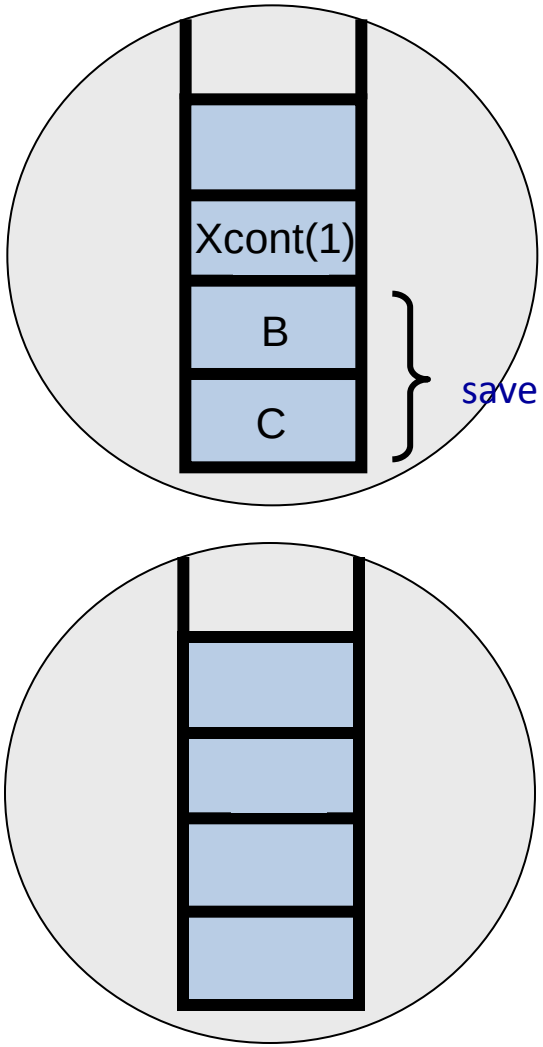
discard in allen Zuständen,
die verschieden von S1 sind





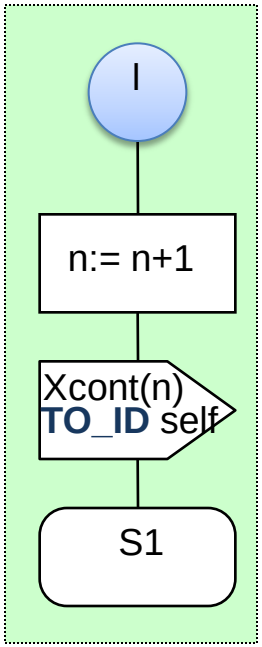
Zur Plausibilität des 4.Transformationsschrittes (2)

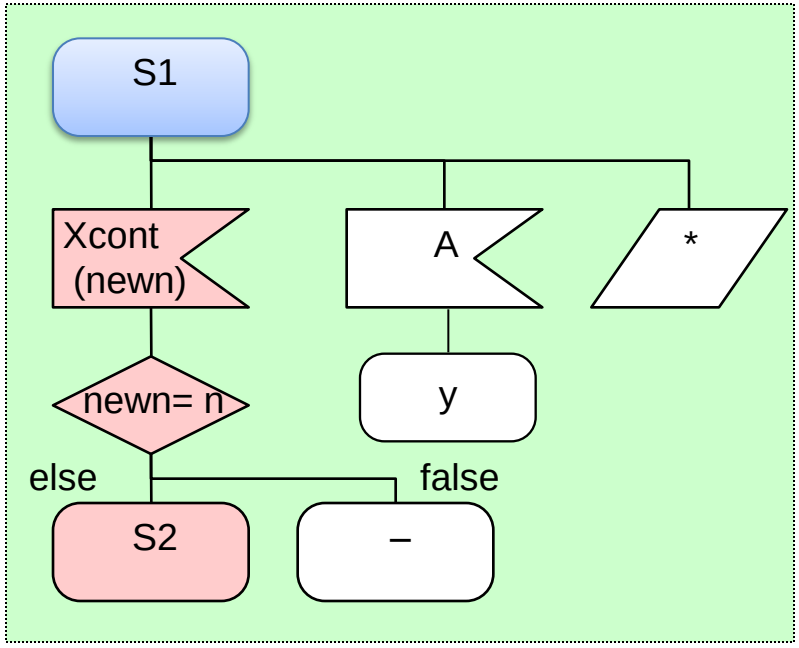
2.Szenario zur Bearbeitung der Signale im Zustand S1: **A sei nicht vorhanden**



