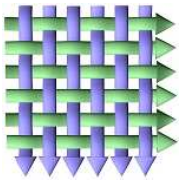
SFB 559 — Teilprojekt M1
LS Informatik IV
Universität Dortmund
23.03.03
Version 1.2

Sonderforschungsbereich 559

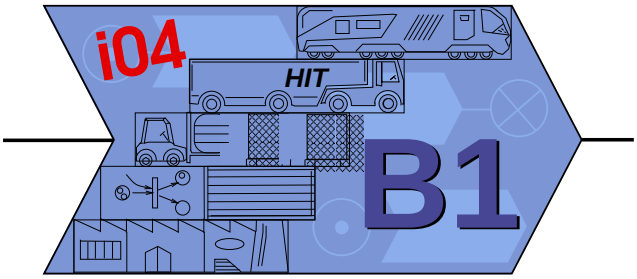
Modellierung großer Netze in der Logistik

Technical Report 03001
ISSN 1612-1376



Semantik des ProC/B-Paradigmas

Spezifikation des ProC/B-Formalismus anhand der Umsetzung in HiSlang



SFB 559 — Teilprojekt M1

Falko Bause, Heinz Beilner, Mathias Schwenke

Zusammenfassung

Der Sonderforschungsbereiches 559 benutzt als Modellbeschreibung den ProC/B-Modellformalismus. In diesem Dokument wird die Bedeutung der ProC/B-Modell-Bestandteile anhand ihrer Übersetzung in HiSlang-Code spezifiziert.

Dieses Dokument basiert auf der Version 0.9 des Prozesskettenеditors.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Dokument spezifiziert die genaue Bedeutung der Modellelemente des ProC/B-Formalismus.

Der erste Teil eines jeden Absschnitts stellt die Modellelemente des ProC/B-Formalismus (siehe [B1]) vor, neben einer kurzen Beschreibung, dem ProC/B-Bild des Elements und Verweisen auf das B1-Dokument sind wichtige Punkte aufgeführt, die bei späteren Konsistenzchecks zu beachten sind.

Um letztendlich die Übersetzung nach HiSlang zu dokumentieren, wird anschließend der HiSlang-Code und Hinweise zur Übersetzung angegeben.

Kapitel 2

Modellelemente

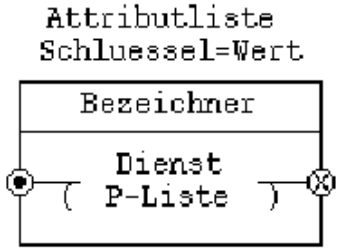
2.1 Funktionseinheiten (FEs)

2.1.1 Funktionseinheit

ProC/B-Bedeutung:

Funktionseinheit (FE):

Die FEs (auch Organisationseinheiten) übernehmen die Durchführung der durch die PKEs beschriebenen Aktivitäten. Hierzu bieten FEs Dienste an, die genutzt werden können.
Neben Standard-FEs (Server, Prioserver, Counter, Storage) können eigene sog. konstruierte FEs erstellt werden, die eine Hierarchisierung ermöglichen.
Siehe **[B1, 31, 42]**.



`Bezeichner`	:	`STRING`	*Als Kommentar im HiSlang-Code*
`Dienste`	:	`STRING`	*Liste von angebotenen Diensten*
`Parameter`	:	`STRING[][]`	*Liste der Parameterlisten der Dienste*
`Attribute`	:	`STRING[]`	*Prägende Attribute*
`Verweise`	:	`STRING[]`	*siehe [B1, 66], externe FEs*

- Der `Bezeichner` bestimmt einen Namen für die Funktionseinheit. Auf diesen Bezeichner wird unter Umständen in anderen Elementen weiter Bezug genommen, weshalb er eindeutig gewählt werden muss. Eine Überprüfung der Eindeutigkeit des Bezeichners übernimmt der Editor.
- Die `Verweise` geben an, auf welchen Dienst welcher FE ein Dienst einer enthaltenen FE abgebildet werden soll, der dort zwar durch eine externe FE angeboten, aber nicht weiter spezifiziert wurde (siehe S. 20).

2.1.2 FE vom Typ Server

ProC/B-Bedeutung:

FE vom Typ Server (ServerFE):

Ein Server modelliert eine Menge von Elementen, an die eine Arbeitsanforderung gestellt werden kann. Die Arbeit verursacht eine Zeitverzögerung (abhängig von den Attributen des Servers), (vgl. Bedienstation aus der Warteschlangentheorie).
Siehe **[B1, 43]**.

`Bezeichner`	`: STRING`	*s.o.*
Attributliste		
`SPEED`	`: REAL`	*AE/ZE jedes Bedienelements*
`SDSPEEDS`	`: [INT][REAL]`	*zustandsabhängige Geschwindigkeit*
`DIS`	`: STRING`	*Bediendisziplin: FCFS/PS/IS*
`CAP`	`: INT`	*Anzahl der Bedienelemente*
Dienste		$=$ `request(amount:real)`

- AE $\hat{=}$ Arbeitseinheiten; ZE $\hat{=}$ Zeiteinheiten
- Der Dienst `request` wird mit dem Umfang der Arbeitsanforderung in der abstrakten Einheit AE parametrisiert.
- Der Parameter `SPEED` (positiv reell) gibt die Geschwindigkeit der Bearbeitung in AE pro ZE jedes einzelnen Bedienelements an. Standardmäßig auf 1.
- Der Parameter `SDSPEEDS` besteht aus einem Vektor von Zuständen (INT) und den dazugehörigen Geschwindigkeiten (positiv reell). Wird SDSPEEDS nicht auf den Standardwert $[[1],[1,0]]$ gesetzt, so wird das durch ein spezielles Symbol für den Server angezeigt.
- Der Parameter `CAP` (positiv und ganzzahlig) gibt die Anzahl der Bedienelemente an. Diese Angabe ist nur sinnvoll, falls **nicht** `IS` als Bediendisziplin angegeben wurde. Standardmäßig auf 1.
- Der Parameter `DIS` gibt die Bediendisziplin des Servers an (standardmäßig auf FCFS, S. [HIT, 294]):

  FCFS **First Come First Serve:** Die Anzahl der Bedienelemente wird als endlich (abhängig von `CAP`) aufgefaßt, ihre Verteilung an auf Bedienung wartende Aufträge erfolgt nach der Reihenfolge ihrer zeitlichen Ankunft. Maximal schreiten also `CAP` verschiedene Arbeitsaufträge gleichzeitig voran, jeweils mit gleicher Geschwindigkeit.

IS **Infinite Server:** Die Anzahl der Bedienelemente wird als unendlich angenommen (CAP wird deswegen ignoriert). Ein Auftrag bewirkt somit eine reine Zeitverzögerung, abhängig vom Umfang der Arbeit und der Geschwindigkeit des Servers.

PS **Processor Sharing:** Jede Arbeitsanforderung wird sofort bearbeitet. Bei n Aufträgen schreitet jeder also mit Geschwindigkeit $\frac{\texttt{CAP}\cdot\texttt{SPEED}}{n}$ voran.

HiSlang-Übersetzung:

ServerFE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 114, 275]**

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
  ⋮
  {*** Server-FE: bezeichner ***}
  COMPONENT bezeichner : server
    ( LET ACCEPT := ALWAYS,
      LET SCHEDULE := schedule_proc,
      LET DISPATCH := dispatch_proc,
      LET OFFER := ALL );
  ⋮
BEGIN
  ⋮
END TYPE _name;
```

- Falls DIS = FCFS:

  schedule_proc ::= FCFS (LET CAPACITY := *CAP*)

  dispatch_proc ::= EQUAL (LET SPEED := *SPEED*)

  – Falls SDSPEEDS $\neq [[1],[1.0]]$: *dispatch_proc* ::= SDEQUAL (LET SDSPEEDS := *SDSPEEDS*,
  LET SPEED := *SPEED*)

- Falls DIS = PS:

  schedule_proc ::= IMMEDIATE

  dispatch_proc ::= SHARED (LET SPEED := *SPEED*CAP*)

  – Falls SDSPEEDS $\neq [[1],[1.0]]$: *dispatch_proc* ::= SDEQUAL (LET SDSPEEDS := *SDSPEEDS*,
  LET SPEED := *SPEED*CAP*)

- Falls DIS = IS:

  schedule_proc ::= IMMEDIATE

  dispatch_proc ::= EQUAL (LET SPEED := *SPEED*)

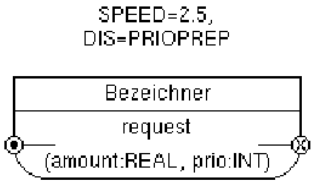
– Falls SDSPEEDS $\neq [[1], [1.0]]$: *dispatch_proc* $::=$ `SDEQUAL ( LET SDSPEEDS := SDSPEEDS,`
`LET SPEED := SPEED )`

2.1.3 FE vom Typ Prioserver

ProC/B-Bedeutung:

FE vom Typ PrioServer (PrioServerFE):

Ein PrioServer modelliert eine Menge von Elementen, an die eine Arbeitsanforderung gestellt werden kann. Die Arbeit verursacht eine Zeitverzögerung (abhängig von den Attributen des Servers). Der Prioserver bedient zunächst die Arbeitsanforderungen mit der höchsten Priorität , (vgl. Bedienstation aus der Warteschlangentheorie).
Siehe **[B1, 43]**.



`Bezeichner`	`: STRING`	*s.o.*
Attributliste		
`SPEED`	`: REAL`	*AE/ZE jedes Bedienelements*
`SDSPEEDS`	`: [INT][REAL]`	*zustandsabhängige Geschwindigkeit*
`DIS`	`: STRING`	*Bediendisziplin: PRIONP/PRIOPREP/PRIOPRES*
Dienste		$=$ `request(amount:real; prio:INT)`

- AE $\hat{=}$ Arbeitseinheiten; ZE $\hat{=}$ Zeiteinheiten
- Der Dienst `request` wird mit dem Umfang der Arbeitsanforderung in der abstrakten Einheit AE sowie der Priorität parametrisiert. 0 steht für die höchste Priorität, 32767 für die geringste.
- Der Parameter `SPEED` (positiv reell) gibt die Geschwindigkeit der Bearbeitung in AE pro ZE jedes einzelnen Bedienelements an. Standardmäßig auf 1.
- Der Parameter `SDSPEEDS` besteht aus einem Vektor von Zuständen (INT) und den dazugehörigen Geschwindigkeiten (positiv reell). Wird SDSPEEDS nicht auf den Standardwert $[[1],[1,0]]$ gesetzt, so wird das durch ein spezielles Symbol für den PrioServer angezeigt.
- Der Parameter `DIS` gibt die Bediendisziplin des PrioServers an (standardmäßig auf PRIONP) (S. [HIT, 288, 294]:

PRIPREP **PRIOority Preemptive REPeat:** Prozesse mit hoher Priorität werden vorrangig bedient. Haben mehrere Prozesse die gleiche Priorität, werden die Prozesse zufällig ausgewählt. Ein Prozess mit niedriger Priorität wird unterbrochen, falls ein Prozess mit hoher Priorität ankommt. Wenn ein Prozess neu startet, so wird die bis dahin verbrauchte Zeit ignoriert, die Abarbeitung des Prozesses beginnt von vorn (Repeat-Strategie).

PRIOPRES **PRIOrity Preemptive RESume:** Ähnlich PRIOPREP. Beim Fortsetzen eines Prozesses startet er an der Stelle, an der er zuvor unterbrochen wurde (Resume-Strategie).

PRIONP **PRIOrity Non Preemptive:** Ähnlich PRIOPREP. Einmal begonnene Prozesse werden für Prozesse höherer Priorität nicht unterbrochen.

HiSlang-Übersetzung:

PrioServerFE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 287]**

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
  ⋮
  {*** PrioServerFE: bezeichner ***}
  COMPONENT bezeichner : prioserver
    ( LET ACCEPT := ALWAYS,
      LET SCHEDULE := schedule_proc,
      LET DISPATCH := dispatch_proc,
      LET OFFER := ALL );
  ⋮
BEGIN
  ⋮
END TYPE _name;
```

- Falls DIS = PRIONP:

  schedule_proc ::= PRIONP

  dispatch_proc ::= EQUAL (LET SPEED := *SPEED*)

  - Falls SDSPEEDS $\neq [[1],[1.0]]$: *dispatch_proc* ::= SDEQUAL (LET SDSPEEDS := *SDSPEEDS*,
    LET SPEED := *SPEED*)

- Falls DIS = PRIOPRES:

  schedule_proc ::= PRIOPRES

  dispatch_proc ::= EQUAL (LET SPEED := *SPEED*)

  - Falls SDSPEEDS $\neq [[1],[1.0]]$: *dispatch_proc* ::= SDEQUAL (LET SDSPEEDS := *SDSPEEDS*,
    LET SPEED := *SPEED*)

- Falls DIS = PRIOPREP:

  schedule_proc ::= PRIOPREP

  dispatch_proc ::= EQUAL (LET SPEED := *SPEED*)

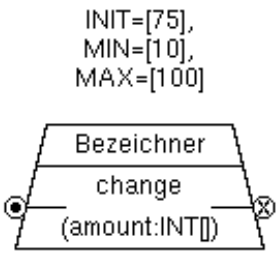
  - Falls SDSPEEDS $\neq [[1],[1.0]]$: *dispatch_proc* ::= SDEQUAL (LET SDSPEEDS := *SDSPEEDS*,
    LET SPEED := *SPEED*)

2.1.4 FE vom Typ Counter

ProC/B-Bedeutung:

FE vom Typ Counter (CounterFE):

Der Counter modelliert einen mehrdimensionalen Raum, dessen Elemente angefordert bzw. wieder freigegeben werden können. Damit lassen sich in begrenzter Anzahl vorhandene Mittel modellieren, die entnommen und wieder zurückgegeben werden können.
Siehe **[B1, 46]**.



`Bezeichner`	`: STRING`	*s.o.*
Attributliste		
`INIT`	`: INT_VEK`	*Beschreibt Initialbelegung*
`MIN`	`: INT_VEK`	*Belegungs-Untergrenzen der Dimensionen*
`MAX`	`: INT_VEK`	*Belegungs-Obergrenzen der Dimensionen*
`DIS`	`: STRING`	*Wartedisziplin: RANDOM*
Dienste		= `change(amount:[INT])`

- Das Attribut `INIT` legt die initiale Anzahl von Elementen in jeder Dimension fest. Durch die Dimension dieses Vektors wird die Dimension des Counters gesetzt.
- Die Unter- und Obergrenzen der Dimensionen legen die minimalen bzw. maximalen Anzahlen von Elementen fest (z.B. Lagergröße).
- Ein Aufruf des Dienstes `change` verändert die Anzahl der Elemente in den Dimensionen des Raumes um den angegebenen Betrag ($(n_1, n_2, \ldots, n_d) := (n_1, n_2, \ldots, n_d) +$ `amount`, wobei `amount` der d-dimensionale Parametervektor ist). Wird dabei eine Ober- oder Untergrenze verletzt, wird gewartet (siehe Wartedisziplin).
- Sind durch eine Veränderung des Counters mehrere bisher wartende Aufrufe des Dienstes `change` durchführbar, bestimmt die Wartedisziplin `DIS`, wie der zuerst durchzuführende Aufruf ausgewählt wird. Bisher ist nur die (standardmäßige) Disziplin `RANDOM` vorgesehen, die diesen Aufruf nach gleichverteiltem Zufall auswählt.

HiSlang-Übersetzung:

CounterFE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 116, 278]**

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
  ⋮
```

```
  {*** Counter: bezeichner ***}
  COMPONENT bezeichner : counter
    ( LET min := MIN,
      LET max := MAX,
      LET init := INIT,
      LET schedule := schedule_proc );
  ⋮
BEGIN
  ⋮
END TYPE _name;
```

- Falls DIS = RANDOM:

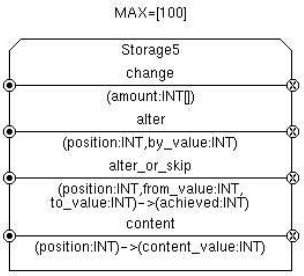
  schedule_proc ::= crandom

2.1.5 FE vom Typ Storage

ProC/B-Bedeutung:

FE vom Typ Storage (LagerFE):

Das Storage erweitert den Counter um spezielle Überwachungsfunktionen, die helfen, Informationen über den Zustand zu gewinnen. Das Storage modelliert einen mehrdimensionalen Raum, dessen Elemente angefordert bzw. wieder freigegeben werden können.
Siehe –.



`Bezeichner`	`: STRING`	*s.o.*
Attributliste		
`MIN`	`: INT_VEK`	*Belegungs-Untergrenzen der Dimensionen*
`MAX`	`: INT_VEK`	*Belegungs-Obergrenzen der Dimensionen*
`INIT`	`: INT_VEK`	*Beschreibt Initialbelegung*
Dienste		= `change(amount:[INT]; prio:INT)`
		= `alter(position:INT; by_value:INT)`
		= `alter_or_skip`
		`(position:INT; from_value:INT; to_value:INT)`
Prozeduren		= `content(position:INT)`

- Das Attribut `INIT` legt die initiale Anzahl von Elementen in jeder Dimension fest. Durch die Dimension dieses Vektors wird die Dimension des Storages gesetzt.
- Die Unter- und Obergrenzen der Dimensionen legen die minimalen bzw. maximalen Anzahlen von Elementen fest.
- Ein Aufruf des Dienstes `change` verändert die Anzahl der Elemente in den Dimensionen des Raumes um den angegebenen Betrag ($(n_1, n_2, \ldots, n_d) := (n_1, n_2, \ldots, n_d) +$ `amount`, wobei `amount` der d-dimensionale Parametervektor ist). Wird dabei eine Ober- oder Untergrenze verletzt, wird gewartet (Als Wartedisziplin dient grundsätzlich `CRANDOM`).
- Sind durch eine Veränderung des Storages mehrere bisher wartende Aufrufe des Dienstes `change` durchführbar, werden die Aufrufe zufällig ausgewählt.
- Ein Aufruf des Dienstes `alter` verändert die Anzahl des Elements `position` um den Wert `by_value`.
- Der Dienst `alter_or_skip` dient zur Behandlung von “out of stock”-Situationen und Teillieferungen. Von der Position `position` wird möglichst die Menge `from_value` entnommen (oder eingelagert), wenn das nicht möglich ist, so viel wie möglich, aber mindestens `to_value`. Ist auch das nicht möglich, so wird gar nichts entnommen (bzw. eingelagert). Der Rückgabewert informiert über die entnommene (bzw. eingelagerte) Menge.

- Ein Aufruf der Prozedur `content` liefert den Bestand des Elements `position`.
- Die Möglichkeit, das Storage mit Prioritäten zu benutzen, ist im Editor nicht vorgesehen.

HiSlang-Übersetzung:

LagerFE (HiSlang-Code mit Umgebung)
Siehe

```
TYPE _name {MODEL|COMPONENT}[(_parameter)];
  ⋮
  {*** Lager: bezeichner ***}
  TYPE bezeichner_typ COMPONENT
          (min : ARRAY OF INTEGER;
           max : ARRAY OF INTEGER;
           init : ARRAY OF INTEGER);
    PROVIDE
      SERVICE change(amount : ARRAY OF INTEGER;
                     prio : INTEGER DEFAULT 32767);
      SERVICE alter (position : INTEGER DEFAULT 1;
                     by_value : INTEGER);
      SERVICE alter_or_skip (position : INTEGER DEFAULT 1;
                     from_value : INTEGER;
                     to_value : INTEGER)
                     RESULT INTEGER;
      PROCEDURE content(position : INTEGER DEFAULT 1)
                     RESULT INTEGER;
    END PROVIDE;

    Streamdefinition

    VARIABLE
      state_array : ARRAY [1..init.upper_bounds[1]]
            OF INTEGER;

    COMPONENT Innenlager : counter
      (LET MIN := min,
      LET MAX := max,
      LET INIT := init,
      LET SCHEDULE := CRANDOM);

    TYPE INIT_state SERVICE (init :ARRAY OF INTEGER);
      VARIABLE position : INTEGER;
    BEGIN
      FOR position:=1 STEP 1 UNTIL init.upper_bounds[1] LOOP
```

```
    state_array[position]:=init[position];
    initStreamBlock;
  END LOOP;
END SERVICE TYPE INIT_state;

TYPE change SERVICE(amount : ARRAY OF INTEGER;
          prio : INTEGER DEFAULT 32767);
  USE
    SERVICE Aufruf_lager_aktualisieren(
         amount : ARRAY OF INTEGER;
         prio : INTEGER DEFAULT 32767);
  END USE;

  VARIABLE position : INTEGER;
 BEGIN
  Aufruf_lager_aktualisieren(amount);
  FOR position:= 1 STEP 1
          UNTIL amount.upper_bounds[1] LOOP
    state_array[position]
             := state_array[position] + amount[position];
    updateStreamBlock;
  END LOOP;
END SERVICE TYPE change;

TYPE alter SERVICE(position : INTEGER DEFAULT 1;
         by_value : INTEGER);
  USE
    SERVICE Aufruf_lager_aktualisieren
             (amount : ARRAY OF INTEGER;
             prio : INTEGER DEFAULT 32767);
  END USE;

  VARIABLE
    amount : ARRAY[1 .. init.upper_bounds[1]] OF INTEGER;
    i : INTEGER;
BEGIN
  FOR i:= 1 STEP 1 UNTIL init.upper_bounds[1] LOOP
    amount[i]:=0;
  END LOOP;
  amount[position] := by_value;
  Aufruf_lager_aktualisieren(amount);
  state_array[position]:=state_array[position]+by_value;
  updateStreamBlock;
END SERVICE TYPE alter;

TYPE alter_or_skip SERVICE(position : INTEGER DEFAULT 1;
                 from_value : INTEGER;
```

```
                    to_value : INTEGER )
          RESULT INTEGER;
    USE
      SERVICE Aufruf_lager_aktualisieren
                      (amount : ARRAY OF INTEGER;
                      prio : INTEGER DEFAULT 32767);
    END USE;

    VARIABLE
      amount: ARRAY[1 .. init.upper_bounds[1]] OF INTEGER;
      i: INTEGER;
   BEGIN
    FOR i:= 1 STEP 1 UNTIL init.upper_bounds[1] LOOP
      amount[i]:=0;
    END LOOP;
    IF from_value < 0 THEN [1]
      IF (state_array[position] - abs(from_value))
              >= min[position] THEN [2]
        amount[position] := from_value;
      ELSE [3]
        IF (to_value <= 0) AND (to_value >= from_value) THEN
[4]
          IF (state_array[position] - abs(to_value))
                  >= min[position] THEN [5]
            amount[position]
                    := (state_array[position]
                    - min[position]) *(-1);
            alter_or_skipStreamBlock1
            ELSE [6]
              amount[position] := 0;
              alter_or_skipStreamBlock2
            END IF;
          ELSE [7]
            writeln(''Warnung: kein gueltiger Wert
                    fuer 'to_value'');
            amount[position]:=0;
          END IF;
        END IF;
      ELSE [8]
```

[1] Abfrage, ob etwas entnommen werden soll
[2] Abfrage, ob die gesamte Menge `from_value` entnommen werden kann
[3] Es kann nicht die gesamte Menge `from_value` entnommen werden.
[4] Abfrage, ob korrekter Wert für `to_value`
[5] Abfrage, ob zumindest noch die Menge `to_value` ausgelagert werden kann
[6] Es kann nichts mehr ausgelagert werden.
[7] Fehlerhafter Wert für `to_value`
[8] Einlagern

```
    IF (state_array[position] + from_value)
            <= max[position] THEN [9]
      amount[position] := from_value;
    ELSE[10]
      IF (to_value >= 0) AND (to_value <= from_value) THEN[11]
        IF (state_array[position] + to_value)
                <= max[position] THEN [12]
          amount[position]
                  := max[position] - state_array[position];
          alter_or_skipStreamBlock3
        ELSE[13]
          amount[position] := 0;
          alter_or_skipStreamBlock4
        END IF;
      ELSE
        writeln(''Warnung: kein gueltiger Wert
                fuer 'to_value'');
        amount[position]:=0;
      END IF;
    END IF;
  END IF;
  Aufruf_lager_aktualisieren(amount);
  state_array[position]
          :=state_array[position]+amount[position];
  updateStreamBlock
  RESULT amount[position];
END SERVICE TYPE alter_or_skip;

PROCEDURE content(position : INTEGER DEFAULT 1)
        RESULT INTEGER;
BEGIN
  RESULT state_array[position];
END PROCEDURE;

REFER
  change, alter, alter_or_skip
TO
  Innenlager
EQUATING
  change.Aufruf_lager_aktualisieren
    WITH Innenlager.change;
  alter.Aufruf_lager_aktualisieren
```

[9] Abfrage, ob die gesamte Menge `from_value` eingelagert werden kann

[10] Es kann nicht die gesamte Menge `from_value` eingelagert werden.

[11] Abfrage, ob korrekter Wert für `to_value`

[12] Abfrage, ob noch mindestens die Menge `to_value` eingelagert werden kann

[13] Es kann nichts mehr eingelagert werden.