int newn 1

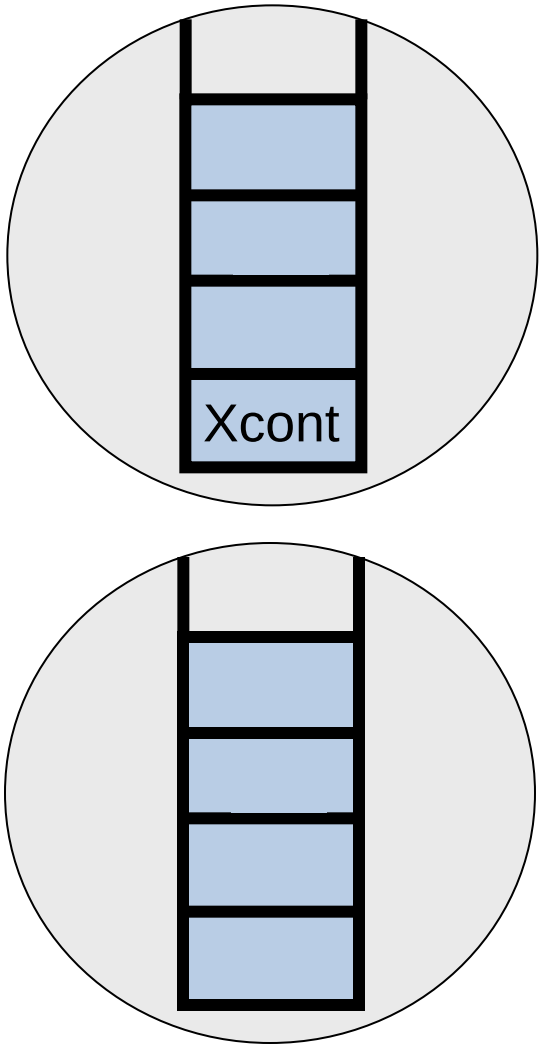
int n 1

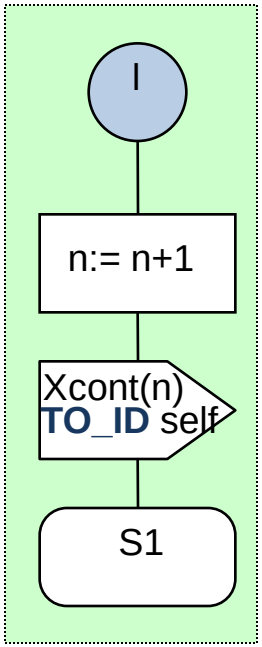


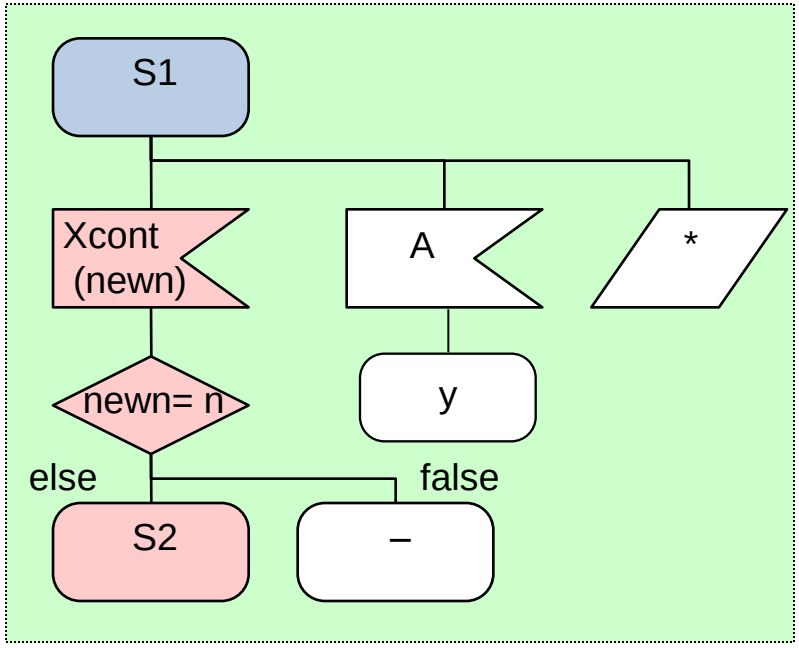


Zur Plausibilität des 4.Transformationsschrittes (3)

3.Szenario zur Bearbeitung der Signale im Zustand S1: **sei „kein“ Signal vorhanden**

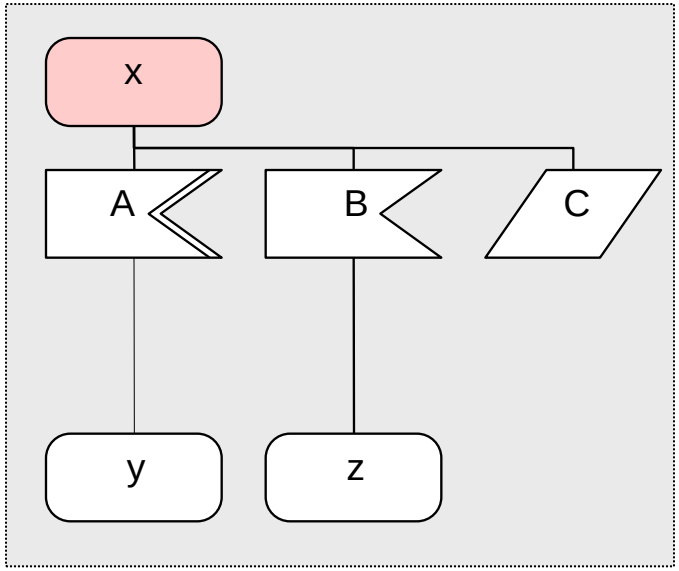


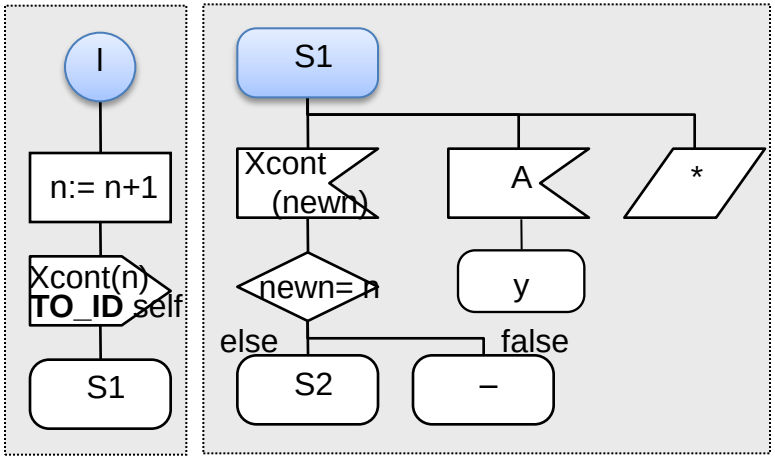


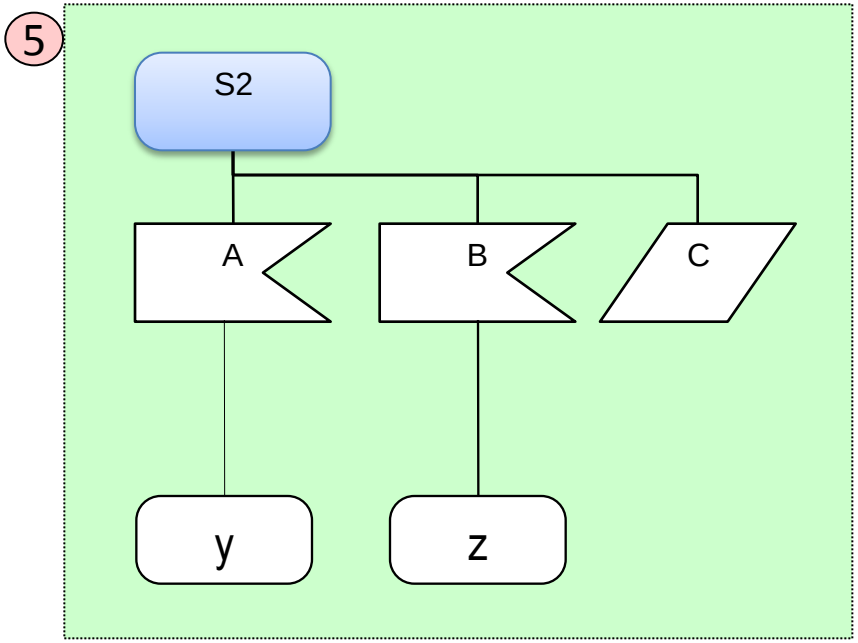


Ersetzungsmodell für priorisierten Input (9)

5. Schritt: Definition der Übergänge für S2