```
      WITH Innenlager.change;
    alter_or_skip.Aufruf_lager_aktualisieren
      WITH Innenlager.change;
  END REFER;

 BEGIN
  CREATE 1 PROCESS INIT_state(init) AT 0.0;
 END COMPONENT TYPE bezeichner_Typ;
 ⋮
BEGIN
 ⋮
END TYPE _name;
```

- Falls in dem Storage Messungen vorgenommen werden, n ist die Dimension des Storages: *Streamdefinition* ::=

```
STREAM
  state1 ... staten : STATE;
 STREAM
  in1 ... inn : EVENT;
 STREAM
  out1 ... outn: EVENT;
 STREAM
  outofstock_calls1 ... outofstock_callsn: COUNT;
 STREAM
  outofstock_amount1 ... outofstock_amountn: EVENT;
 STREAM
  outofspace_calls1 ... outofspace_callsn: COUNT;
 STREAM
  outofspace_amount1 ... outpfspace_amountn: EVENT;
```

initStreamBlock ::=

```
CASE position
  WHEN 1 : UPDATE state1 BY init[position];
  ⋮
  WHEN n : UPDATE staten BY init[position];
END CASE;
```

updateStreamBlock ::=

```
CASE position
   WHEN 1 :
    UPDATE state1 BY amount[position];
    IF amount[position] > 0 THEN UPDATE in1 BY amount[position];
END IF;
    IF amount[position] < 0 THEN UPDATE out1 BY -amount[position];
END IF;
```

```
⋮
 WHEN n :
   UPDATE staten BY amount[position];
   IF amount[position] > 0 THEN UPDATE inn BY amount[position];
END IF;
   IF amount[position] < 0 THEN UPDATE outn BY -amount[position];
END IF;
END CASE;
```

alter_or_skipStreamBlock1 ::=

```
CASE position
   WHEN 1 :
    UPDATE outofstock_calls1 BY 1;
    UPDATE outofstock_amount1 BY (abs(from_value) - abs(amount[position]));
⋮
 WHEN n :
   UPDATE outofstock_callsn BY 1;
   UPDATE outofstock_amountn BY (abs(from_value) - abs(amount[position]));
END CASE;
```

alter_or_skipStreamBlock2 ::=

```
CASE position
   WHEN 1 :
    UPDATE outofstock_calls1 BY 1;
    UPDATE outofstock_amount1 BY abs(from_value);
⋮
 WHEN n :
   UPDATE outofstock_callsn BY 1;
   UPDATE outofstock_amountn BY abs(from_value);
END CASE;
```

alter_or_skipStreamBlock3 ::=

```
CASE position
   WHEN 1 :
    UPDATE outofspace_calls1 BY 1;
    UPDATE outofspace_amount1 BY (from_value - amount[position]);
⋮
 WHEN n :
   UPDATE outofspace_callsn BY 1;
   UPDATE outofspace_amountn BY (from_value - amount[position]);
END CASE;
```

alter_or_skipStreamBlock4 ::=

```
CASE position
   WHEN 1 :
    UPDATE outofspace_calls1 BY 1;
    UPDATE outofspace_amount1 BY (from_value);
⋮
```

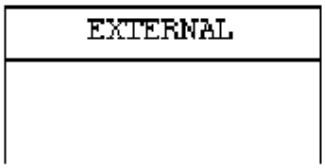
```
  WHEN n :
    UPDATE outofspace_callsn BY 1;
    UPDATE outofspace_amountn BY (from_value);
END CASE;
```

2.1.6 Externe Funktionseinheit

ProC/B-Bedeutung:

Externe Funktionseinheit (ExterneFE):

Die Externe FE ist eigentlich Bestandteil der Schnittstelle der sie enthaltenden FE. Sie bietet Dienste an, die von einer anderen FE realisiert werden. Die Anbindung an die tatsächlichen Dienste geschieht auf der nächsthöheren Hierarchieebene der Enthaltensein-Relation. Somit sind Dienste nutzbar, die lokal in einer FE gar nicht sichtbar sind.
Siehe **[B1, 66]**.



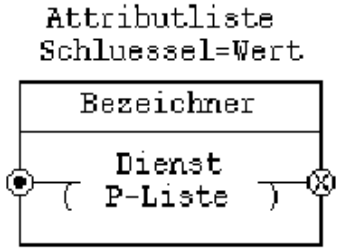
`Dienste`	`: STRING`	*Liste von angebotenen Diensten*
`Parameter`	`: STRING[][]`	*Liste der Parameterlisten der Dienste*

2.1.7 Konstruierte Funktionseinheit

ProC/B-Bedeutung:

Konstruierte Funktionseinheit (KonstrFE):

Neben den Standard-FEs können auch vom Modellierer konstruierte FEs in ProC/B-Modellen eingesetzt werden. Diese enthalten einen Prozesskettenplan, der selber Prozessketten und FEs enthalten darf. Über virtuelle Quellen und Senken lassen sich Dienste an interne Prozessketten der FE anbinden. Solche Prozesse werden dann nicht durch bedingte oder unbedingte Quellen instanziiert, sondern durch einen Dienstaufruf. Nach Erreichen der virtuellen Senke erfolgt eine Rückmeldung an die aufrufende Umgebung.
Siehe **[B1, 49, 52]**.



`Bezeichner`	`:`	`STRING`	
`Dienste`	`:`	`STRING[]`	*Liste von angebotenen Diensten*
`Parameter`	`:`	`STRING[][]`	*Liste von Parameterlisten der Dienste*
`Attribute`	`:`	`STRING[]`	*Prägende Attribute*
`Verweise`	`:`	`STRING[]`	*siehe [B1, 66], externe FEs*

2.2 Prozesskette (PK)

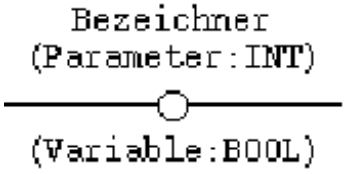
Eine Prozesskette definiert die Verhaltensvorschriften für Prozesse einer spezifischen Art, einer Familie von Prozessen. Sie spielt die Rolle eines Prozessmusters, einer Vorschrift, die für jeden Prozess dieser Familie gilt. Siehe [B1, 27ff]

2.2.1 Prozesskettenschnittstelle

ProC/B-Bedeutung:

Prozesskettenschnittstelle (PKSchnittstelle):

Startpunkt einer PK, welcher von Quellen dazu genutzt wird konkrete Prozesse (Instanzen) zu starten.
Siehe **[B1, 34]**.



`Bezeichner`	`: STRING`	*beschreibt die PK*
`Eingabeparameter`	`: STRING`	*Eingabeparameterliste der PK*
`Variablen`	`: STRING`	*Variablendeklaration der PK*
`Ausgabeparameter`	`: STRING`	*Ausgabeparameterliste*

- Die **Eingabeparameterliste** wird im Editor nicht als Text, sondern einzeln in eine Liste eingetragen. D.h. eine Umwandlung dieser Eingabe in eine einzelne (dementsprechend lange) Zeichenkette ist nötig. Gleiches gilt für die **Variablendeklaration** (Zielform in der Form `i : INT INIT 0`) sowie die **Ausgabeparameterliste**.

HiSlang-Übersetzung:

Prozesskettenschnittstelle (HiSlang-Code mit Umgebung)
Siehe **[HIT, 73]**

```
TYPE bezeichner SERVICE [(parameter)]
  [RESULT ausgabeparameter];
  [USE SERVICE
    {dienst(aufrufparameter:TYP) RESULT TYPEN;}
    ⋮
  END USE;]
  [VARIABLE
    {variable : TYP;}
    ⋮]
BEGIN
  {PKEs}
```

```
  ⋮
  {Senke}
END TYPE bezeichner;
```

- Der Code beschreibt eine Typendeklaration eines Services.
- Die *ausgabeparameter* werden als Liste von Typen angegeben, jeweils durch Kommata separiert.

2.2.2 Prozesskonnektoren

Mittels Konnektoren lassen sich logische Zusammenhänge zwischen Aktivitätsteilmengen zur Erfassung alternativer beziehungsweise gemeinsamer Fortsetzungen sowie parallelisierender und synchronisierender Reihenfolgevorschriften beschreiben.

Es stehen Oder-Konnektoren sowie Und-Konnektoren zur Verfügung. Zu beachten ist, dass die Konnektoren entsprechend “regulärer Klammerungen” anzuordnen sind, d. h. bei einer Verschachtelung von Konnektoren ist der untergeordnete (also später geöffnete) Konnektor zunächst wieder zu schließen, bevor der übergeordnete Konnektor geschlossen werden kann.

Zum Schließen eines Oder-Konnektors steht das gleiche GUI-Element zur Verfügung wie zum Öffnen.

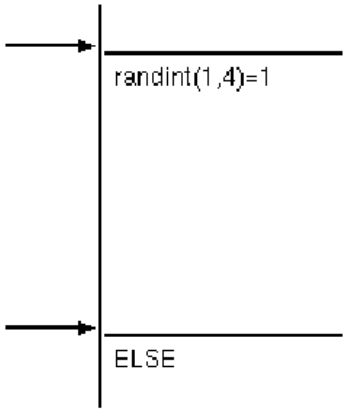
Ein schließender und ein öffnender Und-Konnektor lassen zusammenfassen zum schließenden und öffnenden Und-Konnektor, ein schließender Und- und ein öffnender Oder-Konnektor lassen sich zusammenfassen mit dem schließenden Und-Konnektor, ein schließender Oder- und ein öffnender Und-Konnektor mit dem öffnenden Und-Konnektor sowie ein schließender und ein öffnender Oder-Konnektor mit dem Oder-Konnektor.

2.2.2.1 Oder-Konnektor

ProC/B-Bedeutung:

Oder-Konnektor (OderOeffnend, OderSchliessend):

Öffnende Oder-Konnektoren ermöglichen eine alternative Fortsetzung von Prozessen, schließende Oder-Konnektoren die Zusammenführung (und damit gemeinsame Fortführung) alternative Zweige. Für beide wird im Editor das selbe GUI-Element benutzt.
Die Verzweigung kann nach Wahrscheinlichkeiten (probabilistisch) oder nach Bedingungen (boolesch) erfolgen.
Siehe **[B1, 37, 40]**.



1. Fall (nach Wahrscheinlichkeiten)

`Wahrscheinlichkeit`	`: (STRING → REAL)[]`	*Wert* ≤ 1*; Summe* ≤ 1
`ELSE`	`: ''ELSE''`	*optional*

2. Fall (nach Bedingungen)

`Bedingung`	`: (STRING → BOOL)[]`	
`ELSE`	`: ''ELSE''`	*optional*

- Der Konnektor kann mehrere Eingänge und mehrere Ausgänge haben.
- Die Wahrscheinlichkeiten werden als Ausdrücke angegeben, die Werte aus $[0, 1]$ ergeben müssen. Die Summe aller Wahrscheinlichkeiten muss ≤ 1 sein.
- An den ausgehenden Kanten können auch Bedingungen stehen. Wertet sich genau eine Bedingung zu wahr aus, so wird genau über diese Kante verzweigt. Wertet sich keine aus, dann wird der ELSE-Zweig genommen. Ist dieser nicht vorhanden, ist das Modell ungültig. Falls mehrere Bedingungen wahr sind, ist unklar, wohin verzweigt wird (wahrscheinlich wird dies nach Kantenindex geschehen, ist aber in dem Sinne nicht spezifiziert, d.h. explizite Prioritäten gibt es nicht).
- Die Wahrscheinlichkeit $p = 1$ ist möglich (im Gegensatz zur Definition im B1-Formalismus).
- Es wird nicht die Idee verfolgt, statt Wahrscheinlichkeiten Häufigkeiten zu verwenden.

HiSlang-Übersetzung:

Oder (probabilistisch, nach Wahrscheinlichkeiten) (HiSlang-Code mit Umgebung)
Siehe —

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
  ⋮
BEGIN
  ⋮
  *** --- : OderOeffnend (id=id) ***
  BRANCH
    PROB Wahrscheinlichkeit1
      *** (ProzessAnfang) : LinearePK (id=id) ***
      ⋮
    [ PROB Wahrscheinlichkeit2    ⋮
    ]
    ⋮
    [ PROB Wahrscheinlichkeitn    ⋮
    ]
    [ ELSE
    ⋮
    ]
  END BRANCH;
  *** --- : OderSchliessend (id=id) ***

⋮
END TYPE _name;
```

HiSlang-Übersetzung:

Oder (boolesch, nach Bedingungen) (HiSlang-Code mit Umgebung)
Siehe —

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
  ⋮
BEGIN
  ⋮
  *** --- : OderOeffnend (id=id) ***
  IF Bedingung1 THEN
    *** (ProzessAnfang) : LinearePK (id=id) ***
    ⋮
```

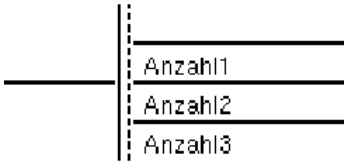
```
  [ ELSE
  IF Bedingung2 THEN
    ⋮
  ]
  ⋮
  [ ELSE
  IF Bedingungn THEN
    ⋮
  ]
  [ ELSE
  ⋮
  ]
  END IF;
  *** --- : OderSchliessend (id=id) ***

⋮
END TYPE _name;
```

2.2.2.2 Öffnender Und-Konnektor

ProC/B-Bedeutung:

Öffnender Und-Konnektor (UndOeffnend):

Dieser Konnektor ermöglicht eine gleichzeitige/parallele Fortsetzung von Prozessen; ggf. eine Erzeugung mehrerer Prozesse pro ausgehende Kante.
Siehe **[B1, 39]**.



`Anzahl : (STRING → INT)[]` *initial 1*

- Über den Parameter `Anzahl` können mehrere Prozesse parallel über einen Zweig geschickt werden. Entsprechend müssen diese mit dem schließenden Und-Konnektor wieder zusammengefasst werden.

HiSlang-Übersetzung:

UndOeffnend (HiSlang-Code mit Umgebung)
Siehe

```
TYPE _name {MODEL | COMPONENT} [(_parameter)];
  ⋮
BEGIN
  ⋮
  *** --- : UndOeffnend (id=id) ***
  CONCURRENT
    Prozess1
  TO
    *** Wiederholung: 1 ***
    Prozess1
  ⋮
  TO
    *** Wiederholung: anzahl - 1 ***
    Prozess1
  TO
    Prozess2
  TO
  ⋮
  TO
    Prozessn
  END CONCURRENT;
```

```
  ⋮
END TYPE _name;
```

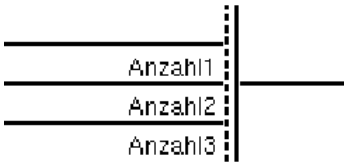
- Es gebe n verschiedene parallel auszuführende Prozesse.

2.2.2.3 Schließender Und-Konnektor

ProC/B-Bedeutung:

Schließender Und-Konnektor (UndSchliessend):

Mit Hilfe des schließenden Und-Konnektors werden parallele Abläufe eines Prozesses zusammengeführt; ggf. an jeder eingehenden Kante eine bestimmte Anzahl.
Siehe **[B1, 39]**.



`Anzahl : (STRING → INT)[]` *initial 1*

- Beim schließenden Und-Konnektor kann nur ein Prozess mit sich selbst zusammengeführt werden. Andere Prozesse der gleichen Prozesskette haben eine andere Prozess-ID und werden bei diesem Konnektor nicht verschmolzen.

- Die schließenden Und-Konnektoren brauchen nicht übersetzt zu werden. Dies geschieht mit der Übersetzung der öffnenden Und-Konnektoren.

2.2.3 Prozesskettenelemente

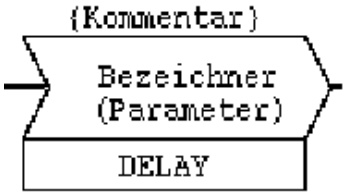
Mit den Prozesskettenelementen werden die Aktivitäten in einer Prozesskette erfasst.

2.2.3.1 Delay-Prozesskettenelement

ProC/B-Bedeutung:

Delay-Prozesskettenelement (DelayPKE):

Spezialform des allgemeinen PKE. Dieses PKE verbraucht (z.B. wartet) eine vorgegebene Zeitdauer (vgl. Verzögerungsstation).
Siehe **[B1, 36]**.



`Kommentar`	`: STRING`	*wird nicht in den HiSlang-Code übersetzt*
`Bezeichner`	`: STRING`	*s.o.*
`Parameter`	`: STRING → REAL`	*Vorgabezeit als Wert bzw. Ausdruck*
`Ausf.-Hinw.`	`: STRING`	`DELAY`

HiSlang-Übersetzung:

DelayPKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 88, 257]**

```
TYPE _name {MODEL | COMPONENT} [(_parameter)];
⋮
BEGIN
  ⋮
  {*** Delay-PKE: bezeichner ***}
  hold(vorgabezeit);
  ⋮
END TYPE _name;
```

- Prinzipiell hat man mehrere Möglichkeiten, eine Zeitverzögerung in HiSlang zu realisieren. Wir haben uns für den Befehl **hold** entschieden, da dieser eine Zeitverzögerung unabhängig von dem HiSlang-Parameter `speed` erlaubt. Im ProC/B-Modellformalismus ist es nicht möglich einen entsprechenden Parameter `speed` für konstruierte Funktionseinheiten oder das gesamte Modell anzugeben. Daher wird der HiSlang-Parameter `speed` immer den Standardwert 1 haben und der Befehl `spend` würde unnötige Rechenoperationen verursachen.

2.2.3.2 Code-Prozesskettenelement

ProC/B-Bedeutung:

Prozesskettenelement vom Typ Code (CodePKE):

Spezialform des allgemeinen PKEs. Dieses PKE führt Anweisungen aus, die Parameter und Variablen des Prozesses verändern.
Siehe **[B1, 36]**.

`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Anweisungen`	`: STRING`	*Auszuwertende Anweisungen (Parameter)*
`Ausf.-Hinw.`	`: STRING`	`CODE`

- Als **Anweisungen** ist gültiger HiSlang-Code erlaubt, etwa arithmetische Ausdrücke. Kontrollausgaben können mit `writeln` erzeugt werden. Die Anweisungen können auch mehrere HiSlang-Codezeilen enthalten, die durch Semikolons zu trennen sind.
- Für größere Eingaben kann statt des Prozesskettenelements vom Typ Code auch das Code-Element verwendet werden, in dem sich mehrere Codezeilen eingeben lassen.
- Die **Anweisungen** sollten keine Befehle enthalten, die Modellzeit verbrauchen. Dies würde der zeitlosen Natur des Code-PKEs widersprechen.
- Auskunft über HiSlang-Anweisungen gibt das HiSlang-Reference Manual, [HIT, 21ff]

HiSlang-Übersetzung:

CodePKE (HiSlang-Code mit Umgebung)
Siehe —

```
TYPE _name {MODEL|COMPONENT}[(_parameter)];
⋮
BEGIN
  ⋮
  {*** Code-PKE: bezeichner ***}
  anweisungen;
  ⋮
END TYPE _name;
```

- Bei den *anweisungen* handelt es sich um HiSlang-Code. Sie können auch mehrere Befehlszeilen enthalten.

  Die *anweisungen* werden, ohne deren Form zu ändern, an der entsprechenden Zeilenposition in den Begin-End-Teil eines HiSlang-Services eingefügt.

- An die *anweisungen* wird immer ein Semikolon angefügt.

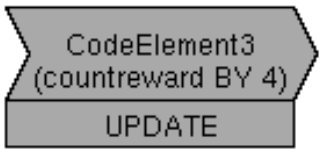
- Statt des Prozesskettenelements kann auch das Code-Element verwendet werden, dass mehrere Anweisungen in separaten Codezeilen aufnimmt. Die Codezeilen brauchen nicht mit Semikola abgeschlossen zu werden.

2.2.3.3 Update-Prozesskettenelement

ProC/B-Bedeutung:

Update-Prozesskettenelement (UpdatePKE):

Das Update-PKE dient zur Benutzung von Rewards (Streams in HiSlang). Dabei wird angegeben, mit welchem Wert der selbstdefinierte Reward aktualisiert werden soll.
Siehe **Rewards, S. 66**.



`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Name des Rewards`	`: STRING`	
`Wert`	`: STRING`	

HiSlang-Übersetzung:

UpdatePKE (HiSlang-Code mit Umgebung)
Siehe —

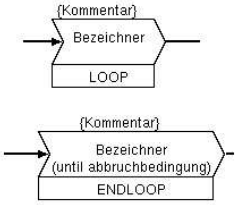
```
TYPE _name {MODEL | COMPONENT} [(_parameter)];
⋮
BEGIN
  ⋮
  {*** Code-PKE: bezeichner ***}
  UPDATE Name des Rewards BY Wert;
  ⋮
END TYPE _name;
```

2.2.3.4 Loop-Prozesskettenelemente

ProC/B-Bedeutung:

LOOP-Prozesskettenelemente (LoopPKE):

Die LOOP/ENDLOOP-Elemente können zur Realisierung von Schleifen verwendet werden. Eine Schleife beginnt immer mit einem Element vom Typ LOOP und endet mit einem Element vom Typ ENDLOOP. Der zwischen diesen beiden Elementen liegende Teil der Prozesskette wird so lange ausgeführt, bis die angegebene Abbruchbedingung zutrifft.
Siehe —.



`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Abbruchbedingung`	`: STRING → BOOLEAN`	

- Die **Abbruchbedingung** ist ein boolscher Ausdruck. Sie ist im **EndLoop-PKE** einzutragen.
- LOOP und ENDLOOP sind entsprechend “regulärer Klammerstrukturen” anzuwenden. LOOP und ENDLOOP sind also in der gleichen logischen Klammerebene zu benutzen, das heißt, vor dem LOOP geöffnete Strukturen dürfen erst nach dem ENDLOOP wieder geschlossen werden, innerhalb von LOOP/ENDLOOP geöffnete Strukturen müssen auch innerhalb von LOOP/ENDLOOP wieder geschlossen werden.

HiSlang-Übersetzung:

LoopPKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 41]**

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
⋮
BEGIN
  ⋮
  LOOP
  PKEs
  END LOOP UNTIL abbruchbedingung;
  ⋮
END TYPE _name;
```

2.2.3.5 Aufruf-Prozesskettenelement

ProC/B-Bedeutung:

Aufruf-Prozesskettenelement (AufrufPKE):

Spezialformen des allgemeinen PKE. Diese PKEs rufen einen Dienst einer Funktionseinheit auf. Dies kann zunächst die Benutzung eines (exklusiven) `Servers` oder eines `Counters` sein.
Siehe **[B1, 35], vgl. S. 39, S. 43**.

`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Parameter`	`: STRING`	*Übergabep.-liste als Wert oder Ausdruck*
`Ausf.-Hinw.`	`: STRING`	`Funktionseinheit.Dienst`
`Variablen`	`: STRING`	*Übergabe an eigene Variablen, etc.*

- Die angegebenen **Variablen** werden nach Ausführung des Aufruf-PKEs mit dessen Ausgabewerten belegt.
- **Typen** und **Anzahl** von Eingabe- sowie Ausgabeparametern werden implizit durch `Funktionseinheit.Di` festgelegt.

HiSlang-Übersetzung:

AufrufPKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 77, 202]**

```
TYPE _compname {MODEL | COMPONENT}[(_parameter)];
 ⋮
TYPE _servname SERVICE [(_parameter)]
  [RESULT _ausgabetypen];
  USE SERVICE
    ⋮
    {*** Aufruf-PKE: bezeichner (used Service) ***}
    bezeichner[({parameterbezeichner:typ} [; ...])]
              [RESULT ausgabetyp [, ...]];
    ⋮
  END USE;
  [VARIABLE
    {variable : typ} [, ...];]
```

```
  ⋮
  BEGIN
    {PKEs}
    ⋮
    {*** Aufruf-PKE: bezeichner (Aufruf) ***}
    [{variable | {( variable [, ...] )}} :=]
      bezeichner[({parameter} [, ...])];
    {Senke}
  END TYPE servname;
  ⋮
  REFER servname [, ...] TO funktionseinheit [, ...] EQUATING
    bezeichner WITH funktionseinheit.dienst;
    ⋮
   END REFER;
END TYPE compname;
```

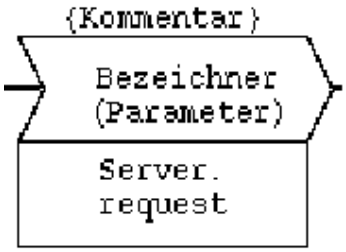
- Für jedes Aufruf-PKE wird ein eigener **used Service** erzeugt. Dieser wird per REFER auf die Komponente (den Dienst) bezogen, die der angegebenen Funktionseinheit (dem angegebenen Dienst) entspricht.

2.2.3.6 Server-Prozesskettenelement

ProC/B-Bedeutung:

Server-PKE (Server-PKE):

Spezialform des Aufruf-PKEs. Dieses PKE ruft einen Server auf. Der Server kann auch von anderen Server-PKEs genutzt werden.
Siehe **[B1, 43]**.