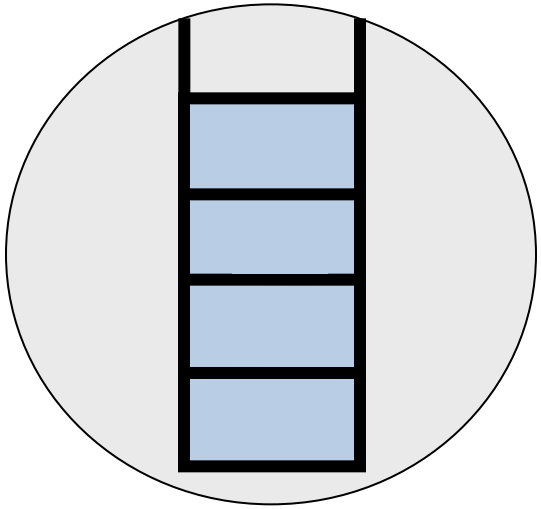


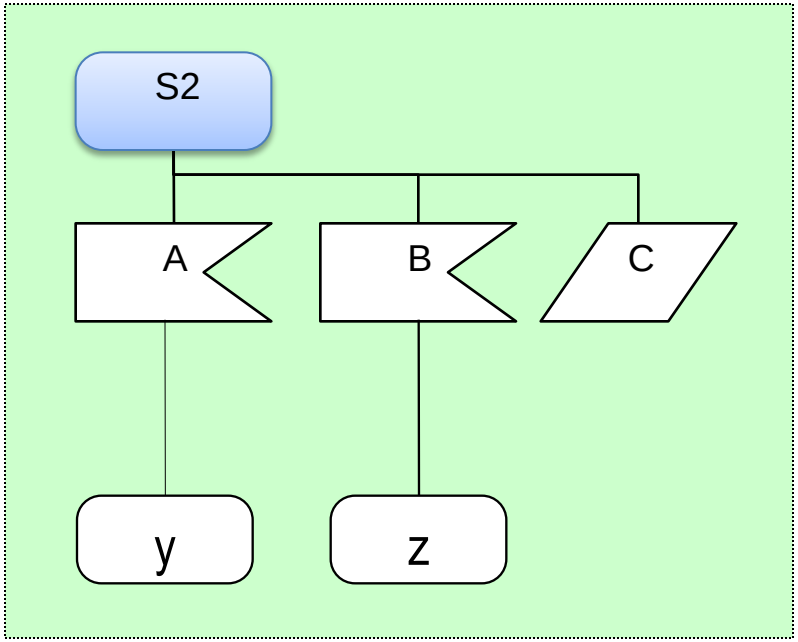




Zur Plausibilität des 5.Transformationsschrittes (1)

1.Szenario zur Bearbeitung der Signale im Zustand **S2: sei kein Signal vorhanden**



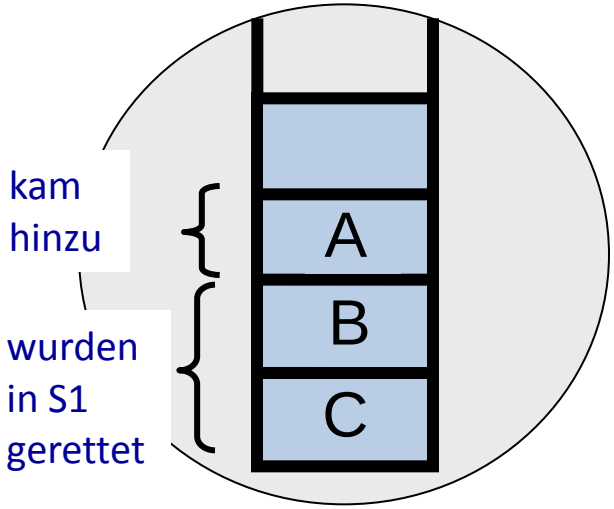


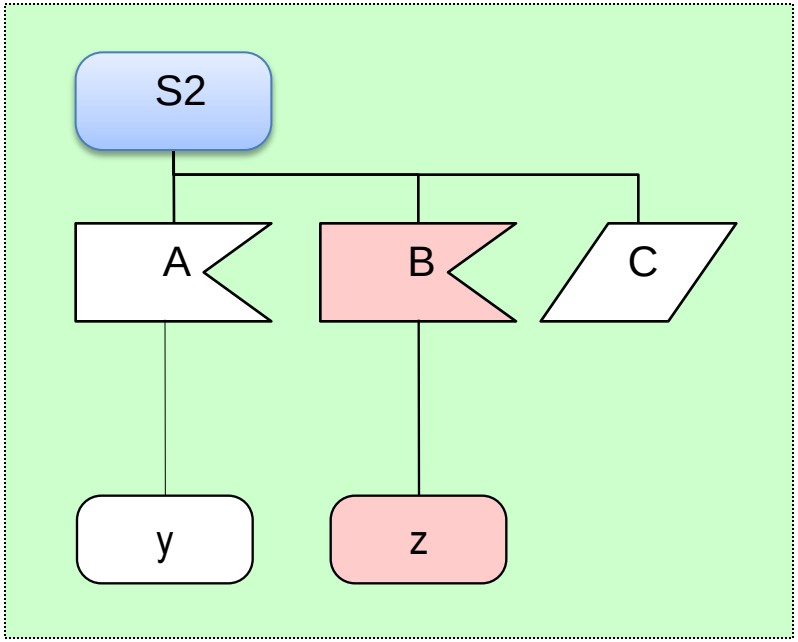
- Das erste ankommende Signal entscheidet die Fortsetzung

- Sollte nun zuerst A eintreffen, darf es nicht verloren gehen

Zur Plausibilität des 5.Transformationsschrittes (2)

2.Szenario zur Bearbeitung der Signale im Zustand **S2: seien Signale vorhanden**