`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Parameter`	`: STRING → REAL`	*Arbeitsmenge (*`amount`*) als Ausdruck*
`Ausf.-Hinw.`	`: STRING`	`Server.request`

- Die Arbeitsmenge bei Servern ist als Vorgabezeit interpretierbar, wenn die Server-Geschwindigkeit gleich 1 ist. Ansonsten müsste die Arbeitsmenge noch mit dem Kehrwert der Servergeschwindigkeit multipliziert werden, um die Vorgabezeit zu erhalten.

HiSlang-Übersetzung:

Server-PKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 77, 202, 116, 277]**

```
TYPE _compname {MODEL|COMPONENT}[(_parameter)];
  ⋮
  TYPE _servname SERVICE [(_parameter)]
    [RESULT _ausgabetypen];
    USE SERVICE
      ⋮
      {*** Server-PKE: bezeichner (used Service) ***}
      bezeichner(amount:REAL);
      ⋮
    END USE;
    [VARIABLE
      {variable : typ} [, ...];]
  ⋮
  BEGIN
    {PKEs}
```

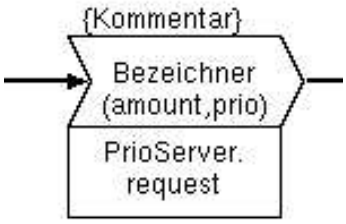
```
    ⋮
    {*** Server-PKE: bezeichner (Aufruf) ***}
    bezeichner(parameter);
    {Senke}
  END TYPE _servname;
  ⋮
  REFER _servname [, ...] TO server [, ...] EQUATING
    bezeichner WITH server.request;
    ⋮
  END REFER;
END TYPE _compname;
```

2.2.3.7 PrioServer-Prozesskettenelement

ProC/B-Bedeutung:

PrioServer-PKE (PrioServer-PKE):

Spezialform des Aufruf-PKEs. Dieses PKE ruft einen PrioServer auf. Der PrioServer kann auch von anderen PrioServer-PKEs genutzt werden.
Siehe **[B1, 43]**.



`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Parameter`	`: STRING → REAL`	*Arbeitsmenge (*`amount`*) als Ausdruck*
	`: STRING → REAL`	*Priorität (*`prio`*) als Ausdruck*
`Ausf.-Hinw.`	`: STRING`	`Server.request`

- Die Arbeitsmenge bei PrioServern ist als Vorgabezeit interpretierbar, wenn die Server-Geschwindigkeit gleich 1 ist. Ansonsten müsste die Arbeitsmenge noch mit dem Kehrwert der Servergeschwindigkeit multipliziert werden, um die Vorgabezeit zu erhalten.
- `prio` gibt die Priorität an, mit der die Arbeitsanforderung bedient wird. Siehe 2.1.3.

HiSlang-Übersetzung:

PrioServer-PKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 117, 289]**

```
TYPE _compname {MODEL | COMPONENT}[(_parameter)];
 ⋮
TYPE _servname SERVICE [(_parameter)]
  [RESULT _ausgabetypen];
  USE SERVICE
    ⋮
    {*** PrioServer-PKE: bezeichner (used Service) ***}
    bezeichner(amount:REAL, prio: INTEGER);
    ⋮
  END USE;
  [VARIABLE
    {variable : typ} [, ...];]
 ⋮
```

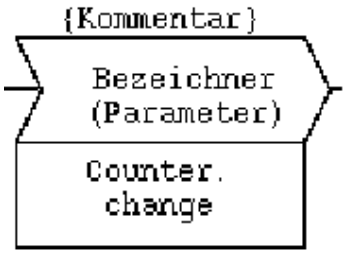
```
  BEGIN
    {PKEs}
    ⋮
    {*** PrioServer-PKE: bezeichner (Aufruf) ***}
    bezeichner(parameter);
    {Senke}
  END TYPE _servname;
  ⋮
  REFER _servname [, ...] TO prioserver [, ...] EQUATING
    bezeichner WITH prioserver.request;
    ⋮
  END REFER;
END TYPE _compname;
```

2.2.3.8 Counter-Prozesskettenelement

ProC/B-Bedeutung:

Counter-Prozesskettenelement (Counter-PKE):

Spezialform des Aufruf-PKE. Dieses PKE ruft einen Counter auf.
Siehe **verweisAufruf-PKE**.



`Kommentar`	`: STRING`	*s.o.*
`Bezeichner`	`: STRING`	*s.o.*
`Parameter`	`: INT_VEK`	*Veränderungsvektor des Counters*
`Ausf.-Hinw.`	`: STRING`	`Counter.change`

- Bei einem Counter kann ein Ausdruck angegeben werden.

HiSlang-Übersetzung:

Counter-PKE (HiSlang-Code mit Umgebung)
Siehe **[HIT, 77, 202, 116, 278]**

```
TYPE _compname {MODEL | COMPONENT}[(_parameter)];
  ⋮
TYPE _servname SERVICE [(_parameter)]
  [RESULT _ausgabetypen];
  USE SERVICE
    ⋮
    {*** Counter-PKE: bezeichner (used Service) ***}
    bezeichner(amount: ARRAY OF INTEGER;
      prio : INTEGER DEFAULT 32767);
    ⋮
  END USE;
  [VARIABLE
    {variable : typ} [, ...];]
⋮
BEGIN
  {PKEs}
  ⋮
```

```
    {*** Counter-PKE: bezeichner (Aufruf) ***}
    bezeichner(parameter);
    {Senke}
  END TYPE _servname;
  ⋮
  REFER _servname [, ...] TO counter [, ...] EQUATING
    bezeichner WITH counter.change;
    ⋮
  END REFER;
END TYPE _compname;
```

2.2.4 Senken

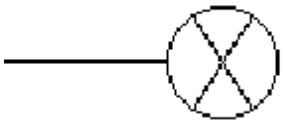
Senken spezifizieren allgemein Umstände der Beendigung von Prozessen. Unbedingte Senken beschreiben gemeinsam mit den unbedingten Quellen die Schnittstelle zwischen dem Modell und der (nicht spezifizierten) Umgebung des Modells, sie beschreiben also das Verhalten und die Charakteristika dieser Umgebung so weit wie erforderlich.

2.2.4.1 Bedingte Senke

ProC/B-Bedeutung:

Bedingte Senke (BedSenke):

Ein Dienst einer Funktionseinheit wird in Anspruch genommen, welcher einen FE-internen Prozess anstößt, wobei dieser Prozess innerhalb der FE endet, ohne eine diesbezügliche Antwort nach außen zu geben.
Siehe **[B1, 42]**.



HiSlang-Übersetzung:

BedSenke (HiSlang-Code mit Umgebung)
Siehe **—**

```
TYPE _compname {MODEL | COMPONENT}[(_parameter)];
  ⋮
  TYPE pkbezeichner SERVICE [(parameter)]
  BEGIN
    ⋮
    CREATE 1 PROCESS _bedSenkeService;
  END {SERVICE} TYPE _pkbezeichner

  TYPE _bedSenkeService SERVICE;
    USE
      SERVICE Aufruf(data : POINTER for voidRecordName);
    END USE;
  BEGIN
    Aufruf(parameter);
  END {SERVICE} TYPE _bedSenkeService;
  ⋮
  {*** Refer-Teil der FE ***} REFER
```

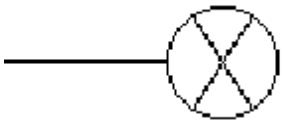
```
    _bedSenkeService
  TO
    aufzurufendeFE
  EQUATING
    _bedSenkeService.Aufruf
      WITH aufzurufendeFE.aufzurufendePK;
    END REFER;

END TYPE bezeichner;
```

2.2.4.2 Unbedingte Senke

ProC/B-Bedeutung:

Unbedingte Senke (UnbedSenke):

Standardelement um Prozessketten zu beenden. Eine unbedingte Senke beendet Prozesse, wenn sie im Ablauf der Prozesse erreicht wird.
Siehe **[B1, 63]**.



HiSlang-Übersetzung:

UnbedSenke (HiSlang-Code mit Umgebung)
Siehe **—**

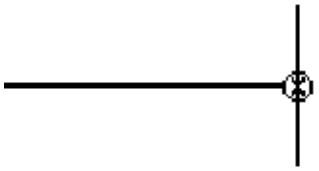
```
TYPE _name SERVICE; {vereinfacht}
BEGIN
  ⋮
  {PKEs}
  ⋮
  {*** Unbed. Senke: bezeichner ***}
END TYPE _name;
```

2.2.4.3 Virtuelle Senke

ProC/B-Bedeutung:

Virtuelle Senke (VirSenke):

Die virtuelle Senke bewirkt eine Rückmeldung der Beendigung interner Prozesse einer FE an die aufrufende Umgebung. Sie kann bedingte Quellen triggern.
Siehe **[B1, 31, 52, 64]**.



HiSlang-Übersetzung:

VirSenke (HiSlang-Code mit Umgebung)
Siehe **bedingte Quelle**

```
TYPE _compname COMPONENT;

⋮
  TYPE _compname SERVICE(data : POINTER FOR RECid_prozess);
    USE
      SERVICE bedQuelleAktivieren;
    END USE;

BEGIN
    NEW id_prozess POINTER data;

      {*** : VirSenke (id=id) ***} bedQuelleAktivieren;

END SERVICE TYPE _compname ;
⋮

{*** Refer-Teil der FE _fe ***} REFER
      P11ProzessID4
    TO
      Q19ProzessID7
    EQUATING
      prozess.bedQuelleAktivieren
        WITH prozess.setzen;
      END REFER;

BEGIN
```

```
    ⋮
    END COMPONENT TYPE F3Funktionseinheit2_Typ;
```

2.3 Quellen

Quellen spezifizieren allgemein die Umstände der Entstehung von Prozessen. Unbedingte Quellen beschreiben gemeinsam mit den unbedingten Senken die Schnittstelle zwischen dem spezifizierten Modell und der (nicht spezifizierten) Umgebung des Modells. Diese Quellen und Senken beschreiben also das Verhalten und die Charakteristika dieser Umgebung so weit wie erforderlich.

2.3.1 Quelle

ProC/B-Bedeutung:

Quelle (Quelle):

Eine Quelle erzeugt und parametrisiert Prozesse, die durch eine PK beschrieben werden. Prozesse können als Instanz einer PK angesehen werden. Man unterscheidet **unbedingte** (zeitgesteuerte) und **bedingte** (aus einer Funktionseinheit heraus gesteuerte) Quellen.
Siehe **[B1, 29, 41, 62], PK (2.2), virtuelle Quellen (2.3.4)**.

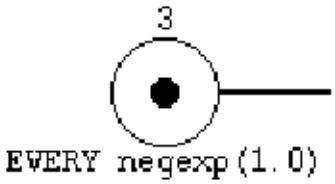
`Bezeichner : STRING` *ggf. als Kommentar im HiSlang-Code*

2.3.2 Unbedingte Quellen

ProC/B-Bedeutung:

Unbedingte Quelle (UnbedQuelle):

Unbedingte Quelle sind selbst aktiv und erzeugen zu festen Zeitpunkten oder mit einer gewissen Rate Prozesse. Siehe **[B1, 29, 41]**.



`Bezeichner`	`: STRING`	$\hat{=}$ *dem angeschlossenen PK-Namen*
`Anzahl`	`: STRING → INT`	*Anzahl gleichzeitig zu startender Proz.*
`Parameter`	`: STRING`	*Parameterbelegung der PK*
`Typ`	`: INT`	*Quellentyp (*`AT` *oder* `EVERY`*)*
`Zeitangabe`	`: STRING`	*Zeitpunktliste oder Zw.-Ankunftszeit*

- Für arithmetische Ausdrücke, die bei den Zeitangaben benutzt werden können, siehe [HIT, 27ff]

HiSlang-Übersetzung:

UnbedQuelle (HiSlang-Code mit Umgebung)
Siehe **[HIT, 84, 82, 208]**

```
TYPE _name {MODEL | COMPONENT}[(_parameter)];
⋮
 TYPE UnbedQuellen SERVICE;
 BEGIN
   CONCURRENT
     quellenblock
   TO
     quelleblock
     ⋮
   END CONCURRENT;
  END {SERVICE} TYPE UnbedQuellen;
END TYPE _name;
```

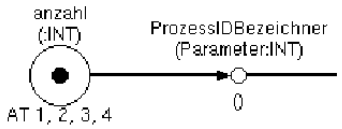
- *quellenblock*: Block mit At-Quelle (s. 2.3.2.1) oder Every-Quelle (s. 2.3.2.2).
- Mehrere Quellen werden durch `CONCURENT ... TO ... END CONCURRENT` parallel ausgeführt. Ist nur eine Quelle vorhanden, fehlt diese Concurrent-Anweisung.

2.3.2.1 At-Quelle

ProC/B-Bedeutung:

At-Quelle (AtQuelle):

Eine At-Quelle erzeugt zu den angegebenen Zeitpunkten Prozesse.
Siehe **[B1, 29, 41]**.



`Bezeichner`	`: STRING`	$\hat{=}$ *dem angeschlossenen PK-Namen*
`Anzahl`	`: STRING → INT`	*Anzahl gleichzeitig zu startender Proz.*
`Parameter`	`: STRING`	*Parameterbelegung der PK*
`Typ`	`: INT`	*Quellentyp (*`AT`*)*
`Zeitangabe`	`: STRING`	*Zeitpunktliste*

- Die Zeitpunkte sind in einer durch Kommata getrennten Liste in aufsteigender Reihenfolge anzugeben. Die Zeitpunktangaben werden nacheinander ausgewertet, und zwar zur Modellzeit des jeweils vorhergehenden Zeitpunkts. Ausgewertet wird also der i-te Zeitpunkt in der Liste, wenn die Modellzeit des Zeitpunkts i-1 erreicht wird.

  Ist bei der Auswrtung eines Zeitpunkts die entsprechende Modellzeit bereits abgelaufen, so wird die gegenwärtige Modellzeit genommen. `AT(2,1)` bedeutet demnach, dass zum Zeitpunkt `2` je ein Prozess für `AT 1` und `AT 2` erzeugt wird.

- Die Anzahl der zu erzeugenden Prozesse wird in dem Moment der Modellzeit ausgewertet, zu dem die Prozesse erzeugt werden. Dies spielt dann eine Rolle, wenn als Anzahl nicht eine feste Zahl eingegeben wird, sondern eine Variable, deren Wert sich im Laufe der Modellzeit andert.

- Für jeden einzelnen Prozess werden die (optionalen) Parameter getrennt ausgewertet, und zwar zum Zeitpunkt der Prozesserzeugung.

HiSlang-Übersetzung:

At-Quelle (HiSlang-Code mit Umgebung)
Siehe **[HIT, 84, 82, 208]**

```
BLOCK
  VARIABLE
    hitLoopSchleifenZaehler : INTEGER;
    hitLoopSchleifenProzessanzahl : INTEGER;
    hitAtZaehler : INTEGER;
    hitQuellenZeitpunktAnzahl : INTEGER;
BEGIN
  hitQuellenZeitpunktAnzahl :=zeitpunktanzahl;
```

```
  BLOCK
    VARIABLE
      hitQuellenZeitpunkt :
        ARRAY [0..hitQuellenZeitpunktAnzahl] OF INTEGER;
      hitQuellenAbstand : INTEGER;
  BEGIN
    hitQuellenZeitpunkt[0] := 0;
      {***Wert dient zur Berechnung von erstem
Zeitpunkt***|}.[14]
    hitQuellenZeitpunkt[1] := zeitpunkt1;
    hitQuellenZeitpunkt[2] := zeitpunkt2;
    ⋮
    hitQuellenZeitpunkt[zeitpunktanzahl] := letzterZeitpunkt;
    LOOP
      hitQuellenAbstand :=
        hitQuellenZeitpunkt[hitAtZaehler]
        - hitQuellenZeitpunkt[hitAtZaehler - 1];
      IF hitQuellenAbstand < 0
      THEN
        hitQuellenAbstand := 0;
        hitQuellenZeitpunkt[hitAtZaehler] :=
          hitQuellenZeitpunkt[hitAtZaehler -1];
      END IF;
      hold (hitQuellenAbstand);
      hitLoopSchleifenZaehler := 0;
      hitLoopSchleifenProzessanzahl := anzahl;
      WHILE hitLoopSchleifenZaehler
        < hitLoopSchleifenProzessanzahl
      LOOP
        CREATE 1 PROCESS prozessbezeichner
          (NONE,
          parameter);
        hitLoopSchleifenZaehler :=
          hitLoopSchleifenZaehler +1;
      END LOOP;
      hitAtZaehler:= hitAtZaehler + 1;
    END LOOP UNTIL hitAtZaehler =
      hitQuellenZeitpunktanzahl + 1;
  END BLOCK;
END BLOCK;
```

[14] Zum Zeitpunkt `hitQuellenZeitpunkt[0]` wird kein Prozess gestartet. `hitQuellenZeitpunkt[0]` dient zur Berechnung von `hitQuellenAbstand`

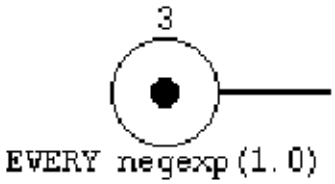
- Der Parameter *anzahl* gibt an, wie viele Prozesse gleichzeitig gestartet werden. Es ist nur ein ganzzahliger Wert > 0 sinnvoll (obwohl andere Werte gerundet werden).
- *parameter* bezeichnet die Parameterbelegung der prozesskette.
- *zeitpunktanzahl* ist die Anzahl der Werte in der Zeiptunktliste `Zeitangabe`.
- Die einzelnen Zeitpunkte der *zeitangabe* werden mit *zeitpunkt*1, *zeitpunkt*2 bis *letzter-Zeitpunkt* bezeichnet.
- In der `Zeitangabe` können auch Variablen benutzt werden.
- *parameter* bezeichnet die Parameterbelegung der prozesskette. Wird kein Parameter angegeben, so lautet der Aufruf `CREATE 1 PROCESS` *prozessbezeichner*`(NONE);`

2.3.2.2 Every-Quelle

ProC/B-Bedeutung:

Every-Quelle (EveryQuelle):

Eine At-Quelle erzeugt mit einer gewissen RateProzesse.
Siehe **[B1, 29, 41]**.



`Bezeichner`	`: STRING`	$\hat{=}$ *dem angeschlossenen PK-Namen*
`Anzahl`	`: STRING` $\rightarrow$ `INT`	*Anzahl gleichzeitig zu startender Proz.*
`Parameter`	`: STRING`	*Parameterbelegung der PK*
`Typ`	`: INT`	*Quellentyp (*`EVERY`*)*
`Zeitangabe`	`: STRING`	*Zwischenankunftszeit*

- Der Kehrwert der **Ankunftsrate** (Objekte pro Zeiteinheit) ist die mittlere **Zwischenankunftszeit** (Zeit zwischen zwei Prozesserzeugungen).
- Bei der `Zeitangabe` wird angegeben, in welchen Abständen (sog. Zwischenankunftszeit) ein Prozess erzeugt wird (z.B. ... `EVERY negexp(1/10)`).
- Die Anzahl der zu erzeugenden Prozesse wird in dem Moment der Modellzeit ausgewertet, zu dem die Prozesse erzeugt werden. Dies spielt dann eine Rolle, wenn als Anzahl nicht eine feste Zahl eingegeben wird, sondern eine Variable, deren Wert sich im Laufe der Modellzeit ändert. Es ist nur ein ganzzahliger Wert > 0 sinnvoll, andere Werte werden gerundet.
- Für jeden einzelnen Prozess werden die (optionalen) Parameter getrennt ausgewertet, und zwar zum Zeitpunkt der Prozesserzeugung.

HiSlang-Übersetzung:

Every-Quelle (HiSlang-Code mit Umgebung)
Siehe **[HIT, 84, 82, 208]**

```
LOOP
  BLOCK
    VARIABLE
      hitLoopSchleifenZaehler : INTEGER;
      hitLoopSchleifenProzessanzahl : INTEGER;
  BEGIN
    hitLoopSchleifenZaehler := 0;
    hitLoopSchleifenProzessanzahl := anzahl;
```

```
    WHILE hitLoopSchleifenZaehler
      < hitLoopSchleifenProzessanzahl
    LOOP
      CREATE 1 PROCESS prozessbezeichner(NONE,
                       parameter);
      hitLoopSchleifenZaehler := hitLoopSchleifenZaehler +1;
    END LOOP;
  END BLOCK;
  hold(zeitangabe);
END LOOP;
```

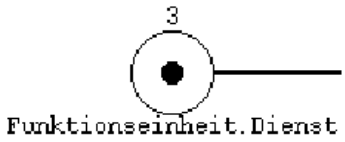
- In der *zeitangabe* können auch Variablen benutzt werden. Die Zwischenankunftszeit wird nach jeder Prozesserzeugung aktualisiert.
- *parameter* bezeichnet die Parameterbelegung der prozesskette. Wird kein Parameter angegeben, so lautet der Aufruf `CREATE 1 PROCESS` *prozessbezeichner*`(NONE);`

2.3.3 Bedingte Quelle

ProC/B-Bedeutung:

Bedingte Quelle (BedQuelle):

Eine Bedingte Quelle instanziiert Prozesse ihrer PK abhängig von der Beendigung eines Dienstes einer FE. Dementsprechend enthält die Beschreibung einer Bedingten Quelle den Namen einer FE und eines ihrer Dienste. Siehe **[B1, 62]**.



`Bezeichner`	`: STRING`	*ggf. als Kommentar im HiSlang-Code*
`Anzahl`	`: STRING → INT`	*Anzahl zu startender Proz.*
`Parameter`	`: STRING`	*Parameterbelegung der PK*
`Funktionseinheit`	`: STRING`	*Bedingende FE*
`Dienst`	`: STRING`	*Bedingender Dienst der FE*

- Der Dienst, der die Quelle bedingt, muß vom Typ `out` sein. Sein Aufruf muß also in der FE geschehen. Ein externer Aufruf des Dienstes würde eine Rückmeldung an die aufrufende Stelle statt an die bedingte Quelle erfordern.

HiSlang-Übersetzung:

BedQuelle (HiSlang-Code mit Umgebung)
Siehe **virtuelle Senke**

```
TYPE _name {MODEL|COMPONENT}[(_parameter)];
  ⋮
  {*** (._name) : Hilfskomponente_fuer_BedQuelle ***}
  TYPE _name_Typ COMPONENT;
    PROVIDE
      SERVICE setzen;
      SERVICE auslesen;
    END PROVIDE;
    COMPONENT sem : semaphor(0);
    TYPE setzen SERVICE;
      USE SERVICE belegen; END USE;
    BEGIN
      belegen;
    END SERVICE TYPE setzen;
    TYPE auslesen SERVICE;
      USE SERVICE freigeben; END USE;
    BEGIN
```

```
    freigeben;
  END SERVICE TYPE auslesen;
  REFER setzen, auslesen TO sem
    EQUATING setzen.belegen WITH sem.v;
             auslesen.freigeben WITH sem.p;
  END REFER;
 END COMPONENT TYPE _name;
⋮

{*** Alle BedQuellen der FE ***}
 TYPE bedQuellen SERVICE;
  USE
    SERVICE auslesen1;
  END USE;
 VARIABLE
  hitLoopSchleifenZaehler : INTEGER;
  hitLoopSchleifenProzessanzahl : INTEGER;
 BEGIN
  {*** (bezeichner) : BedQuelle (id=id) ***} LOOP
    auslesen1;
    BLOCK
      VARIABLE
        hitLoopSchleifenZaehler : INTEGER;
        hitLoopSchleifenProzessanzahl : INTEGER;
      BEGIN
        hitLoopSchleifenZaehler := 0;
        hitLoopSchleifenProzessanzahl := 1;
        WHILE hitLoopSchleifenZaehler
           < hitLoopSchleifenProzessanzahl
        LOOP
          CREATE 1 PROCESS P23ProzessID7(NONE);
          hitLoopSchleifenZaehler :=
             hitLoopSchleifenZaehler +1;
        END LOOP;
      END BLOCK;
    END LOOP;
  END SERVICE TYPE bedQuellen;

  {*** Refer-Teil der FE ***}
  REFER
    bedQuellen
  TO
    prozess
  EQUATING
    bedQuellen.auslesen1
      WITH prozess.auslesen;
```

```
   END REFER;

BEGIN
   ⋮
   {*** Aktivierung des Bedingte-Quellen-Service ***}
   CREATE 1 PROCESS bedQuellen;
   ⋮
END {MODEL|COMPONENT} TYPE _name;
```

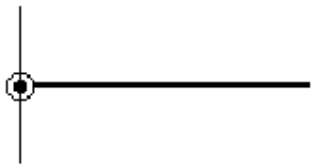
- Mit *prozess* wird die PK bezeichnet, die die bedingte Quelle triggert.

2.3.4 Virtuelle Quelle

ProC/B-Bedeutung:

Virtuelle Quelle (VirQuelle):

Eine virtuelle Quelle ist einem Dienst umgebenden FE zugeordnet, dessen Aufrufe und Aufrufsparameter für Instanziierung und Parametrisierung von Prozessen der PK zuständig sind. Der Dienstname entspricht dem Namen der PK.
Siehe **[B1, 52]**.



`Bezeichner : STRING` *ggf. als Kommentar im HiSlang-Code*

- Virtuelle Quellen erzeugen bei einem Aufruf des Dienstes genau eine Instanz des Prozesses.

HiSlang-Übersetzung:

VirQuelle (HiSlang-Code mit Umgebung)
Siehe

```
TYPE _name_Typ COMPONENT;
  PROVIDE
    SERVICE pk(data : POINTER FOR RECid_pk;
  END PROVIDE;

  TYPE RECid_pk;
  ⋮
  END {RECORD} TYPE pk;
  ⋮
END COMPONENT TYPE _name;
```

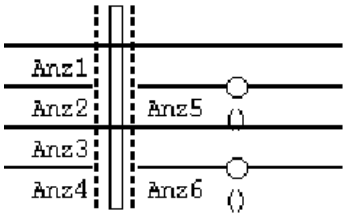
2.4 Prozesskettenkonnektoren

2.4.1 PK-Konnektor

ProC/B-Bedeutung:

PK-Konnektor (PKKonnektor):

Prozesskettenkonnektoren synchronisieren Anfänge und Enden von einfachen[15] Prozessketten. Man unterscheidet zwischen der öffnenden und schließenden Variante und dem **gemischten** PK-Konnektor. Der Editor sieht nur den gemischten PK-Konnektor vor, er kann sowohl als öffnender als auch als schließender PK-Konnektor benutzt werden.
Siehe **[B1, 59–62]**.



- Prozesse können am PK-Konnektor fortgeführt werden (graphisch **durchgezogen**), d.h. sie synchronisieren sich lediglich mit den anderen Prozessen. Nach der Synchronisation wird der Prozess mit den alten Parametern weiter fortgesetzt. Die zugehörige Prozesskette geht über den PK-Konnektor hinweg.
- Prozesse können aber auch am Konnektor enden und andere Prozesse, nach der Synchronisation mit anderen Parametern starten. D.h. eine Prozesskette endet in diesem Fall.
- Es gibt auf eingehenden Kanten Anzahlen, die ausdrücken, wie viele Prozesse zur Synchronisation ankommen müssen. Die Anzahl am Ausgang entspricht der bei den Quellen und gibt die Anzahl zu startender Prozesse an.

  Bei durchgezogenen Kanten entspricht die Anzahl n, der Anzahl bei eingehenden Kanten. Da alle n zu synchronisierenden Prozesse nach der Synchronisation weiterlaufen, ist eine Angabe einer Anzahl an der durchgezogenen ausgehenden Kante nicht erlaubt.
- Semantisch enthält der gemischte Konnektor die öffnende und schliessende Variante.

HiSlang-Übersetzung:

PKKonnektor (HiSlang-Code mit Umgebung)
Siehe —

```
TYPE _name {MODEL|COMPONENT}[(_parameter)];
  ⋮
```

[15] Eine einfache PK kann aus PKEs, Oder- und Und-Konnektoren bestehen und hat genau einen Anfang und ein Ende.

```
COMPONENT PKKONAnkunft id : counter
  (LET MIN := minVektor,
  LET MAX := maxVektor,
  LET INIT := minVektor);
COMPONENT PKKONSynchro id : counter
  (LET MIN := minVektor,
  LET MAX := maxVektor,
  LET INIT := minVektor);

ankommendePK1
ankommendePK2
⋮
ankommendePKn

startendePK1
startendePK2
⋮
startendePKm

{*** Alle PKKonnektoren der FE ***}
TYPE pkKonnektoren SERVICE;
  USE
    SERVICE ankunft1(amount : ARRAY OF INTEGER;
            prio : INTEGER DEFAULT 32767);
    SERVICE synchro1(amount : ARRAY OF INTEGER;
            prio : INTEGER DEFAULT 32767);
  END USE;
BEGIN
  LOOP
    ankunft1(ankunftsvektor);
    synchro1(synchrovektor);
    CREATE startanzahl1 PROCESS startendePK1prozess(parameter);
    CREATE startanzahl2 PROCESS startendePKmprozess(parameter);
    ⋮
    CREATE startanzahlm PROCESS startendePKmprozess(parameter);
  END LOOP;
END SERVICE TYPE pkKonnektoren;
⋮

{*** Refer-Teil der FE ***}
REFER
  ankommendePK1prozess,
  ankommendePK2prozess,
  ⋮
```

```
 ankommendePKnprozess,
 pkKonnektoren
TO
 PKKONAnkunftid,
 PKKONSynchroid
   EQUATING
    ankommendePK1prozess.PKKONAnkunftid
      WITH PKKONAnkunftid.change;
    ankommendePK1prozess.PKKONSynchroid
      WITH PKKONSynchroid.change;
    ankommendePK2prozess.PKKONAnkunftid
      WITH PKKONAnkunftid.change;
    ankommendePK2prozess.PKKONSynchroid
      WITH PKKONSynchroid.change;
    ⋮
    ankommendePKnprozess.PKKONAnkunftid
      WITH PKKONAnkunftid.change;
    ankommendePKnprozess.PKKONSynchroid
      WITH PKKONSynchroid.change;
    pkKonnektoren.ankunftid
      WITH PKKONAnkunftid.change;
    pkKonnektoren.synchroid
      WITH PKKONSynchroid.change;
  END REFER;

 BEGIN
   ⋮
   {*** Aktivierung des PK-Konnektoren-Service ***}
   CREATE 1 PROCESS pkKonnektoren;
   ⋮
  END TYPE _name;
```

- n sei die Anzahl der am PK-Konnektor ankommende, m die der abgehende Prozessketten.
- *minvektor* ist ein n-stelliger Vektor mit den Werten 0 und *maxvektor* der entsprechende Vektor mit den Werten 32767, also z. B. $[0, 0, 0]$ bzw. $[32767, 32767, 32767]$ für drei ankommende lineare Prozessketten.
- *ankunftsvektor* ist ein n-stelliger Vektor mit den Werten *-anzahl*, *synchrovektor* ein n-stelliger Vektor mit den Werten *anzahl*, wobei *anzahl* die Anzahl auf den ankommenden Prozessketten ist.

- *ankommendePK*1 bis*ankommendePK*n bezeichnen die den an dem Prozesskettenkonnektor ankommenden Prozessketten zugehörigen Codeblöcke. In der `USE SERVICE`-Deklarationen dieser Prozessketten befindet sich folgender Code:

  – `USE`
  ```
  SERVICE PKKONAnkunft<id>(amount: ARRAY OF INTEGER);
          prio: INTEGER DEFAULT 32767);
  SERVICE PKKONSynchro<id>(amount: ARRAY OF INTEGER);
          prio: INTEGER DEFAULT 32767);
  ⋮
END USE
```

- Im Hauptcodeblock dieser Prozessketten werden diese Services benutzt:

  – `PKKONAnkunft`*id*`(pkankunftsvektor);`
  `PKKONSynchro`*id*`(pksynchrovektor);`

  Dabei bezeichnet *pkankunftsvektor* einen n-stelligen Vektor, bei dem für die i-te ankommende Prozesskette die i-te position 1 ist, alle übrigen Positionen sind 0. Entsprechend ist *pksynchrovektor* der Vektor, in dem die i-te Position -1 ist.

- *startendePK*1 bis *startendePK*n bezeichnen die den an dem Prozesskettenkonnektor startenden Prozessketten zugehörigen Codeblöcke.

- *startendePK*1*prozess* bis *startendePK*m*prozess* bezeichnen die zu den startenden Prozessketten gehörenden Typbezeichner.

- *startanzahl*1 bis *startanzahl*m bezeichnen die Anzahlen der in den startenden Prozessketten 1 bis m zu startenden Prozesse.

- Werden die Prozesse einer Prozesskette fortgeführt, so existiert dafür keine `startendePK`. Der Code der zu dieser Prozesskette gehörenden Prozesskettenelemente befindet sich unter dem der Benutzung von `PKKONAnkunft`*id* und `PKKONSynchro`*id*.

2.5 Sonstiges

2.5.1 Globale Variablen

ProC/B-Bedeutung:

Globale Variablen (Variablen):

Im Element Globale Variablen wird eine Liste von Variablen verwaltet.
Siehe —.

Bezeichner:STRING
Bezeichner2:REAL=1
{Kommentar}

- Globale Variablen sind in dem Modellteil bzw. der Funktionseinheit, in dem sie definiert wurden, sowie in allen darin enthaltenen Funktionseinheiten sichtbar.

HiSlang-Übersetzung:

Globale Variablen (HiSlang-Code mit Umgebung)
Siehe —

```
VARIABLE
  bezeichner : TYP DEFAULT Defaultwert;
  [
  bezeichner : TYP DEFAULT Defaultwert;
  ⋮
  bezeichner : TYP DEFAULT Defaultwert;
  ]
```

- Mögliche Typen: STRING, INT, REAL, BOOL

2.5.2 Rewards

ProC/B-Bedeutung:

Rewards (Rewardslabel):

Im Element Rewards wird eine Liste von selbstdefinierten Rewards (Streams in HiSlang) verwaltet.
Siehe —.

```
REWARD eventreward: EVENT
REWARD statereward: STATE
REWARD countreward: COUNT
```

- Rewards sind in dem Modellteil bzw. der Funktionseinheit, in dem sie definiert wurden, sowie in allen darin enthaltenen Funktionseinheiten sichtbar.

HiSlang-Übersetzung:

Globale Variablen1 (HiSlang-Code mit Umgebung)
Siehe —

```
STREAM
  bezeichner : TYP;
  [
  bezeichner : TYP;
  ⋮
  bezeichner : TYP;
  ]
```

- Als Typen stehen EVENT, COUNT sowie STATE zur Verfügung.

Literaturverzeichnis

[B1] H. Beilner, F. Bause, H. Tatlitürk, A. van Almsick, M. Völker: Zum B-Modellformalismus, Version B1, Bericht Nr. 99002, siehe A.

[HIT] HiSlang-Reference-Manual (Version 3.6.00 von 99), www4.cs.uni-dortmund.de/HIT/HITDOCUPDF/HISLANG-refman.pdf.

Die in diesem Bericht benutzten verweisenden Seitenzahlen beziehen sich auf die Seitennummern des jeweiligen Dokumentes, nicht auf die Seitennummern in diesem Bericht.

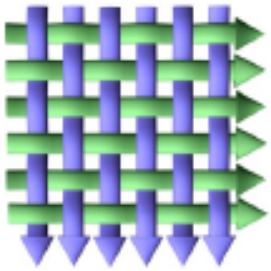
Anhang A

ProC/B-Modellformalismus (Vers. 1)

Dieses Kapitel enthält die ursprüngliche informale Definition des ProC/-Paradigmas in seiner ersten Version.

SFB 559 - Teilprojekt M1
LS Informatik IV
Universität Dortmund
02.06.1999

Sonderforschungsbereich 559
Modellierung großer
Netze in der Logistik



Bericht Nr. 99002

Zum B-Modellformalismus
- Version B1 -

zur Vorbereitung automatisierter Analysen
von Modellen logistischer Systeme hinsichtlich
technischer, ökonomischer und ökologischer Ziele

SFB 559 - Teilprojekt M1

H. Beilner, F. Bause, H. Tatlitürk,
A. van Almsick, M. Völker

Inhaltsverzeichnis

1 Einleitung

Dieser Bericht beinhaltet einen Vorschlag des Methodenteilprojekts M1 (TP M1: Strukturierte GNL-Modelle und effiziente Simulation) zur Konkretisierung einer Teilmenge des Prozeßkettenparadigmas. Diese Teilmenge wird im folgenden mit **B-Paradigma** bezeichnet, um sie von dem „umfassenden" (vollständigen) Prozeßkettenparadigma unterscheiden zu können, welches in diversen Veröffentlichungen auftritt und welches in diesem Dokument mit **A-Paradigma** bezeichnet werden soll. Der Bericht beschreibt insbesondere eine erste Version des B-Paradigmas, das B1-Paradigma. Hierbei wird unter dem Begriff „Paradigma" der Aspekt eines beschreibenden Modellformalismus wesentlich betont und daher einschränkenderweise auch von B- bzw. **B1-Modellformalismus** gesprochen.

Für das Methodenteilprojekt M1 ist eine präzise Definition der Modellierungskonstrukte und deren Semantik notwendig, da andernfalls eine maschinengestützte Analyse, z. B. mittels Simulation oder anderer Verfahren (vgl. TP M2: Effiziente Analyseverfahren) nicht möglich ist. Es ist (für spätere Phasen des SFBs) beabsichtigt, eine Modellbeschreibung im B-Modellformalismus direkt analysieren zu können, d. h. Modellanalysen, ohne händische Umsetzung der Modellbeschreibung in eine andere Beschreibungsform, vornehmen zu können.

Ausgangspunkt für unsere Überlegungen sind logistische Systeme und zugehörige Beurteilungs- und Bewertungs-Fragen, wie sie innerhalb der Anwendungsprojekte des SFB 559 auftreten. In diesem Sinne wurde versucht, in bilateralen Kontakten typische Anwendungsbeispiele zu erhalten, und aus ihnen die Beschreibungsmöglichkeiten des B1-Formalismus zu entwickeln. Als globale Leitlinien der Entwicklung galten durchgehend einerseits die existierenden Beschreibungsformen des A-Paradigmas, andererseits die im Vorgängerabschnitt motivierte Notwendigkeit einer präzisen Beschreibung.

Unsere Vorgehensweise spiegelt sich auch in der Struktur dieses Dokuments wider. Kapitel 2 stellt eine Auswahl allgemeiner Aspekte des Prozeßkettenparadigmas vor, wie sie in der Literatur, insbesondere in /KLÖP91/, geschildert werden. Diese Sichtweise ist Ausgangspunkt für die weiteren Betrachtungen. Kapitel 3 skizziert einige typische Anwendungsbeispiele. Deren Beschreibung erfolgt oft in Prosa oder mit (noch nicht präzise definierten) prozeßkettenorientierten Darstellungen. Wie oben angesprochen, sollen mittels dieser Anwendungsbeispiele Anforderungen an die Beschreibungsmöglichkeiten des zu definierenden B1-Formalismus identifiziert werden. In Kapitel 4 präsentieren wir einen Vorschlag für eine solche Definition und zeigen in dem nachfolgenden Kapitel 5 auf, wie die ausgewählten Anwendungsbeispiele derart modelliert werden könnten, daß eine maschinengestützte Analyse möglich ist. Eine Auswahl erzielbarer Analyseergebnisse wird in Kapitel 6 im Kontext der Anwendungsbeispiele aufgezeigt.

Die vorgestellten Beispiele entstanden in mehreren Gesprächen mit unseren Projektpartnern im Zeitraum Oktober 1998 bis Februar 1999. Herrn Möller (TP A4: Netze und GVZ) verdanken wir das interessante Beispiel über Güterverteilzentren. Bei unseren Überlegungen im Bereich Distribution unterstützte uns Herr Dr. Schürholz vom IML. Herr Krabs (TP A11: Redistributionsnetze) stellte uns ein Beispiel aus dem Bereich der Redistributionsnetze zur Verfügung und Herr Schmitz (TP A2: Beschaffung) steuerte das Beispiel aus dem Bereich der Beschaffungskanäle bei. Bei allen genannten Personen möchten wir uns recht herzlich für ihre Mitarbeit und Unterstützung bedanken. Ohne sie würde der B1-Formalismus sicherlich entscheidende Defizite aufweisen, zusätzlich zu den höchstwahrscheinlich ohnehin verbliebenen. Ferner gebührt unser Dank Herrn Prof. Dr.-Ing. Kuhn und Herrn Manthey sowie ihren Mitarbeitern vom IML, die uns bei allgemeinen Fragen zum Prozeßkettenparadigma, sowie zu speziellen Fragen zum Tool LogiChain hilfreich zur Seite standen.

2 Prozeßketten

Im folgenden werden einige Aspekte des A-Paradigmas beschrieben, welche uns bei der Definition des B1-Formalismus geleitet haben.

Prozeßketten beschreiben einen sogenannten Transformationsprozeß (vgl. /KLÖP91/). Ein Leistungsobjekt, welches die Prozeßkette durchläuft, erfährt dabei eine Wertschöpfung. Zur Erzielung dieser Wertschöpfung werden Ressourcen benötigt. Der Ablauf der Wertschöpfung wird durch den Prozeßkettenplan beschrieben, der im wesentlichen die Abfolge einzelner Prozeßkettenelemente (PK-Elemente) zeigt. Der Transformationsprozeß an einem (oder mehreren) Leistungsobjekten unterliegt Restriktionen, die sich in Form von Lenkungsregeln und Strukturen widerspiegeln, wobei die Strukturen das mögliche Lenkungsverhalten bestimmen /BECK96/. Wichtig ist festzuhalten, daß eine Prozeßkette immer den Ablauf eines(!) Leistungsobjektes beschreibt. Bei der Betrachtung der gesamten Modelldynamik können mehrere Leistungsobjekte gleichzeitig existieren, welche nebenläufig die einzelnen „Abschnitte“ ihrer(!) Prozeßkette durchlaufen. Hierbei kommt es zu Wechselwirkungen, z. B. durch die Nutzung gemeinsamer Ressourcen oder die Notwendigkeit, sich mit anderen Prozessen zu synchronisieren.

Bezüglich der Lenkung lassen sich lt. /KLÖP91/ mehrere Ebenen unterscheiden: die strategische und die taktische Ebene, welche lang- bzw. mittelfristige Zeithorizonte erfassen und die operative Ebene, die sich vornehmlich mit dem „Tagesgeschäft“ befaßt. Da uns derzeit keine Informationen/Regeln vorliegen, wie sich Entscheidungen der strategischen und taktischen Ebene auf die automatisierte Prozeßkettenbewertung auswirken, werden im folgenden nur Einflüsse der taktischen Ebene berücksichtigt, also die Lenkungsregeln des Prozesses und des Netzwerkes (vgl. /BECK96/).

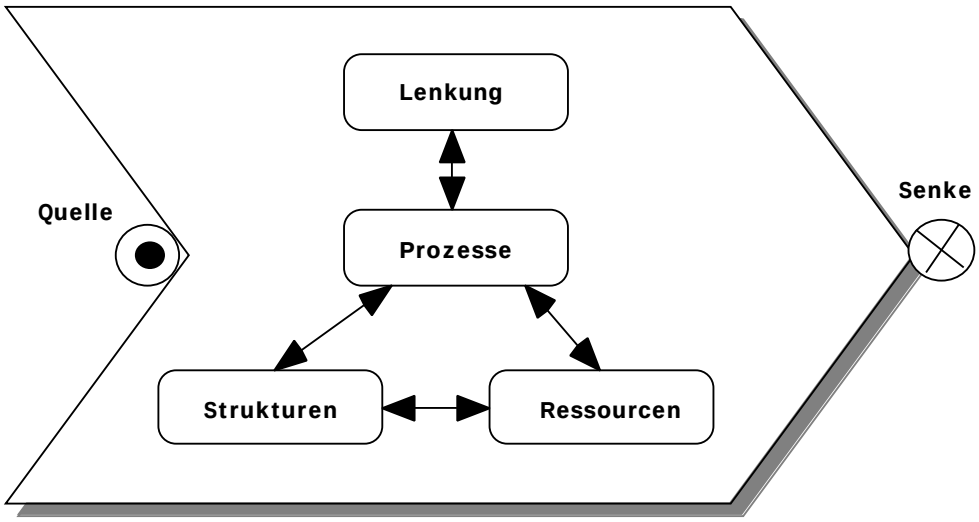


Abbildung 2-1: Graphische Darstellung eines Prozeßkettenelements

Die wesentlichen beschreibenden Elemente einer Prozeßkette bzw. eines Prozeßkettenelements werden in sogenannten Potentialklassen erfaßt, welche wiederum 4 Kategorien zuzuordnen sind /KUHN95/, /WiQu97/:

- Lenkungsebenen: Normative Ebene, Administration, Dispositive Ebene, Netzwerk, Steuerung.
  Zur taktischen Ebene zählen wir Netzwerk und Steuerung.
- Strukturen: Layout, Aufbauorganisation, technische Kommunikationsstruktur.
- Ressourcen: Personal, Flächen, Bestand, Arbeitsmittel, Hilfsmittel, Organisationsmittel.

· Prozesse: Quellen, Senken, Prozeßstrukturen.

Diese Potentialklassen haben eine checklistenartige Funktion bei der Erfassung der Daten und späteren Modellierung eines logistischen Systems. Graphisch werden diese Potentialklassen mittels des Symbols in Abbildung 2-1 repräsentiert, welches auch zur Darstellung eines Prozeßkettenelements genutzt wird.

2.1 Prozeßkettenelemente

Nach dem ursprünglichen Konzept der Wertkette nach Porter (/PORT86/) beschreiben die Elemente einer Prozeßkette Aktivitäten, die ein z. B. von einer Quelle ankommendes Leistungsobjekt verändern. Das so veränderte Leistungsobjekt verläßt die Prozeßkette an einer Senke bzw. betritt ein eventuell nachfolgendes Prozeßkettenelement. Jedes Prozeßkettenelement (PK-Element) wird wiederum durch Daten aus den o. g. Potentialklassen beschrieben. Hierbei besteht die Möglichkeit, die Aktivität des PK-Elements wiederum durch eine Prozeßkette verfeinert darzustellen, wodurch die gesamte Modellbeschreibung eine hierarchische Struktur erhält.

2.2 Prozeßkette

Eine Prozeßkette beschreibt die (zeitliche) Reihenfolge der einzelnen Aktivitäten eines(!) Prozesses, welche teilweise in kausaler Beziehung stehen, durch eine sequentielle Anordnung der Prozeßkettenelemente von links nach rechts. Ferner hält die Kette alternative Aktivitätsfolgen des Prozesses fest, sowie Bedingungen für das Fortschreiten. Eine Prozeßkette kann auch das Aufspalten eines Prozesses in mehrere nebenläufige (parallele) Aktivitätsfolgen enthalten, die sich nach erfolgter „Abarbeitung“ wieder zu einem Prozeß synchronisieren. Ein Beispiel für eine solche Prozeßkette illustriert Abbildung 2-2, wobei die Darstellung der Verbindungselemente willkürlich gewählt wurde. Die Beschreibung ein oder mehrerer Prozeßketten erfolgt mittels des sogenannten Prozeßkettenplans.

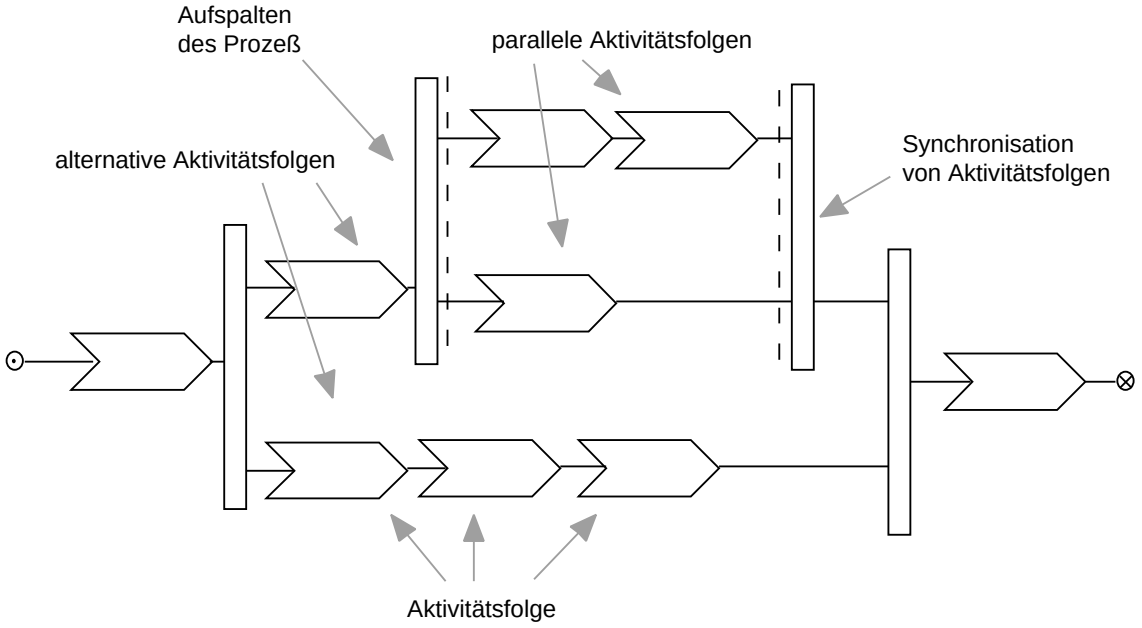


Abbildung 2-2: Beispiel einer Prozeßkette (einfacher Prozeßkettenplan)

2.3 Prozeßkettenplan

Ein Prozeßkettenplan beschreibt ein oder mehrere Prozeßketten und stellt Abhängigkeiten zwischen diesen dar. Alle Aktivitäten, welche durch Prozeßketten-Elemente dargestellt werden, werden im sogenannten Prozeßkettenplan in eine zeitliche Anordnung gebracht. Laut /KLÖP91/ ist die Zeit der

entscheidende Erfolgsfaktor in der Logistik und steht daher als Strukturierungs- und Anordnungskriterium im Vordergrund. Diese Form der Strukturierung impliziert, daß die Verbindung der Prozeßketten-Elemente keine Schleifen enthält.