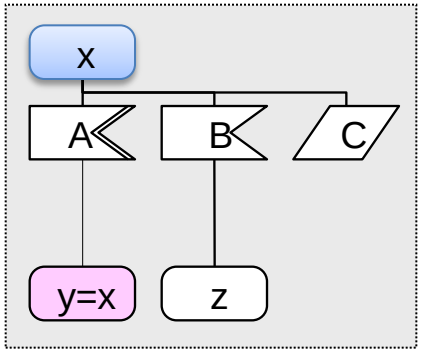




- Das erste ankommende Signal entscheidet die Fortsetzung

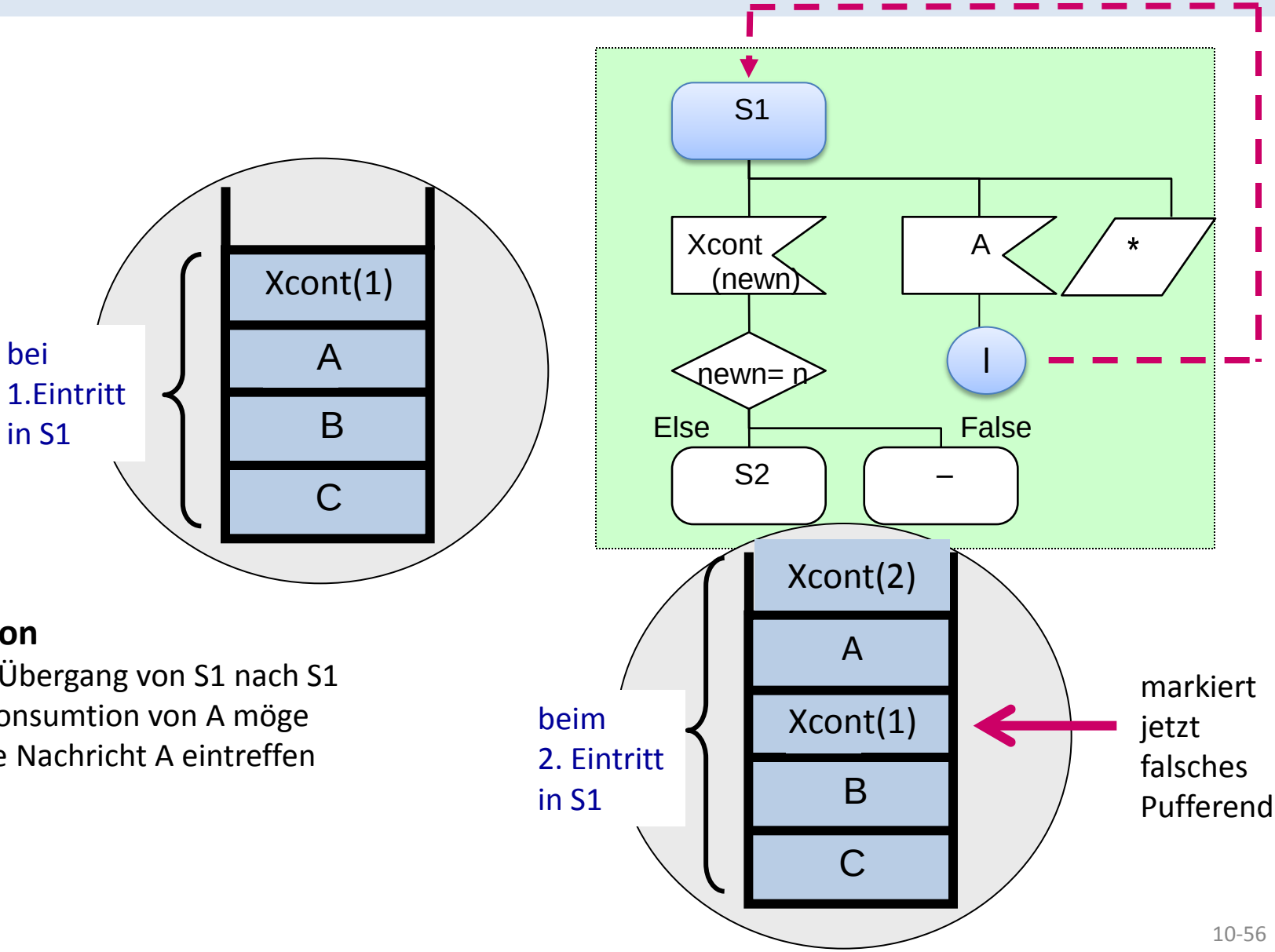
- A wird nicht berücksichtigt: zum Zeitpunkt des Eintritts in den Zustand x war nämlich A noch nicht vorhanden

Zur Plausibilität der Xcont-Parametrisierung



notwendig für Spezialfall:
der Folgezustand *y* für priorisierten A-Trigger ist wieder Ausgangszustand *x*

Zur Plausibilität der Xcont-Parametrisierung



Situation

- beim Übergang von S1 nach S1 nach Konsumtion von A möge weitere Nachricht A eintreffen