Der Prozeßkettenplan beschreibt neben den einzelnen Prozeßketten insbesondere (kausale) Abhängigkeiten zwischen diesen Prozeßketten, z. B. durch Angabe von Synchronisationspunkten. Als diesbezügliche Beschreibungsmittel werden sogenannte Konnektoren verwendet, welche die einzelnen PK-Elemente miteinander verbinden. Grob betrachtet stehen 3 Verbindungstypen zur Verfügung (vgl. /LOGI/):

- Ablauflogische Verbindungen. Sie legen die Reihenfolge der einzelnen Aktivitäten fest, die an einem bzw. durch ein Leistungsobjekt durchgeführt werden.
- Konnektoren zur Auswahl alternativer Aktivitätsreihenfolgen (*Flußkonnektoren*).
- Zeitkonnektoren. Sie werden über ablauflogische Verbindungen gelegt und geben an, daß die Weitergabe von Leistungsobjekten über alle so verknüpften ablauflogischen Verbindungen zeitgleich zu geschehen hat. Hierdurch wird eine Synchronisation von Leistungsobjekten erreicht. Dies erfolgt z. B. häufig bei der Modellierung von Informations- und Materialflüssen, beispielsweise wenn ein (separat modellierter) Lieferschein am (ebenfalls separat modellierten) zugehörigen Paket angebracht ist.

Abbildung 2-3 zeigt einen Prozeßkettenplan, welcher obige Verbindungstypen illustriert.

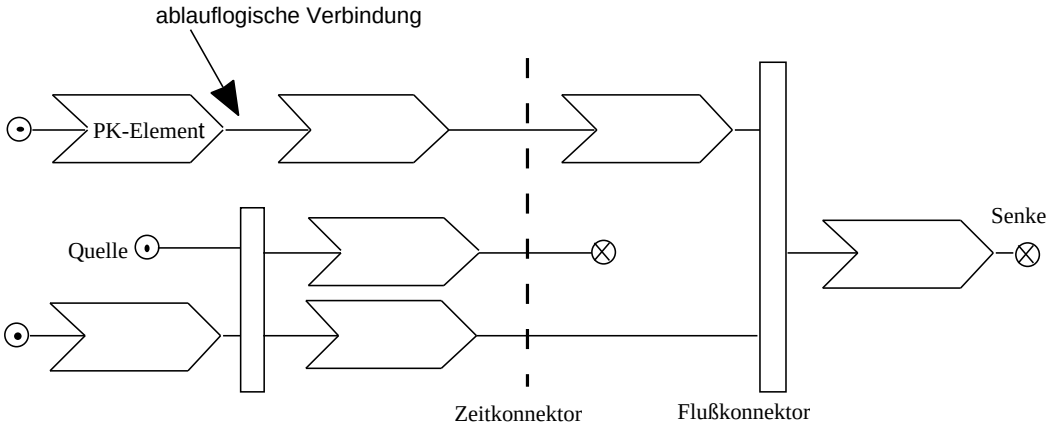


Abbildung 2-3: Beispiel eines Prozeßkettenplans

In der Literatur tauchen stellenweise weitere Konnektoren auf, wie z. B. UND-Konnektoren (vgl. /WiQu97/), welche graphisch als Kombination o. g. Konnektoren dargestellt werden, allerdings oft eine andere Semantik besitzen.[1] Vielfach werden hierdurch Synchronisationsbedingungen für einen oder mehrere(!) Prozesse definiert, wobei das Starten oder die Beendigung einzelner Prozesse implizit beschrieben wird. Beispielsweise wird in Abbildung 2-4 die Aktivitätsreihenfolge eines Prozesses *P1* beschrieben. Nach Beendigung der Aktivität *P1a* setzt der Prozeß seine Aktivitäten mit der Aktivität *P1f* fort und startet einen neuen (Unter-)Prozeß, der als erstes eine Aktivität *P2* ausführt.

Die bisher dargestellten Aspekte des Prozeßkettenparadigmas sollen uns im folgenden bei der Definition des B1-Formalismus leiten. Im nächsten Kapitel wollen wir einige Anwendungsbeispiele betrachten und hieran Anforderungen an die Beschreibungsmöglichkeiten des Formalismus ableiten. Dabei sollen die beschriebenen Systeme vorerst schwerpunktmäßig hinsichtlich ihrer Leistungsaspekte beurteilt werden. Hierunter fallen Kennzahlen, wie Durchlaufzeit, Termintreue, Bestände etc. (vgl. /KLÖP91/). Im Laufe der Abstimmungsphase innerhalb des SFB werden ökonomische und ökologische Aspekte hinzukommen.

[1] Eine solche Kombination von Konnektoren wurde bereits in Abbildung 2-2 verwendet.

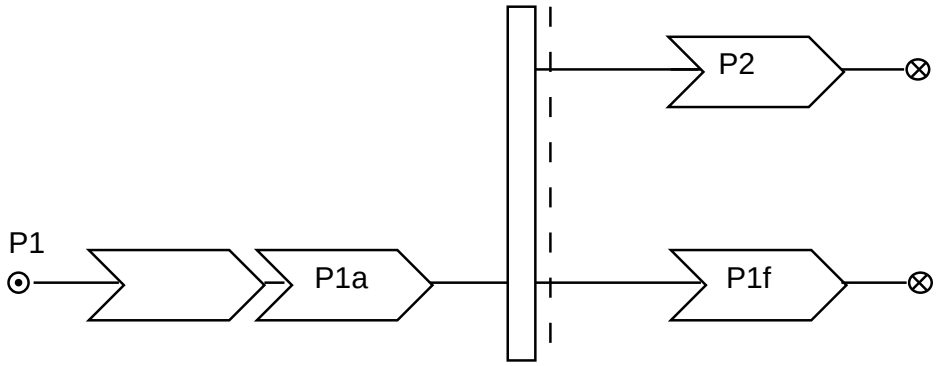


Abbildung 2-4: Beispiel in Anlehnung an /WiQu97/

3 Anwendungsbeispiele

Im folgenden werden einige Beispiele aus unterschiedlichen Anwendungsbereichen beschrieben. Ziel ist es, Beschreibungsanforderungen an den zu definierenden B1-Formalismus zu identifizieren, so daß möglichst viele Aspekte der jeweiligen Anwendungsbereiche erfaßt werden können. Bei der Beschreibung werden teilweise auch prozeßkettenorientierte Darstellungen zur Illustration verwendet. Diese Darstellungen entsprechen nicht den Darstellungen des zu definierenden B1-Formalismus.

Die Beispiele der Kapitel 3.1 bis 3.4.1 beschreiben konkrete Anwendungssysteme. Kapitel 3.4.2 widmet sich der Darstellung einer allgemeinen Modellarchitektur der Distribution, wie sie von Herrn Dr. Schürholz in seiner Dissertation /SCHÜ99/ vorgestellt wird. Die Umsetzung dieser Darstellung in den B1-Modellformalismus betrachten wir als besondere Anforderung, um die Tauglichkeit des verfolgten Ansatzes zu untermauern.

3.1 Güterverteilzentrum (GVZ)

In diesem Beispiel wird ein Teilbereich eines GVZ modelliert. In dem betrachteten Bereich kommen Züge (aus Richtung eines Hauptgleises) auf einem Nebengleis an und werden ent- und beladen. Vereinfachend wollen wir in diesem Beispiel annehmen, daß sich immer nur maximal ein Zug innerhalb des betrachteten Teilbereichs des GVZ befindet.

Die zu be- und entladenden Güter gehören zwei Klassen an: Huckepack-Verkehr und KV-affine Güter (KV = kombinierter Verkehr). Huckepack-Verkehr bezeichnet komplette LKWs (Zugmaschine + Anhänger). KV-affine Güter sind Container, die auf LKWs geladen werden können. Zum Be- und Entladen von KV-affinen Gütern werden sogenannte Reach-Stacker verwendet, welches spezielle Gabelstapler sind. Die Dynamik des Systems ist durch folgende Abläufe spezifiziert.

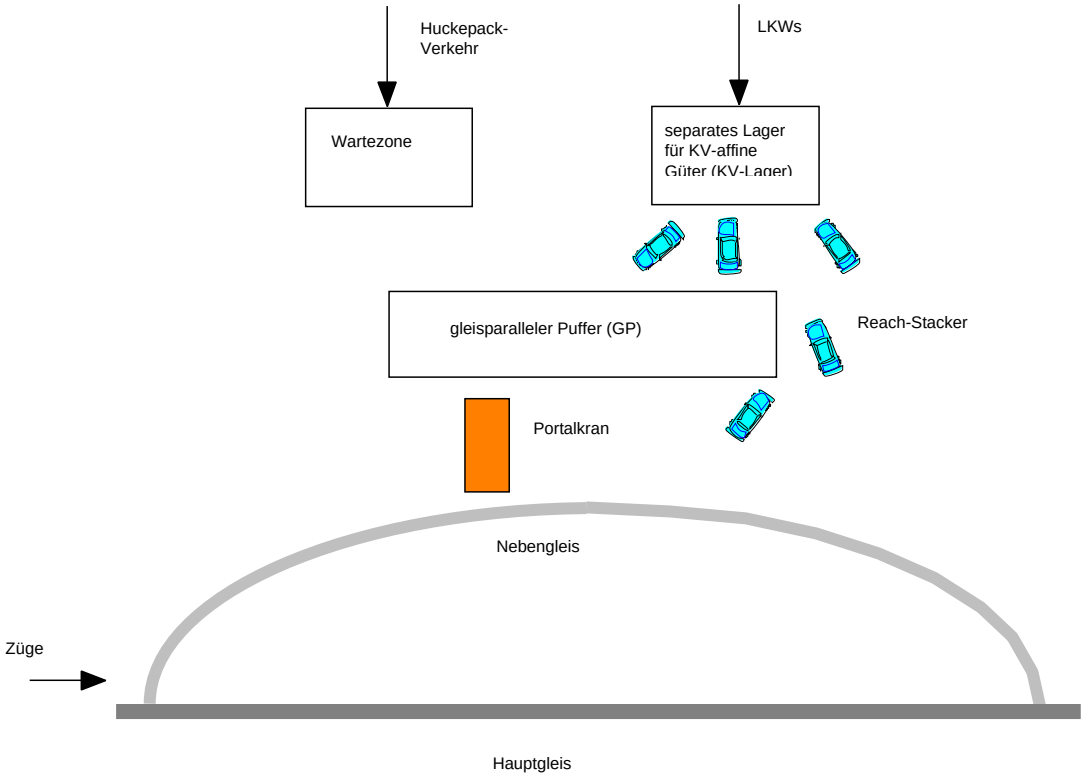


Abbildung 3-1: Layout GVZ

Huckepack-Verkehr kommt am GVZ an und wartet in einer Wartezone auf Beladung auf einen Zug. Des weiteren kommen LKWs an, die KV-affine Güter in ein Lager abladen und aus diesem Lager aufnehmen. Hierzu werden ein oder mehrere Reach-Stacker benutzt. Kommt ein Zug an, so werden folgende Be- und Entladevorgänge parallel durchgeführt: Huckepack-Verkehr wird be- und entladen und KV-affiner Verkehr wird be- und entladen, wobei alle Entladevorgänge vor den entsprechenden Beladevorgängen ausgeführt werden. Für den Umschlag von Huckepack-Verkehr wird ein sog. gleisparalleler Puffer genutzt, welcher den Platz neben dem Zug bezeichnet, sowie ein Portalkran, der die LKWs vom bzw. auf den Zug hebt. Im Modell wird angenommen, daß nur ein Portalkran zur Verfügung steht. Züge sind mit Gütern beider Klassen beladen. Nach Entladung fährt der entladene Hukkepack-Verkehr aus dem betrachteten GVZ-Bereich heraus. Sind genügend gleisparallele Pufferplätze frei, so fährt der wartende Huckepack-Verkehr in diesen Bereich ein und wird auf den Zug geladen. Neben der Kapazität des gleisparallelen Puffers ist auch der Platz auf dem Zug für Huckepack-Verkehr beschränkt. KV-affine Güter werden mittels ein oder mehrerer Reach-Stacker direkt vom Zug in ein separates Lager für KV-affine Güter transportiert. Diese Reach-Stacker unterstützen auch die Be- und Entladevorgänge KV-affiner Verkehre am Zug, sowie der LKWs.

Abbildung 3-1 gibt eine schematische Darstellung des Layouts wieder.

Die oben geschilderten Abläufe lassen sich wie folgt recht abstrakt beschreiben. Ein Zug gelangt vom Hauptgleis auf das Nebengleis, wird ent- und beladen (*umschlagen*) und verläßt nachfolgend das Nebengleis wieder in Richtung Hauptgleis. Analog fahren LKWs, die KV-affine Güter transportieren, in das GVZ ein, werden dort umgeschlagen und verlassen daraufhin wieder das GVZ. Züge und LKWs für KV-affine Güter treten also aus der Systemumwelt in das GVZ ein und verlassen dieses wieder nach Durchführung einiger Aktivitäten:

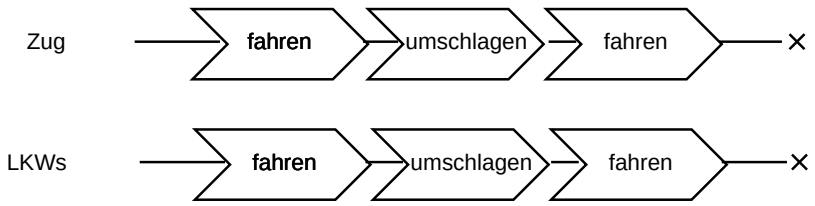


Abbildung 3-2

Interessant ist die Betrachtung der Abläufe des Huckepack-Verkehrs. Hier kommen z. B. LKWs an (siehe *Huck-arr*), welche das GVZ nur mittels Zügen verlassen. Dies bedeutet, daß das Leistungsobjekt LKW in dem Leistungsobjekt Zug aufgenommen wird, wodurch sich für Analysezwecke die Frage nach der Lebensdauer von Leistungsobjekten stellt, z. B. zwecks Durchlaufzeitberechnung.

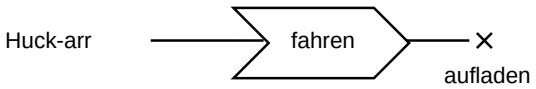


Abbildung 3-3

Analoges geschieht beim Entladen des Huckepack-Verkehrs vom Zug. Bei dieser Aktivität entstehen ein oder mehrere Leistungsobjekte, nämlich die abgeladenen LKWs:

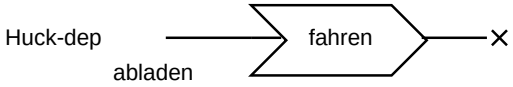


Abbildung 3-4

Werden die einzelnen Aktivitäten zusammenhängend betrachtet, so sind gewisse zusätzliche Restriktionen zu berücksichtigen. Beispielsweise kann mit dem Umschlagen des Huckepack-Verkehrs erst nach Ankunft eines Zuges begonnen werden. Solche Kausalitätsbeziehung und damit auch zeitliche Abfolgen sind sicherlich grundlegend in unserem Paradigma und bedürfen daher der Beschreibung. Eine mögliche Form wird durch folgende Abbildung aufgezeigt:

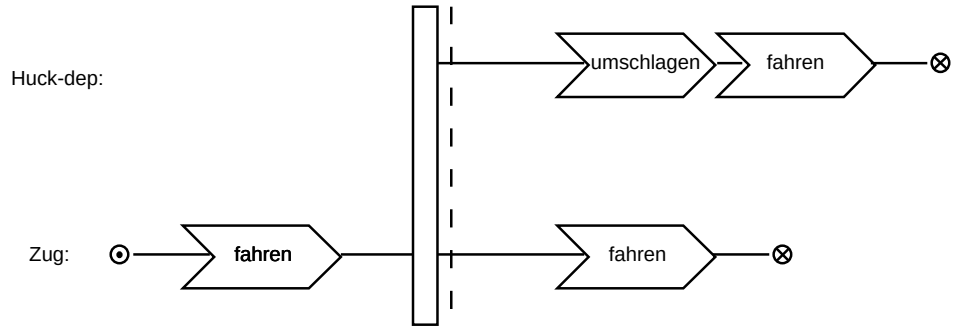


Abbildung 3-5

Leider entspricht diese Darstellung nicht ganz der vorgegebenen Systembeschreibung, da der Zug erst frühestens dann weiterfahren darf, wenn alle(!) LKWs abgeladen worden sind. Auch eine Darstellung der Form

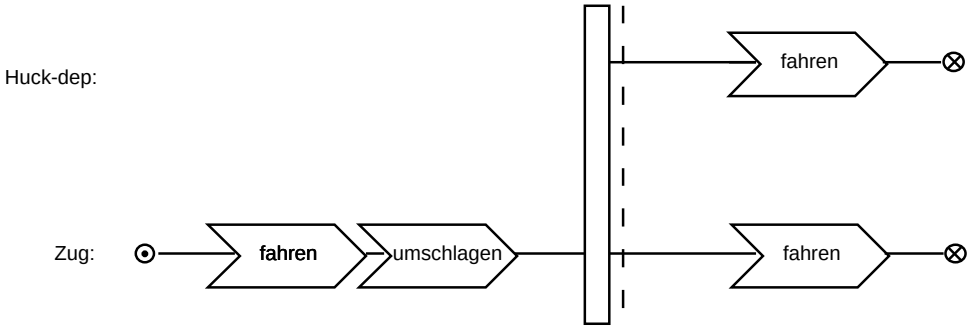


Abbildung 3-6

stellt das Systemverhalten nicht ganz korrekt dar, da hier alle LKWs erst dann weiterfahren können, wenn auch der Zug weiterfährt. Bei der angestrebten Definition des B1-Formalismus werden wir also auf solche Aspekte achten müssen.

Eine weitere Ungenauigkeit, die unsere derzeitige Modellbeschreibung noch enthält, bezieht sich auf die Anzahl der auftretenden Leistungsobjekte. So wird z. B. beim Entladen des Huckepack-Verkehrs nicht nur ein LKW entladen, sondern im allgemeinen mehrere. In unserer Darstellung wird diese Information noch nicht wiedergegeben. Hierbei ist zu beachten, daß die Anzahl der entladenen LKWs nicht konstant ist, sondern von der Ladung des Zuges abhängt.

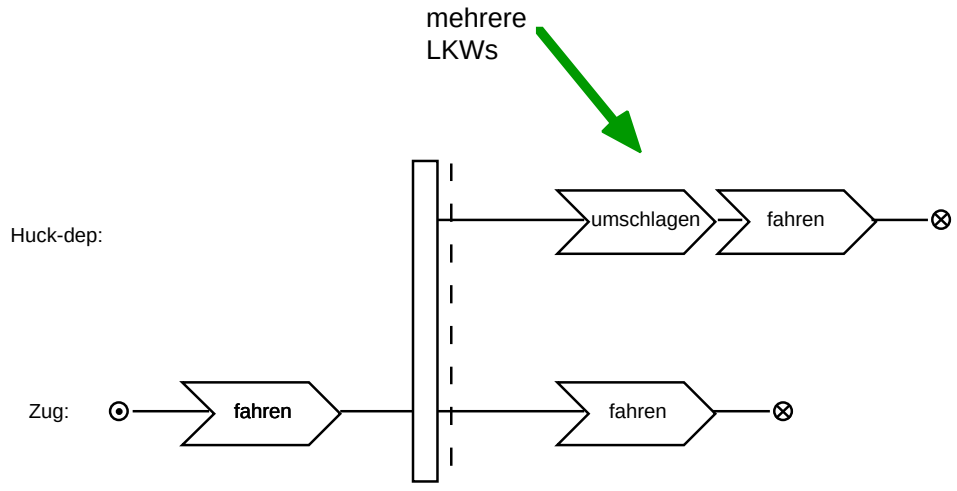


Abbildung 3-7

Neben den angesprochenen Systemcharakteristika müssen wir zusätzlich weitere in unsere Beschreibung aufnehmen. So benötigt beispielsweise eine Vielzahl der Tätigkeiten Ressourcen/Funktionseinheiten zur ihrer Durchführung. Z. B. wird zum Umschlagen des Huckepack-Verkehrs der Portalkran benötigt. Ein LKW des Huckepack-Verkehrs kann hierdurch das Umschlagen anderer beeinflussen, indem er den Portalkran gerade nutzt und dieser daher nicht für andere LKWs zur Verfügung steht. Ferner fehlt die Beschreibung, daß Huckepack-Verkehr nur aufgenommen werden kann, wenn noch entsprechende Plätze auf dem Zug zur Verfügung stehen. Somit wird in Abhängigkeit von gewissen Bedingungen bzw. Zuständen des Systems der Fluß der Leistungsobjekte „Huckepack-Verkehr“ gesteuert. Als Bedingung muß im wesentlichen ausgedrückt werden, daß das Abladen des Huckepack-Verkehrs beendet ist und dieser den gleisparallelen Puffer verlassen hat und, daß die Anzahl freier Plätze auf dem Zug ausreicht, um den Huckepack-Verkehr aufzunehmen.

Wie bereits dieses recht kleine Beispiel zeigt, sind mehrere Problemfelder bei einer (zur Leistungsbewertung geeigneten) Definition des B1-Modellformalismus zu beachten, insbesondere

- Nutzung von Ressourcen/Funktionseinheiten,
- Zustandsabfrage/-änderung von Ressourcen/Funktionseinheiten und/oder Prozessen,
- Erzeugung und Löschung von Leistungsobjekten.

3.2 Redistributionsnetz

In diesem Beispiel betrachteten wir Kreisläufe von Mehrweg-Transportverpackungen, im folgenden auch Behälter genannt. Abbildung 3-8 skizziert das Layout des Redistributionsnetzes.

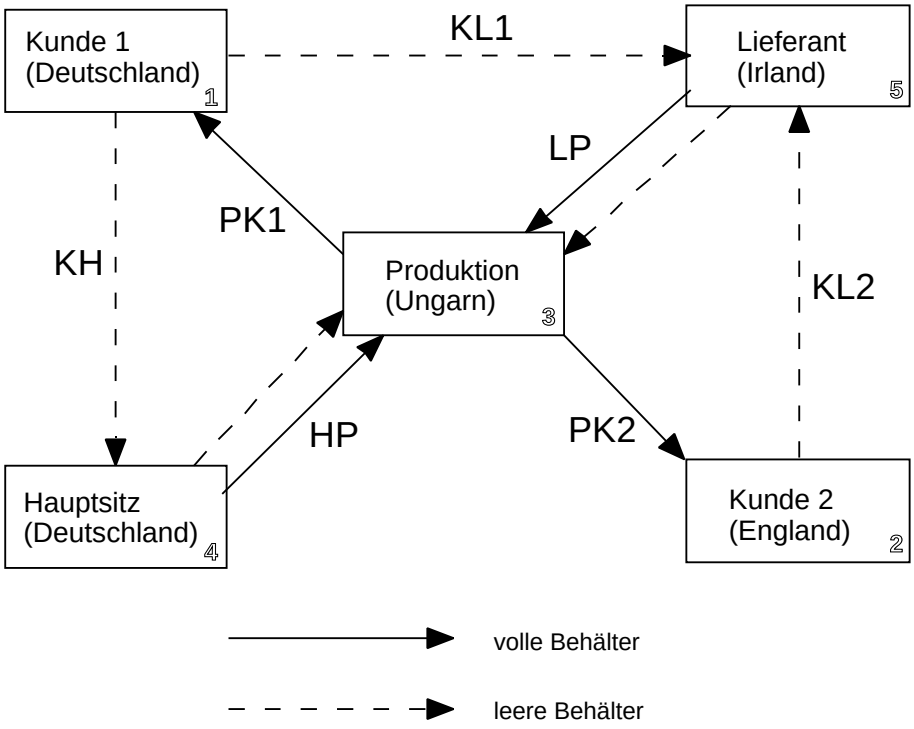


Abbildung 3-8: Redistributionsnetz

Waren werden in Ungarn produziert, in Behälter verpackt und zu Kunden in Deutschland und England versandt. Zur Produktion werden Einbauteile benötigt, die von einem Lieferanten in Irland bzw. vom Hauptsitz eines der Kunden aus Deutschland geliefert werden. Diese Einbauteile werden in den gleichen Behältern wie die Produkte verpackt und transportiert. Die Belieferung aller Akteure erfolgt wöchentlich (also periodenweise und i. a. am Wochenende), wobei die Produktionsdaten für mehrere

Wochen im voraus bekannt sind. Es wird angenommen, daß es mittels dieser Daten möglich ist, die jeweils in einer Woche benötigte Anzahl an Behältern zur Versendung der Güter zu ermitteln und der Behälterkreislauf wie folgt ist.

Vom ungarischen Hersteller werden volle Behälter an die Kunden in Deutschland (*PK1*) und England (*PK2*) versandt. Beide Kunden entnehmen die gelieferten Waren und geben diese beispielsweise an ihre Auftraggeber weiter. Die leeren Behälter werden je nach, aus den Produktionsdaten ermittelten, Bedarfen an die entsprechenden Lieferanten in Deutschland (*KH*) und Irland (*KL2*) weitergeleitet. Die Lieferanten (in Irland und Hauptsitz Deutschland) ihrerseits liefern volle Behälter an die Produktionsstätte in Ungarn (*HP* bzw. *LP*). Falls in Ungarn mehr Güter zur Produktion eingeplant sind, kann es vorkommen, daß zusätzliche Behälter benötigt werden. In diesen Fällen werden auch leere Behälter von den Lieferanten versendet. Im allgemeinen erfolgt dies zusammen mit der Lieferung der vollen Behälter. In diesem System ist als Besonderheit noch vorgesehen, daß bei einem Behälterengpaß in Irland notfalls auch leere Behälterlieferungen des Kunden aus Deutschland erfolgen, also ein sogenannter Leerausgleich stattfindet (*KL1*).

Bei diesem Redistributionsnetz ist z. B. die durchschnittliche bzw. maximale Anzahl an Behältern von Interesse, die am jeweiligen Ort vorgehalten werden muß, da hierdurch Lagerraum belegt wird. Ferner interessiert auch der Umfang der Leerausgleiche, da diese zusätzliche Kosten verursachen, sowie die Gesamtanzahl an Behältern im Netz, die für einen geregelten Betrieb notwendig sind.

Im Hinblick auf eine Modellierung des Netzes mittels einer prozeßkettenorientierten Darstellung spielt natürlich die Wahl des Leistungsobjektes eine wesentliche Rolle. Als Leistungsobjekt kommt hier der Behälter in Frage, der die jeweiligen Orte (1,...,5, siehe Abbildung 3-8) „aufsucht“, dort beladen oder entleert und ggf. zwischengelagert wird. Eine prozeßkettenorientierte Notation könnte beispielsweise folgendermaßen aussehen:

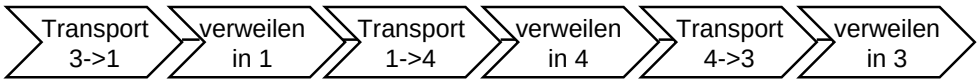


Abbildung 3-9: Prozeßkettenorientierte Darstellung Redistributionsnetz (Ausschnitt)

Diese Abfolge von Aktivitäten soll den Prozeß beschreiben, der den Transport eines Behälters vom Ort 3 (Ungarn, vgl. Abbildung 3-8) über die Orte 1 und 4 wieder zum Ausgangsort 3 beinhaltet.

Obige Darstellung ist natürlich noch unvollständig, da sie einerseits nicht die potentiell möglichen Besuche des betrachteten Behälters an den Orten 2 und 5 berücksichtigt, andererseits auch nicht beschreibt wie lange der jeweilige Behälter an einem Ort verweilt. Wie bereits angesprochen sollen die Behältertransporte so abgestimmt werden, daß für die in einer Woche produzierten Güter ausreichend Behälter zum Transport bereitstehen. Dies bedingt beispielsweise, daß eine bestimmte Anzahl leerer Behälter 2 Wochen zuvor von Ort 1 via 4 nach 3 zurückgeschickt wird, um so einen Engpaß an Behältern in 3 zu vermeiden. Durch solche Lenkungsmaßnahmen wird die Verweilzeit eines Behälters maßgebend beeinflußt, denn würden, um bei unserem Beispiel zu bleiben, keine zusätzlichen Leerbehälter in 2 Wochen benötigt, so könnten die leeren Behälter länger in 1 oder nachfolgend in 4 verweilen.[2] Es genügt also nicht, Behälter isoliert zu betrachten, sondern es müssen alle Behälter des Systems, und somit alle beschriebenen Prozesse in ihrer Gesamtheit erfaßt werden, da sich hierdurch u. a. die Anzahl der Leertransporte ergibt.

Der Prozeßkettenausschnitt in Abbildung 3-9 stellt einen möglichen Ablauf dar. Betrachten wir beispielsweise einen Behälter verweilend im Orte 1. Dieser Behälter kann nach einer gewissen Zeit nach 4 versandt werden und von dort z. B. noch die Orte 3, 2, 5, 3 aufsuchen, bevor er nach 1 zurückkehrt, wobei auch zusätzliche (zyklische) Besuche der Orte 3, 2, 5 denkbar sind. Diese potentiell möglichen Abläufe werden durch unsere bisherige Modellvorstellung noch nicht erfaßt, da wir bisher immer nur endliche sequentielle Abläufe beschrieben haben. So auch in Abbildung 3-9. Hier kann das Ende der Aktivität „verweilen in 3“ (am Ende der Prozeßkette beschrieben) den Beginn der Aktivität

[2] Eventuell müßten sie sogar in 4 verbleiben um kurzfristig anstehende Engpässe dort zu vermeiden.

„Transport 3->1“ (am Anfang der Prozeßkette beschrieben) bedingen, wenn es sich um denselben Behälter handelt.

Im Vergleich zu unserem letzten Beispiel aus den Bereich der Güterverteilzentren kommt also zusätzlich die Notwendigkeit auf, zyklische Abläufe beschreiben zu müssen. In dem vorliegenden Beispiel konnte diese Problematik noch umgangen werden, da alle Kreisläufe über den Ort 3 laufen, der sich somit als Punkt „zum Aufschneiden“ des Zyklus anbietet. Wie oben angesprochen würde ein Schneiden im Orte 1 nicht zu einer adäquaten sequentiellen Darstellung führen.

3.3 Beschaffung

In diesem Beispiel wird die Beziehung zwischen einem Hersteller eines Gutes und einem Lieferanten eines bei der Herstellung häufig verwendeten Teils (A-Teil) betrachtet. Ferner wird ebenso die Beziehung zwischen diesem Lieferanten und dem mit dem Transport der A-Teile beauftragten Dienstleister dargestellt.

Folgender Ablauf liegt dem Beispiel zugrunde: Täglich wird beim Hersteller geprüft, wieviel A-Teile für den siebtfolgenden Tag für die Produktion benötigt werden (es wird also stets eine Woche vorausgeplant). Der Bedarf (ca. 800 pro Tag) wird verglichen mit dem Lagerbestand (Lagerkapazität 1600) und daraus die Bestellmenge ermittelt. Diese wird dem Lieferanten übermittelt, der die benötigte Menge produziert (Fertigungsdauer etwa 4 Stunden) und anschließend einer Qualitätskontrolle unterzieht. Die Prüfung erfolgt parallel an 10 Arbeitsplätzen und beansprucht ca. 10 Minuten pro A-Teil, so daß sich eine Prüfungsdauer von durchschnittlich 1 Minute je A-Teil ergibt. Für den ordnungsgemäßen Anteil wird einem Transportdienstleister ein Transportauftrag erteilt, anschließend werden die A–Teile beim Lieferanten gepuffert, bis sie in spezielle Behälter verpackt werden. Die verpackten A–Teile werden bis zum Beginn des Transports ein weiteres Mal beim Lieferant gepuffert. Der Transportdienstleister erstellt nach der Annahme des Transportauftrags seine Tourenplanung und wird den Transport der verpackten A-Teile zum Hersteller übernehmen. Die Kapazität der LKWs beträgt dabei 60 Teile je Fahrzeug[3]. Ca. alle 2 Stunden fährt ein LKW vom Lieferanten zum Hersteller. Die Fahrzeit beträgt im Mittel 4 Stunden. Nach der Ankunft beim Hersteller werden die A-Teile entladen (Dauer etwa 2 Stunden) und erneut geprüft. Die Ausschußware wird gesammelt und die Gutteile werden eingelagert, bis sie zur Fertigung benötigt werden. Nach der Fertigung werden die Fertigteile geprüft und eingelagert sowie das Leergut für die A-Teile beim Hersteller gepuffert.

Zur Auswertung dieses Ablaufs sollen die Logistikkosten (Lagerkosten, Transportkosten etc.), die Versorgungssicherheit, die Bestände in den Lägern sowie die Lieferzeit bzw. die Termintreue betrachtet werden.

Die Abbildung 3-10 spiegelt die Grobstruktur dieser Vorgänge wider.

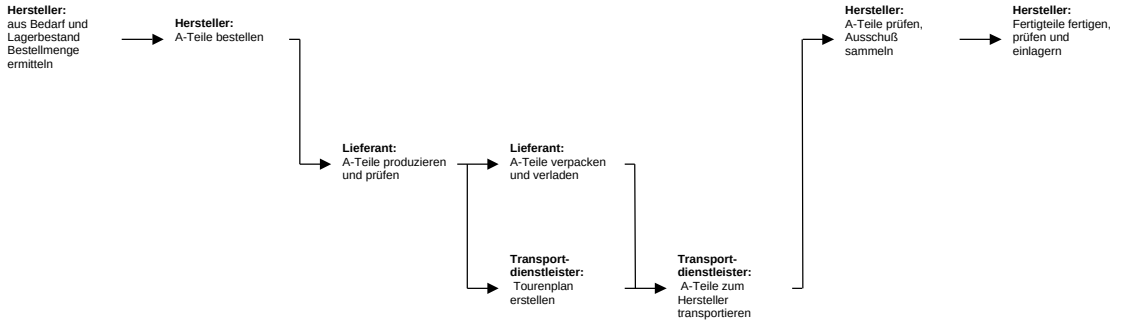
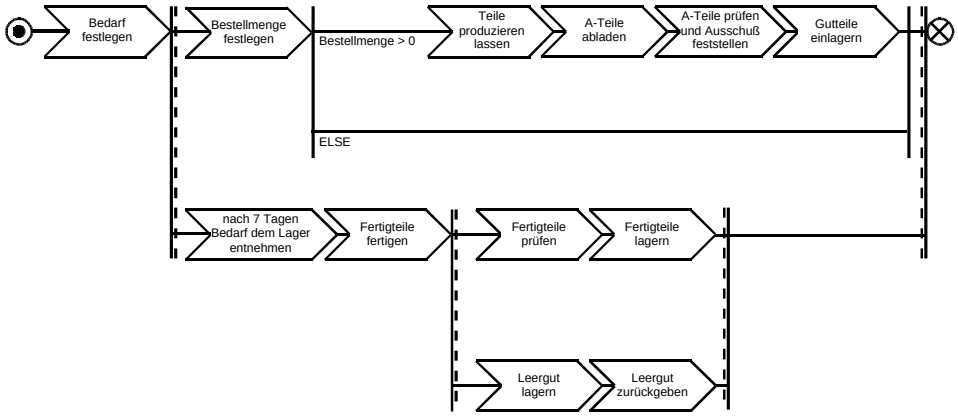


Abbildung 3-10: Grobstruktur der Vorgänge bei der Beschaffung

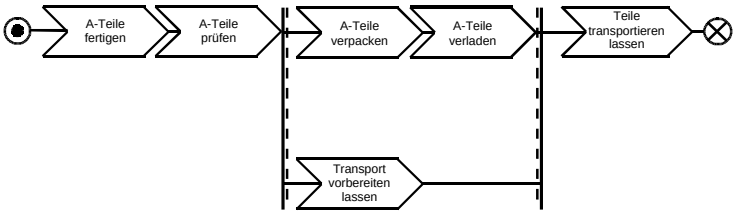
[3] Die A-Teile (Stoßfänger) werden in Behältern mit einem Fassungsvermögen von 10 A-Teilen transportiert. Ein LKW faßt ca. 6 Behälter.

Für die Darstellung dieses Beispiels werden neben einem Lager beim Hersteller für die A-Teile Einheiten benötigt, die für das Ausladen der A-Teile beim Hersteller, für das Fertigen der Fertigteile, das Prüfen dieser Teile, das Fertigen der A-Teile beim Lieferanten, das Prüfen dieser Teile, das Verpacken und Verladen der A-Teile beim Lieferanten, die Tourenplanung und den Transport benötigt.

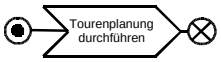
Hersteller:



Lieferant:



Transportdienstleister:



sowie

Abbildung 3-11: Grobe Prozeßkettenstruktur der Vorgänge bei der Beschaffung

Für die Umsetzung des Beispiels in eine prozeßkettenorientierte (!) Darstellung soll im folgenden geprüft werden, welche Bestandteile in welchen Potentialklassen zu berücksichtigen sind:

- Lenkungsebenen: Im wesentlichen betrifft die Steuerung den Aspekt, daß der in obiger Abbildung beschriebene Vorgang einmal täglich gestartet wird.

- Strukturen: Die Struktur wird durch die Unterscheidung der Einheiten Hersteller, Lieferant und Transportdienstleister sowie durch Untereinheiten, die für bestimmte Teilaufgaben zuständig sind, erfaßt.

- Ressourcen: Als Ressource wird das A-Teile-Lager beim Hersteller benötigt.

- Prozesse: Zur Außenwelt ist das Modell über eine Quelle für die Starts des in obiger Abbildung beschriebenen Ablaufs sowie eine Senke für das Beenden dieser Abläufe verbunden. Daneben sind die Einheiten zu definieren, die für die Erfüllung der oben beschriebenen Teilaufgaben verantwortlich sind.

Als Leistungsobjekte dienen in diesem Beispiel die an einem Tag bestellte Menge an A-Teilen bzw. die daraus entstehenden Fertigteile. Innerhalb des Beispiels beauftragt der Hersteller den Lieferanten mit der Produktion der A-Teile und dieser wiederum den Transportdienstleister mit dem Transport dieser Teile zum Hersteller.

In Abbildung 3-11 sind die Grobstrukturen aus Abbildung 3-10 für die Bereiche Hersteller, Lieferant und Transportdienstleister getrennt in Prozeßkettenschreibweise wiedergegeben (eine genaue Darstellung und Beschreibung erfolgt im Kapitel 5.3):

Anhand dieses Beispiels ist erkennbar, daß bei der Definition des B1-Modellformalismus zu berücksichtigen ist,

- wie Läger zu handhaben sind (einlagern, auslagern, Bestandsmengen abfragen),
- wie es darzustellen ist, daß die Bereiche Hersteller, Lieferant und Transportdienstleister unabhängig voneinander agieren (z. B. wartet der Hersteller realistischerweise nicht auf eine sich verspätende Lieferung, sondern wird den Herstellungsprozeß pünktlich starten, wenn sich noch entsprechende A-Teile aus vorherigen Lieferungen in seinem Lager befinden),
- wie Werte verschiedener Durchläufe ausgetauscht werden können. (Dies ist notwendig, um feststellen zu können, wieviel A-Teile noch geliefert werden sollen, bis die jetzt zu bestellenden A-Teile benötigt werden, bzw. wie hoch die einzelnen Bedarfe bis dahin sein werden. Dies ist bei der Prüfung des Lagerbestandes zum Zweck der Bestellmengenermittlung zu berücksichtigen.)

3.4 Distribution

In Ermangelung eines konkreten Beispiels aus den Anwendungsprojekten werden hier Beispiele auf Basis aufgearbeiteter Literatur entworfen und betrachtet. Abschnitt 3.4.1 extrahiert einen einfacheres (artifizielles) Distributionsnetz; Abschnitt 3.4.2 basiert auf einer Dissertation, welche insgesamt der Distributionsproblematik gewidmet ist.

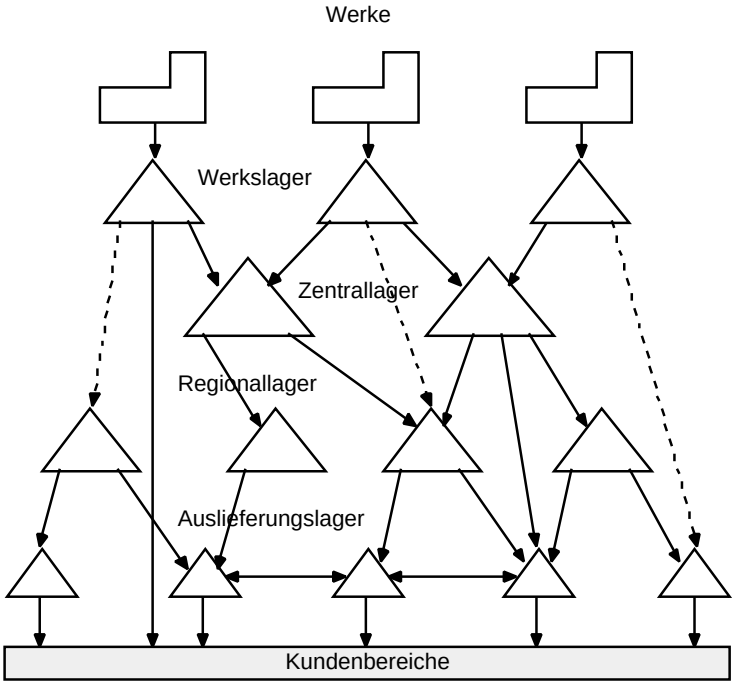


Abbildung 3-12: Allgemeines Distributionsnetz

3.4.1 Anwendungsbeispiel der Distribution

An dieser Stelle wird ein vereinfachtes Distributionsnetz entworfen. Dieses Netz soll klein und einfach sein und trotzdem typische Probleme der Distributionsnetze in sich vereinen.

Zunächst wird kurz auf allgemeine Distributionsnetze eingegangen: Bei der Distribution geht es um die Belieferung von Kunden mit Produkten, die in verschiedenen Werken produziert werden. Diese Produkte werden in verschiedenen Zentral-, Regional- und Auslieferungslagern gelagert, bevor sie die Kunden erreichen. Die gefertigten Produkte werden aus dem Werkslager auf die Zwischenlager verteilt, um die Ware im Falle einer Bestellung möglichst schnell liefern zu können. Die Kunden werden aus den Beständen der Auslieferungslager versorgt. Falls der Bestand nicht ausreicht, wird die fehlende Menge aus Regionallagern bzw. anderen Auslieferungslagern ergänzt. Die Abbildung 3-12 soll ein allgemeines Distributionsnetz darstellen (vgl. /ScKr95/).

Um ein einfaches Distributionsmodell zu entwerfen, wird angenommen, daß nur ein Kunde mit Produkten aus zwei verschiedenen Werken beliefert wird. Diese Produkttypen werden im folgenden mit „A“ und „B“ differenziert. Weiterhin gibt es nur ein Zwischenlager. Dieses Lager wird von den beiden Werken beliefert. Die Produkte werden hier gelagert, bis ein Kundenauftrag vorliegt. Bei einem Kundenauftrag werden die Produkte aus dem internen Lager ausgelagert, kommissioniert und zum Kunden transportiert, d. h. es handelt sich hierbei um ein Kommissionierlager, welches zwei verschiedene Produkttypen lagern und kommissionieren kann. Die Produktion in den Werken wird von den Beständen in diesem Lager gesteuert. Neue Ware wird nur dann produziert, wenn die entsprechenden Bestände in den Werken eine bestimmte untere Grenze unterschreiten. Den schematischen Aufbau dieses kleinen Beispiels gibt die folgende Abbildung wieder.

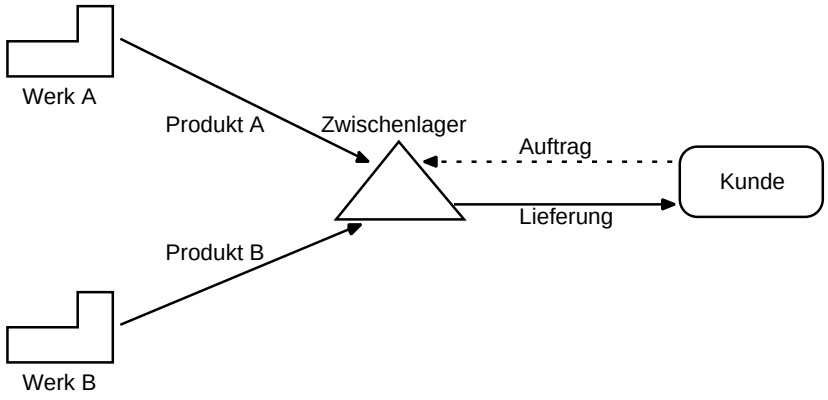


Abbildung 3-13: Ein kleines Beispiel der Distribution

Offensichtlich gibt es in diesem Beispiel genau drei Prozesse, die im folgenden genauer beschrieben werden. Dazu wird eine prozeßkettenorientierte Darstellung verwendet.

Die beiden ersten Prozesse bilden die Produktion, den Transport und die Einlagerung der beiden Produkttypen ab.

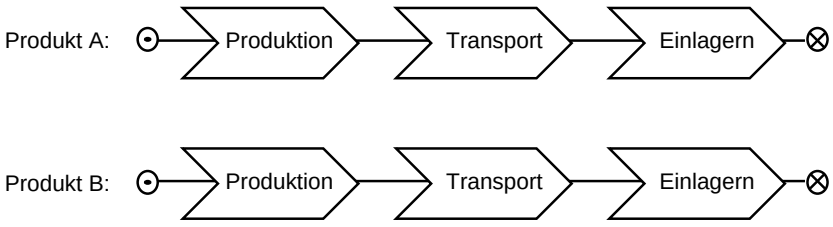


Abbildung 3-14: Prozesse der Produktion

Der dritte Prozeß stellt das Auslagern, den Transport und das Entladen der Lieferung beim Kunden dar. Die Lieferung enthält die Zusammenstellung der beiden Produkttypen, die im Kundenauftrag angegeben ist.

Abbildung 3-15: Auftragsprozeß

Da das Auslagern offensichtlich mehrere Aktivitäten umschließt, muß sie detailliert werden. Die Detaillierung wird im folgenden beim Zwischenlagerbaustein beschrieben.

Zunächst werden die vier Bausteine des Modells in Abbildung 3-13 näher erläutert. Dazu gehören

- die Werke,
- der Kunde,
- die Spedition und
- das Zwischenlager.

Werke

Bei den Werken handelt es sich um Produktionsstätten ohne Werkslager. Die produzierte Ware wird direkt zur Beladestation gebracht. Von dort werden die Produkte auf LKWs geladen und zum Zwischenlager transportiert. Den Transport zur Beladestation und das Beladen der LKWs führen Gabelstapler durch. Der Transport der Ware vom Werk zum Zwischenlager wird dabei von einer Spedition durchgeführt. Die Beladestationen der Werke besitzen eine bestimmte Anzahl von Gabelstaplern, die sowohl den Transport vom Werk zur Beladestation, als auch das Beladen der LKWs mit den entsprechenden Produkten übernehmen. Nach dem Beladen wird die Ware zum Zwischenlager transportiert.

Kunde

Der Kundenbaustein ist analog zu den Werkebausteinen aufgebaut. Anstelle der Beladestation besitzt der Kunde eine Entladestation, wo eine bestimmte Anzahl von Gabelstaplern die Entladung der angekommenen LKWs durchführen.

Spedition

Die Spedition übernimmt den Transport von Produkten sowohl zwischen Werken und Zwischenlager als auch zwischen Zwischenlager und Kunden. Die Spedition besitzt eine bestimmte Menge von LKWs. Diese werden je nach Bedarf auf den drei verschiedenen Strecken eingesetzt.

Zwischenlager

Das Zwischenlager besteht aus der Be-/Entladestation, dem internen Lager und einer Kommissionierstation. Das Be-/Entladen und den lagerinternen Transport führen eine bestimmte Menge von Gabelstaplern durch. Die entladene Ware wird sofort zum internen Lager transportiert. Dabei ist das interne Lager zweigeteilt, d. h. die Produkttypen haben getrennte Lagerplätze (-kapazitäten). Die Ware lagert im internen Lager, bis ein Kundenauftrag vorliegt. Der Kunde kann bestimmte Mengen von beiden Produkttypen bestellen. Diese Mengen werden dann dem Lager entnommen und vor Ort kommissioniert. Die Kommissionierung findet schon auf einer Palette statt, so daß eine zusätzliche Palettierung entfällt. Die Palette wird zur Beladestation gebracht. Dort wird sie auf einen LKW geladen und zum Kunden transportiert. Diese geschilderten Abläufe verbergen sich hinter dem Prozeß *Auslagern* in Abbildung 3-15. Deswegen soll dieser Prozeß im folgenden detailliert werden.

Die Kommissionierung kann erst stattfinden, wenn die benötigten Produkte ausgelagert sind. Die dementsprechend erforderliche Synchronisation ist im Modell mittels eines UND-Konnektors ausgedrückt.

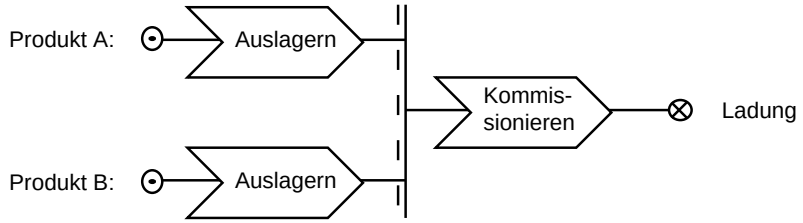


Abbildung 3-16: Detaillierter Auslagerungsprozeß

Der Beschreibung des Beispiels folgend, sind im Kontext der *Auslagern*-Prozesse die jeweiligen Lagerbestände zu überprüfen und ggf. die Produktionsprozesse neu anzuregen. Dieser Typ von Prozeß-Interaktion stellt ein wiederkehrendes Problem dar, das hier nicht explizit berücksichtigt ist, dem aber in den Folgekapiteln besondere Aufmerksamkeit zu widmen ist.

Eine weitere Problematik ist, daß in der Prozeßkette der Abbildung 3-16 potentiell drei verschiedene Leistungsobjekte im Spiele sind: Die vor Kommissionierung existierenden Produkte "verschwinden", die Ladung "entsteht". Dieser Typ von Modellierungs-Problem wurde bereits in Kapitel 3.1 betrachtet. Auch er bedarf erhöhter Aufmerksamkeit.

3.4.2 Ein Gesamtansatz nach Schürholz

Dieser Abschnitt widmet sich nicht wie die vorangegangenen einem kleineren Beispielmodell aus den Anwendungsprojekten, sondern greift einen umfassenden Ansatz zur Darstellung der aktuellen Distributionsproblematik auf, wie ihn A. Schürholz in seiner Dissertation /SCHÜ99/ geschildert hat. Der Abschnitt skizziert insbesondere die in /SCHÜ99/ wiedergegebene prototypische Fassung eines Gesamtmodells. Dieses Gesamtmodell ist vertikal strukturiert in eine Abfolge von Ebenen. Top-down gelistet sind dies (auf die Wiedergabe einer Normativen Lenkungsebene wird in /SCHÜ99/ bewußt verzichtet):

- eine Dispositive Lenkungsebene, welche die „globalen“ Aufgaben der Materialdisposition, der Behälterdisposition und der Fahrzeugdisposition erfaßt;
- eine Netzwerkebene, welche Aufgaben der Informationszustellung zwischen den globalen Systemkomponenten (Standorten), der Auftragszerlegung und der Auftragsbündelung beinhaltet;
- die Lokale Steuerungsebene des jeweiligen Einzel-Standorts unter Einschluß insbesondere der Lokalen Lagersteuerung, der Lokalen Behältersteuerung und der Lokalen Fahrzeugsteuerung;
- die Ebene der Physikalischen Modellbausteine des Standorts, insbesondere des Lagerbausteins, des Beladebausteins und des Ladebausteins,
  - wobei die Physikalischen Modellbausteine repräsentiert werden durch Baustein-Hauptprozesse,
  - die sich ihrerseits auf Modellprozesse abstützen,
  - wobei letztere über typisierten Modellfunktionen und Operationen erklärt sind.

Die Darstellung ist zielgerichtet auf einer „technikunabhängigen“ Detaillierungsstufe vorgenommen. Sie ist in Teilen prozeßkettenorientiert, mit einem diesbezüglichen Schwerpunkt bei der Ebene der Physikalischen Modellbausteine und den dort eingesetzten Operationen und Modellfunktionen. Die Darstellung unterscheidet explizit zwischen „logischem“ Informations- und „physischem“ Materialfluß.

Verfolgt man zur Erfassung des dynamischen Geschehens innerhalb eines Gesamtsystems das Ziel einer prozeßorientierten Darstellung, und läßt man sich hierbei von der Maxime leiten, jeden Prozeß zur Erfassung des „Lebens“ jeweils eines „Leistungsobjektes“ einzusetzen, dann bieten sich in einem

Distributionssystem verschiedene (Typen von) logischen und physischen Leistungsobjekten als Prozeßträger an:

· Aufträge vom Zeitpunkt ihrer Entstehung bis zum Zeitpunkt ihrer Erfüllung (bzw. für Teilintervalle aus dieser Gesamtspanne ihrer Abwicklung),

· Transportmittel (Fahrzeuge, Züge, etc.) in einem bestimmten zeitlichen Intervall ihres „Lebens“,

· Objektträger (Container, Paletten, etc.) in einem bestimmten zeitlichen Intervall,

· Objekte/Objektgruppen der Bestellung und Lieferung. in einem bestimmten Intervall,

· u. U. weitere, wie etwa Personen in ihrer jeweiligen betriebsspezifischen Rolle,

wobei diese Prozeßträger, je nach Bedarf, jeweils in mehreren Alternativen vorliegen.

Diese Prozeßtypen sind offensichtlich wechselseitig stark verschränkt; insbesondere absolvieren diverse Einzelprozesse jeweils Teile ihrer „Leben“ gemeinsam, so daß entsprechende, modellseitig angelegte Prozesse in mannigfaltiger Art zu synchronisieren sind. Der Versuch einer Gesamtdarstellung von Distributionssystemen resultiert demzufolge (erwartungsgemäß) in hochkomplexen Modellen. Eine fallweise Konzentration auf

· Ausschnitte des Gesamtsystems

· konkrete Analyseziele

verspricht, diese Komplexität zu reduzieren. Dieser Weg wird in der zugrundeliegenden Arbeit /SCHÜ99/ auch beschritten. So wird interessanterweise der Auftrag als i. w. alleiniges Leistungsobjekt (Prozeßträger) betrachtet, wohingegen Lieferobjekte, Fahrzeuge, Behälter etc. keine individuelle Existenz besitzen. Auch tauchen die entscheidungstragenden „Strategie-Bausteine“ nicht in prozeßorientierter Beschreibung auf, sondern in Form verbaler Schilderungen, womit die beabsichtigte explizite Unterscheidung logischer und physischer „Flüsse“ zwar durchgehalten, aber nicht prozeßorientiert sichtbar gemacht wird.

Die vorliegende beispielhafte Darstellung des Schürholz'schen Modells setzt den Weg der Reduktion und Vereinfachung aus Übersichtsgründen deutlich fort, wobei die zusätzlichen Vereinfachungen aber modellierungstechnisch unwesentliche Details betreffen. Zur Erleichterung der Unterscheidung zum Schürholz'schen Original wird die Beispieldarstellung mit DIST_BSP bezeichnet. Die Strukturierung von DIST_BSP folgt weitestgehend jener aus /SCHÜ99/; Bezeichnungen der Form „Bezeichnung“ (d. h. in Anführungszeichen) greifen die Schürholz'sche Terminologie auf. /SCHÜ99/ schildert sein Gesamtmodell in bottom-up-Reihenfolge. Diese Schilderung wird im folgenden skizziert; in DIST_BSP vorgenommene Vereinfachungen werden dabei angesprochen.

3.4.2.1 Die Ebene der Modellprozesse

Die „Modellprozesse“ werden hier in der Reihenfolge ihres Auftretens im Prozeßfluß der höheren Ebenen gelistet. In die zeichnerischen Darstellungen eingefügte zusätzliche Kommentare sind durch geschweifte Klammern {…} kenntlich gemacht.

Modellprozesse Auftrennung und Verkettung

Anlieferungen und Auslieferungen erfolgen in DIST_BSP auf Fahrzeugen (Träger_von_Träger_Objekten, TTOs). An- und Auslieferungsmengen sind so bemessen, daß sich jede An- und Auslieferung (bzw. jeder entsprechende An- und Auslieferungsauftrag) auf ein einzelnes Fahrzeug bezieht. „Auftrennung“ und „Verkettung“ (z. B. von und zu Zügen) entfallen somit.

Modellprozeß Entladung "TTO" für "Träger_von_Träger-Objekt", z.B. Fahrzeug;
"TO" für "Träger-Objekt" , z.B. Palette

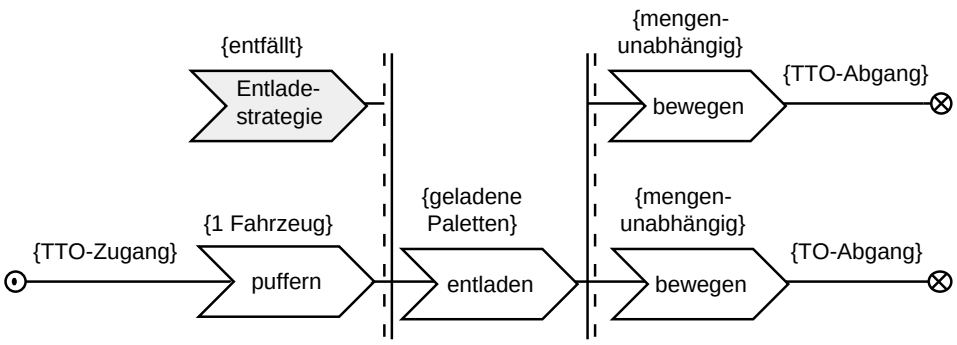


Abbildung 3-17

Dieser (in /SCHÜ99/ nicht explizit dargestellte) Modellprozeß ist in Analogie zu „Depalettierung“ erstellt.

Zur Vereinfachung wird in DIST_BSP eine feste Entladestrategie (nämlich: vollständiges Entladen der Fahrzeuge) angenommen. *Entladestrategie* als Einstellungsoption kann damit entfallen. Ferner wird hier (und durchgängig in ganz DIST_BSP) davon ausgegangen, daß alle Bewegungen zwar entsprechend konkreter Entfernungen unterschiedlich lange dauern können, diese Transportzeiten aber nicht von den bewegten Mengen abhängig sind (demnach als konstante Verzögerungen auftreten).

Modellprozeß Depalettierung "TO" für "Träger-Objekt" , z.B. Palette;
"O" für Objekt;
"Menge TOs" für die Paletten einer Lieferung;
"Mengen Os" für die Objekte einer Lieferung

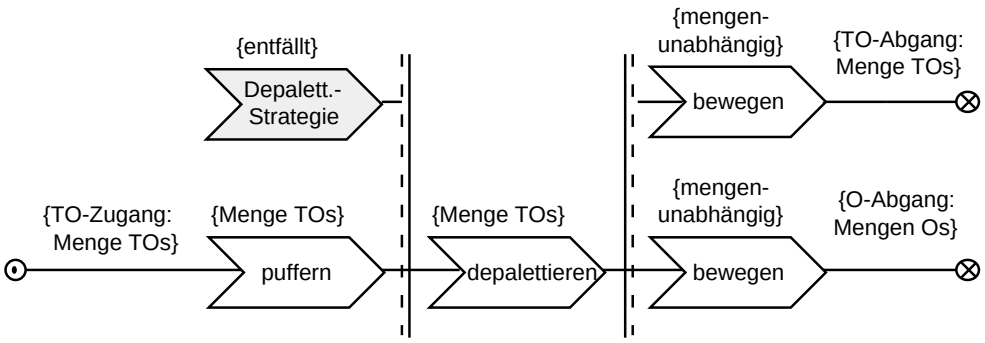


Abbildung 3-18

Wieder wird eine feste *Depalettierstrategie* (Paletten vollständig leeren) angenommen und kann damit als Einstellungsoption entfallen.

Im **Modellprozeß Lagerung** (Abbildung 3-19 nimmt DIST_BSP eine strukturelle Änderung gegenüber /SCHÜ99/ vor: Die Anlieferung beruht ja auf einem Prozeßträger (Leistungsobjekt) *Anlieferungsauftrag*, der sich bis zur physischen Einlagerung der angelieferten Objekte (Mengen Os) konsequent fortsetzt. Die Auslieferung besitzt einen davon zu unterscheidenden Prozeßträger *Auslieferungsauftrag*, welcher die physische Auslagerung anderer Objekte (Mengen Os) verursacht. Die in /SCHÜ99/ vorgenommene Fortsetzung des Einlagerungsprozesses, über eine Zwischenaktivität „lagern“, in den Auslagerungsprozeß erscheint daher nicht angemessen (diese Einschätzung müßte sich ändern, wenn individuelle Lieferobjekte als Prozeßträger angesehen würden, welche damit in ihrem „Leben“ tatsächlich eine Fortsetzung von Anlieferung, über Lagerung, zu Auslieferung

erführen - Lieferobjekte treten hier aber nicht als Individuen auf). Dieser Argumentation folgend sieht DIST_BSP zwei getrennte Modellprozesse *Einlagerung* und *Auslagerung* vor. Ein- und Auslagerungsmeldungen richten sich (siehe später) an die „Lokale Lagersteuerung“.

Modellprozeß Lagerung

"O" für „Objekt“
"Mengen Os" für Objekte einer An- bzw. Auslieferung

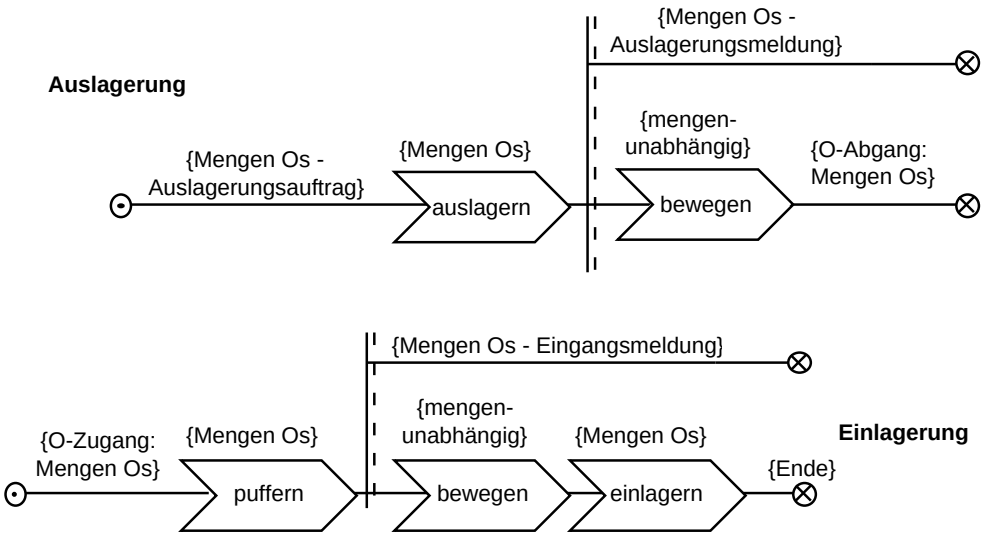


Abbildung 3-19

Modellprozeß Verpackung

DIST_BSP nimmt an, daß keine Verpackung stattfindet, so daß dieser Modellprozeß entfällt.

Modellprozeß Kommissionierung

"Mengen Os" für Objekte einer Auslieferung;
"Anzahl Os“ für Gesamtzahl Objekte einer Auslieferung

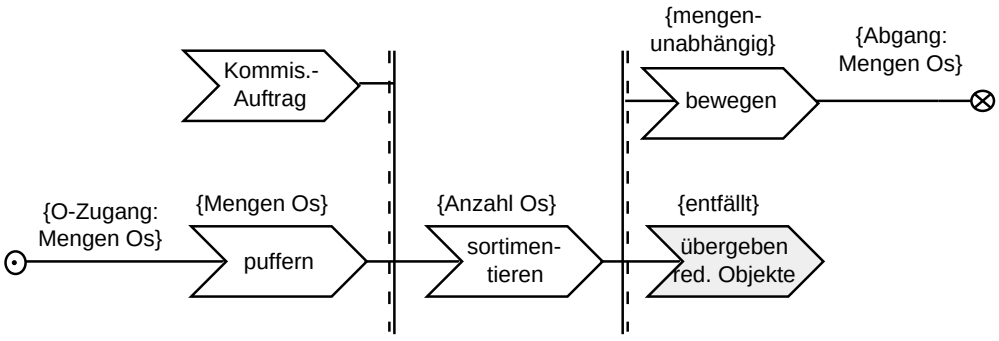


Abbildung 3-20

DIST_BSP nimmt an, daß die zu sortimentierenden Lieferobjekte direkt, also nicht als Teil größerer zusammengefaßter Objekte, zugeführt werden. Ein *Übergeben reduzierter Objekte* kann demnach entfallen.

Modellprozeß Palettierung

"TO" für "Träger-Objekt" , z.B. Palette;
"O" für Objekt;
"Menge TOs" für Paletten einer Auslieferung;
"Anzahl TOs“ für Zahl Paletten einer Auslieferung;
"Mengen Os" für Objekte einer Auslieferung

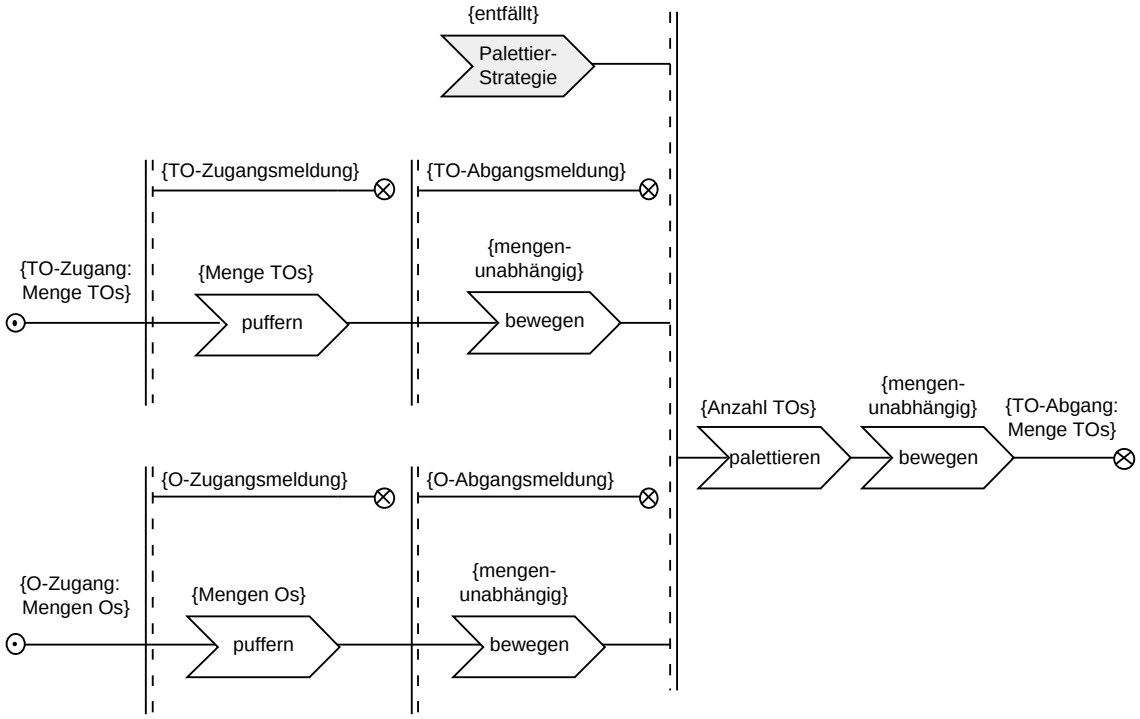


Abbildung 3-21

DIST_BSP nimmt eine feste *Palettierstrategie* (Paletten vollständig füllen) an, so daß diese als Option entfallen kann. TO-Zugangs- und Abgangsmeldungen richten sich (siehe später) an die „Lokale Behältersteuerung“, O-Zugangs- und Abgangsmeldungen (erahnt) an die „Lokale Lagersteuerung“.

Der (in /SCHÜ99/ nicht explizit dargestellte) **Modellprozeß Beladung** (Abbildung 3-22) ist in Analogie zu „Palettierung“ erstellt.

DIST_BSP nimmt eine feste *Beladestrategie* an (Fahrzeug mit einer Auslieferung beladen; Annahme war ja, daß ein Fahrzeug ausreicht, und daß ein Fahrzeug je Auslieferung eingesetzt wird). *Beladestrategie* als Option kann damit entfallen. Meldungen richten sich an die „Lokalen Behälter- und Fahrzeugsteuerungen“.

3.4.2.2 Die Ebene der Bausteine und ihrer Hauptprozesse

Die „Physikalischen Modellbausteine“ eines Standorts werden in /SCHÜ99/ in Form zugehöriger „Hauptprozesse“ beschrieben, zusammengesetzt aus den oben skizzierten „Modellprozessen“.

Hauptprozesse des Entlade- und Beladebausteins

Wegen des Verzichts auf „Auftrennung“ und „Verkettung“ sind diese „Hauptprozesse“ identisch zu den „Modellprozessen“ *Entladung* und *Beladung*.

Modellprozeß Beladung

"TTO" für "Träger_von_Träger-Objekt" , z.B. Fahrzeug;
"TO" für "Träger-Objekt", z.B. Palette;
"Menge TOs" für Paletten einer Auslieferung;
"Anzahl TOs“ für Zahl Paletten einer Auslieferung;

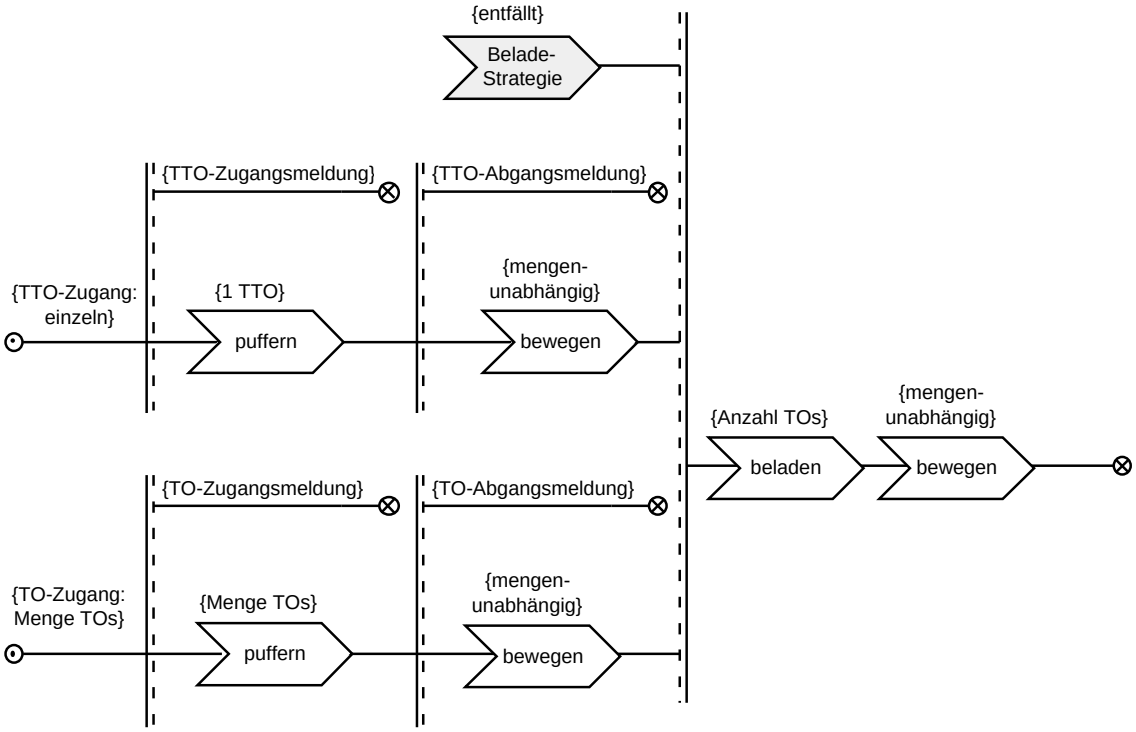


Abbildung 3-22

Hauptprozeß Lagerbaustein

"TO" für "Träger-Objekt", z.B. Palette;
"Menge TOs" für Paletten einer Ein- bzw. Auslieferung;
"Anzahl TOs“ für Zahl Paletten einer Auslieferung;

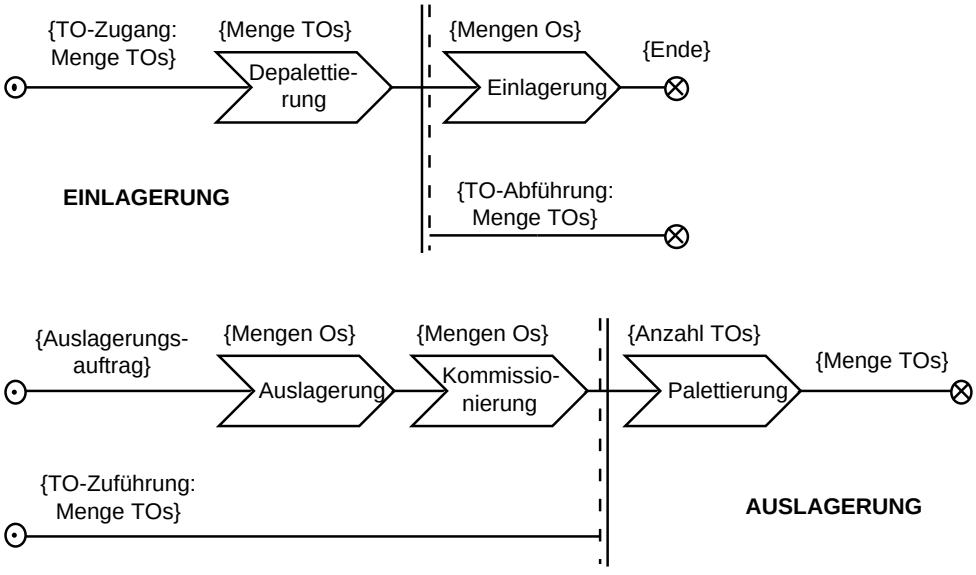


Abbildung 3-23

Der „Hauptprozeß“ Lagerbaustein (Abbildung 3-23) ist aus den „Modellprozessen“ *Depalettierung*, *Lagerung*, *Kommissionierung* und *Palettierung* zusammengesetzt.

Entsprechend der Bemerkungen zum „Modellprozeß Lagerung“ wird in DIST_BSP auch der „Hauptprozeß Lagerbaustein“ in zwei Prozeßketten *EINLAGERUNG* und *AUSLAGERUNG* aufgetrennt. In /SCHÜ99/ enthaltene Anpassungsoptionen „Bypässe“ (zum Überspringen von Modellbausteinen) werden in DIST_BSP nicht vorgesehen. Aus oben genannten Gründen entfallen auch Rücklagerungen reduzierter Objekte.

3.4.2.3 Die Ebene des Standortbausteins und seines Hauptprozesses

Über den Bausteinen und ihren Hauptprozessen ergibt sich, mit der nun schon bekannten Trennung von Ein- und Auslagerungsvorgängen, das Bild gemäß Abbildung 3-24. Gegenüber /SCHÜ99/ eingebrachte Vereinfachungen betreffen insbesondere die Nichtberücksichtigung ankommender und ausfahrender Leerfahrzeuge und, als Konsequenz, die unmittelbare Weiterreichung ankommender Fahrzeuge an die Entladung (ohne Entscheidung einer „Lokalen Fahrzeugsteuerung“).

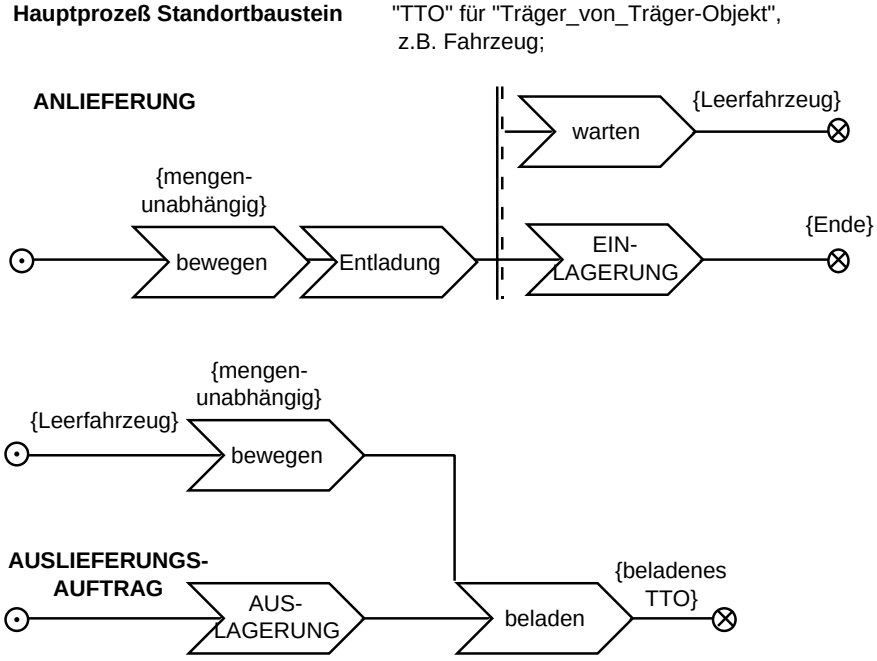


Abbildung 3-24

3.4.2.4 Die Ebenen der Globalen Steuerungen

DIST_BSP betrachtet den Distributions–“Standort“ (DSO) in Isolation. Seine Schnittstellen nach außen sind damit durch wechselseitig zusammenhanglose An- und Auslieferungen (bzw. die zugehörigen An- und Auslieferungsaufträge) gekennzeichnet. Außenwirkungen darüber hinaus (wie etwa Nachbestellungen von Objekten, Paletten und Fahrzeugen) werden nicht betrachtet. Konkrete Annahmen bzgl. der Charakteristika von An- und Auslieferungen haben die Resultate globaler Steuerungsentscheidungen zu repräsentieren. Diese Annahmen werden wie folgt getroffen (und in Abschnitt 5.4.2 wieder aufgenommen):

- Anlieferungen erfolgen in bestimmten zeitlichen Abständen *t_anl*, analog Auslieferungsaufträge mit zeitlichen Abständen *t_ausl*.

- DSO kennt und behandelt Lieferobjekte der Sorten A, B, C, etc. Anlieferungen und Auslieferungen erfolgen auf Fahrzeugen (Träger_von_Träger_Objekten, TTOs). Fahrzeuge transportieren Paletten

(Träger-Objekte, TOs), auf denen sich Lieferobjekte (Objekte, Os) befinden. Die An- und Auslieferungsmengen sind so bemessen, daß sich jede An- und Auslieferung (bzw. jeder entsprechende An- und Auslieferungsauftrag) auf ein einzelnes Fahrzeug bezieht. Fahrzeuge und Paletten sind als systemweit einheitlich (nicht weiter typisiert) angenommen.

· Eine Anlieferung ist durch (ein Einzelfahrzeug und) *p_an* Paletten mit insgesamt (*nA, nB,* …) Objekten der Sorten A, B, … gekennzeichnet, eine Auslieferung durch (ein Einzelfahrzeug und) *p_ab* Paletten mit insgesamt (*mA, mB,* …) Objekten der Sorten A, B, … . Die Gesamtzahl angelieferter Objekte betrage *n_an*, die der ausgelieferten Objekte *m_ab*, jeweils je Lieferung. Bei Annahme spezifischer Lade- und Palettierungsstrategien ergeben sich *p_an* bzw. *p_ab* u. U. auch als Funktionen der (*nA, nB,* …) bzw. (*mA, mB,* …).

Insgesamt hat die Betrachtung des Schürholz'schen Gesamtmodells zum Ziel, die Tragfähigkeit des im Folgenden zu entwickelnden B1-Modellformalismus in größerem Rahmen zu prüfen, als dies in den kleineren Beispielmodellen möglich erscheint.

4 Definition des B1-Modell-Formalismus

Mit der Entwicklung der sog. B-Paradigmen wird das Ziel verfolgt, einen Formalismus zur Beschreibung logistischer Systeme (insbesondere Großer Netze der Logistik) zu entwerfen, welcher direkt (also ohne händische Abbildungen auf die Spezifikationsmittel spezieller Modellierungswerkzeuge) für modellgestützte Analysen tauglich ist.

> Unter modellgestützten Analysetechniken wird insbesondere die Technik der ereignisorientierten Simulation subsumiert; darüber hinaus werden implizit die Charakteristika weiterer Analysetechniken algebraischen und numerischen Typs beachtet.

Zusätzlich zur prinzipiellen Analysetauglichkeit gelten als Ziele der B1-Entwicklung

- eine weitestgehende Nähe zum grundlegenden und umfassenderen A-Paradigma;

  > In diesem Sinne lehnen sich die B-Paradigmen soweit irgend möglich an die Begrifflichkeiten und Formalismen des A-Paradigmas an; auf diesen Umstand ist in den folgenden Abschnitten nur gelegentlich explizit hingewiesen.

- eine Ausdrucksfähigkeit, welche die unterschiedlichen konkreten Problemfelder der Anwendungsprojekte abdeckt.

  > Zu diesem Zwecke werden die in Entwicklung befindlichen B-Paradigmen fortlaufend mit Beispielen aus den Anwendungsprojekten konfrontiert und entsprechend angepaßt.

Dieses Kapitel beschreibt das B-Paradigma in seiner Version B1. Abschnitt 4.1 vermittelt einen begrifflichen Überblick, Abschnitt 4.2 erklärt die Modellierungs-Konstrukte des B1-Formalismus im einzelnen, sowohl ihrer Form (Syntax) als auch ihrer Verwendungsbedeutung (umgangssprachliche Semantik) nach.

> Wie in der vorliegenden Einleitung wird im folgenden Abschnitt 4.1 die Form des Kleindrucks eingesetzt, um (zusätzlich zur Hauptlinie) Ergänzungen, Nebengedanken, Verständnishilfen zu präsentieren.

4.1 Überblick

In Anlehnung an das A-Paradigma spielen in der Begriffswelt des B1-Paradigmas **Prozesse**, **Prozeßketten** und **Prozeßkettenpläne** eine zentrale Rolle. Daneben werden **Funktions- / Organisations–Einheiten** als explizite "Akteure" des Paradigmas eingeführt. Eine weitestgehende Strukturierbarkeit von Modellen, in den Formen der **Modularisierung** und der **Hierarchisierung**, wird durchgehend angestrebt.

Eine **Prozeßkette** (**PK**) definiert die Verhaltens- (Ablauf-) Vorschriften für Prozesse einer spezifischen Art, für eine Familie von Prozessen.

> Sind unterschiedliche Prozeß-Arten zu erfassen, werden sie durch unterschiedliche Prozeßketten definiert. Es scheint sinnvoll, PKs mit unterscheidbaren Namen versehen zu können, sowie einzuräumen, daß Prozeßketten (zumindest potentiell) zusätzlich parametrisierbar sind.

Eine Prozeßkette spielt die Rolle eines "Prozeßmusters", einer Vorschrift, die für jeden Prozeß dieser Familie gilt. Zu diesem Zweck

- benennt die Prozeßkette alle **Aktivitäten**, welche von Prozessen dieser Familie absolviert werden können;

  > Die Bezeichnung "Aktivität" wird hier als Synonym für Begriffe wie "Teilprozeß", "Prozeßschritt", "Prozeßteilschritt", "Vorgang" o. ä. verwendet. Es scheint sinnvoll, den verschiedenen Aktivitäten Namen zuzuordnen, sowie davon auszugehen, daß sie (zumindest potentiell) zusätzlich parametrisierbar sind.

- hält die Prozeßkette alle Bedingungen fest, welche für die **Reihenfolge** des Absolvierens von Aktivitäten in dieser Familie gelten;

  Solche Bedingungen können verschiedenster Art sein; eingeschlossen sind insbesondere

  die "sequentiellen Abfolgen" und "alternativen Fortsetzungen", wonach z. B. eine Aktivität a abgeschlossen sein muß, bevor eine (andere) Aktivität b in Angriff genommen werden kann; oder, allgemeiner, eine Aktivität aus einer gewissen Menge a1,a2,… von Aktivitäten abgeschlossen sein muß, bevor eine Aktivität aus einer (anderen) Menge b1,b2,… von Aktivitäten in Angriff genommen werden kann, und um welche es sich unter den gegebenen Umständen handelt; u. a. m.

  die "Parallelisierungen" und "Synchronisierungen", wonach z. B. nach Abschluß einer Aktivität a die Aktivitäten einer bestimmten Aktivitätsmenge b1,b2,… (allesamt) in Angriff genommen werden können; oder daß alle Aktivitäten einer gewissen Menge a1,a2,… abgeschlossen sein müssen, bevor eine Aktivität b begonnen werden kann; u. a. m.

- ist im Kontext jeder Prozeßkette zu spezifizieren, unter welchen Umständen individuelle **Prozesse** der Familie **entstehen** sowie (im Falle parametrisierter Prozeßketten) zusätzlich, welche Parameterwerte die entstandenen Prozesse jeweils individuell erhalten;

  Umstände des Entstehens können verschiedenster Art sein; zu denken ist insbesondere an Fälle

  des "unbedingten" Entstehens, z. B. gesteuert durch Angabe einer Menge von (vom Modellablauf unabhängigen) Zeitpunkten, zu denen individuelle Prozesse einer bestimmten Prozeßkette entstehen / beginnen / ins Leben treten / inkarniert werden; oder gesteuert durch Charakterisierung von Zeitintervallen, welche zwischen aufeinanderfolgenden Zeitpunkten der Inkarnierung von Prozessen einer Prozeßkette verstreichen; zu berücksichtigen ist auch die gleichzeitige Inkarnierung einer bestimmten Anzahl von Prozessen sowie diverse stochastische Festlegungen (so insbesondere im Falle der intervallgesteuerten Inkarnierung).

  der "bedingten" Entstehung von Prozessen, wonach individuelle Prozesse einer bestimmten Prozeßkette nicht zu vorbestimmtem Zeitpunkten, sondern in Abhängigkeit vom dynamischen Ablauf / bei Vorliegen bestimmter Bedingungen inkarniert werden.

- sind im Kontext einer Prozeßkette die Umstände der **Beendigung** individueller **Prozesse** der Familie festzulegen;

  Hier sei angemerkt, daß in Anlehnung an das A-Paradigma nur Prozeßketten betrachtet werden, welche ausschließlich "temporäre Prozesse" beschreiben (also solche Prozesse, die definitiv und ohne Alternative ein Ende ihres Ablaufs erreichen); die alternativen "permanenten Prozesse" (welche mit Sicherheit oder zumindest möglicherweise kein spezifiziertes Ende erreichen) bleiben zunächst unbeachtet.

  Auch die Umstände der Beendigung können verschiedenster Art sein; eingeschlossen sind insbesondere Fälle

  der "impliziten Beendigung" von Prozessen, welcher Fall dann auftritt, wenn im Ablauf eines Prozesses (gemäß der Reihenfolgevorschriften seiner Prozeßkette) keine der Aktivitäten mehr in Angriff genommen werden kann ("Erschöpfung der Aktivitäten");

  der "expliziten Beendigung" von Prozessen, welcher Fall dann auftritt, wenn in der Prozeßkette ein unbedingter Endpunkt für zugehörige Prozesse spezifiziert ist, und ein entsprechender individueller Prozeß diesen Endpunkt (gemäß der Reihenfolgevorschriften seiner Prozeßkette) erreicht.

- beinhaltet die Prozeßkette u. U. **weitere Angaben** und Festlegungen, welche im Moment unberücksichtigt bleiben.

  Weitere Angaben und Festlegungen werden aber in der Folge auftreten.

Ein **Prozeß** ist eine dynamische Einheit, welche einen individuellen Ablauf nach Vorschriften einer bestimmten Prozeßkette erfaßt.

Ein Prozeß entsteht / wird inkarniert, absolviert Aktivitäten / schreitet fort, endet, alles gemäß der Festlegungen seiner Prozeßkette (s. o.). Die Abläufe unterschiedlicher individueller Prozesse einer Prozeßkette können sich (im Rahmen der Festlegungen dieser Prozeßkette) unterscheiden. (Darüber hinaus unterscheiden sich natürlich die Abläufe von Prozessen unterschiedlicher Prozeßketten.)

In Anlehnung an das A-Paradigma ist die Festlegung der "Umstände des Entstehens von Prozessen" weitestgehend als Bestandteil einer Prozeßkette aufgefaßt. Alternativ dazu läßt sich die Auffassung vertreten, der (dynamische) Vorgang des Inkarnierens / Startens individueller Prozesse (gemäß spezifischer Prozeßketten) sei durch einen separaten Prozeß (oder separate Prozesse) zu erfassen. Diese Auffassung führt zu einem etwas allgemeineren Bild (so wird damit insbesondere das Inkarnieren von Prozessen einer Prozeßkette "von verschiedenen Stellen aus" ermöglicht). Analoge Bemerkungen betreffen die "Umstände der Beendigung von Prozessen" als (hier:) Teil einer Prozeßkette bzw. (alternativ:) als separater Prozeß.

Mit der Vorstellung eines Prozesses als individueller Ablauf intuitiv verknüpft ist die Vorstellung eines diesbezüglichen "Prozeßträgers" (vgl. die "Leistungsobjekte" des A-Paradigmas). Prozesse einer Prozeßkette erfassen damit das individuelle Fortschreiten von Prozeßträgern einer bestimmten Art - Prozesse unterschiedlicher Prozeßketten das von Prozeßträgern unterschiedlicher Art. Problemseitig möglicherweise existierende wechselseitige Abhängigkeiten im Ablauf verschiedener individueller Prozesse sind in dieser intuitiven Deutung nicht von vornherein enthalten und bedürfen der expliziten Definition einer diesbezüglichen "Prozeßkettenstruktur".

Ein **Prozeßkettenplan** notiert die Spezifika einer Prozeßkette (bzw. mehrerer PKs, bzw. einer komplexeren Prozeßkettenstruktur) mit graphischen und textuellen Mitteln. Das A-Paradigma enthält hierzu einige Vorgaben (die im Verlauf eingestreuten Beispiele greifen diese Vorgaben zur Illustration auf, entfernen sich in ihrer Form aber bereits leicht von den Vorgaben). So umfassen die **graphischen** Beschreibungsmittel für Prozeßketten

· **Prozeßkettenelemente** zur Erfassung der Aktivitäten;

Beispiel „Prozeßkettenelement“
- (Aktivitäts-)Name: bearbeiten
- Parameter-Wert: 3.5
  (interpretiert als „Vorgabezeit“)

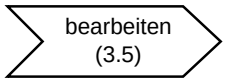


Abbildung 4-1

· verschiedene **ablauflogische Elemente** zur Notation von Reihenfolgevorschriften zwischen Aktivitäten, so

- die **Verbindung** zwischen jeweils 2 Aktivitäten zur Kennzeichnung ihrer sequentiellen Abfolge;

Beispiel „sequentielle Abfolge“
- Aktivität: bearbeiten, P-Wert: 3.5
- vor: prüfen, P-Wert: 1.8

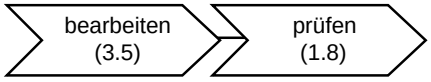


Abbildung 4-2

- **Konnektoren** samt Verbindungen zwischen (beiderseits) Aktivitäts-Teilmengen zur Erfassung alternativer bzw. gemeinsamer Fortsetzungen, sowie parallelisierender und synchronisierender Reihenfolgevorschriften;

Beispiel „alternative Fortsetzung“
- nach: prüfen (1.8)
- entweder: nachbearbeiten (6.0), falls ok
- oder: … („etwas anderes“), andernfalls

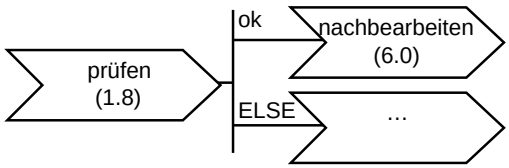
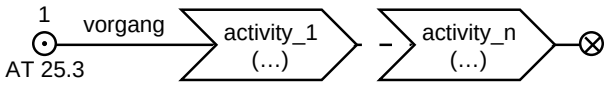


Abbildung 4-3

· **Quellensymbole** bzw. **Senkensymbole** samt Verbindungen zu bzw. von Prozeßkettenelementen zur Erfassung der Entstehungs- und Beendigungspunkte von Prozessen sowie von Angaben zur Entstehung und Beendigung von Prozessen.

Beispiel „Quelle und Senke“



- bezüglich: Prozeßkette namens vorgang
- Erstaktivität: activity_1 (…),
- Letztaktivität: activity_n (…),
- inkl. Startfestlegung:
  1 individueller Prozeß,
  zum Zeitpunkt 25.3

Abbildung 4-4

Zusätzlich dienen diverse Benennungen, Beschriftungen, Attributierungen der genannten Symbole der **textuellen** Ergänzung der Festlegungen einer Prozeßkette.

> In den Beispielen ist von einigen möglichen Benennungen und Beschriftungen bereits in verschiedener Weise Gebrauch gemacht.
>
> Abschnitt 4.2 wird die (in Anlehnung an das A-Paradigma) gewählten Symbole und Beschriftungen des B1-Formalismus im einzelnen beschreiben.

Es ist angestrebt, Prozeßkettenpläne direkt als Ausgangspunkt modellgestützter (automatischer) Analysen einzusetzen (s. o.), ihnen also die Rolle von analysierbaren Modellen zuzuweisen. Die Erreichung dieses Ziel setzt voraus, daß Prozeßketten alle für eine Analyse erforderlichen Informationen enthalten.

> Welche Informationen hier konkret erforderlich sind, ist einerseits von den festzulegenden Analysezielen (d. h. den zu ermittelnden Systemeigenschaften), andererseits von den zur Analyse eingesetzten Analysetechniken abhängig. Sicherlich füllen die erwähnten vorgegebenen Benennungen, Beschriftungen, Attributierungen von Prozeßketten-Symbolen bereits gewisse Teilaspekte dieser Aufgabe aus. Man kann allerdings sicher nicht davon ausgehen, daß damit bereits "alle erforderlichen" Informationen für "alle denkbaren" Analyseziele / Analysetechniken vorliegen. Prozeßkettenpläne werden fallweise (und abhängig von Analysezielen und Analysetechniken) mit zusätzlichen "geeigneten" Informationen "anzureichern" sein.

In einer ersten Konzentration auf **leistungsorientierte** Beurteilungen, also auf Analyseziele wie Zeitdauern, Durchsätze, Auslastungen etc., bedient sich der B1-Formalismus hinsichtlich der erforderlichen Informationsanreicherung von Prozeßkettenplänen der im folgenden skizzierten Sicht.

> Es wird sich herausstellen, daß diese Sicht beschreibungs- und analysefreundliche Hierarchisierungsmöglichkeiten für Prozeßketten impliziert.

· Eine Prozeßkette definiert die Verhaltens-Vorschriften für eine spezifische Prozeß-Familie (s. o.). Sie benennt insbesondere eine Reihe von Aktivitäten, welche von zugehörigen Prozessen je nach Prozeß-Ablauf zu absolvieren sind. Das Absolvieren einer Aktivität wird als **Ausführung** dieser Aktivität verstanden.

> Eine Prozeßkette beschreibt demnach, **was** im Verlauf des Ablaufs zugehöriger Prozesse geschehen soll. Sie beschreibt die Durchführung von Prozessen, aufgelöst in Form einer geordneten Ausführung von Aktivitäten.

· Ein Prozeßkettenplan beschreibt eine Prozeßkette (s. o.). Er weist zusätzlich eine Reihe von **Funktionseinheiten** / Organisationseinheiten (kurz: FEs) aus, verstanden als Repräsentanten von Systembestandteilen, welche die Ausführung der Aktivitäten ablaufender Prozesse ermöglichen und unterstützen.

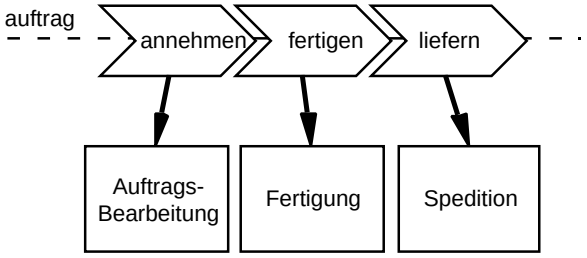
> Funktionseinheiten spielen damit die Rolle von "Akteuren". Im Vergleich zum A-Paradigma sollen FEs insbesondere die dortigen (Potential-)Ressourcen und Ressourcen-Pools umfassen, deren Rolle aber verallgemeinern.

· Die Begleitumstände und Auswirkungen von Aktivitäts-Ausführungen werden dadurch erfaßt, daß jeder Aktivität (genau) eine Funktionseinheit zugeordnet ist, welcher die Ausführung / Durchfüh-

rung der Aktivität obliegt. **Aktivitäten** werden zum Zwecke ihrer Ausführung **an** eine **Funktionseinheit verwiesen**.

Indem FEs insbesondere die Ausführung von Aktivitäten übernehmen, bestimmen sie, **wie** diese Ausführungen geschehen Detailliertere Vorstellungen zum Charakter der Ausführung von Aktivitäten durch Funktionseinheiten werden im weiteren Verlauf vorgestellt und präzisiert.

Beispiel „Aktivitäten und Funktionseinheiten“



- Aktivitäten der Prozeßkette namens auftrag:
  annehmen, fertigen, liefern
- zur Ausführung verwiesen an Funktionseinheiten:
  Auftragsbearbeitung, Fertigung, Spedition

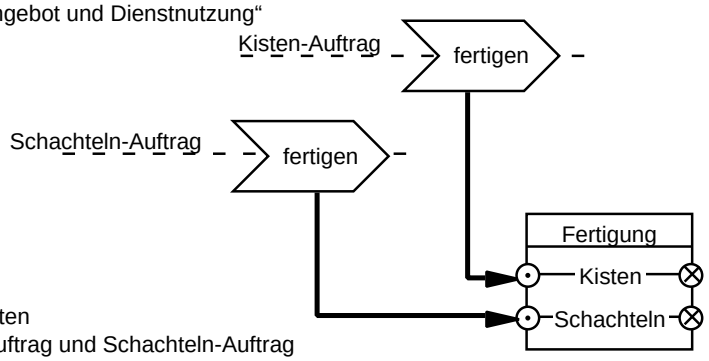
Abbildung 4-5

· Eine FE zeigt an, daß sie zur Ausführung einer bestimmten Aktivitätsart in der Lage ist, indem sie (Sprechweise:) einen diesbezüglichen **Dienst anbietet**.

Sinnvollerweise wird eine Funktionseinheit so gestaltet sein, daß sie in der Lage ist, eine ihr zugeordnete Aktivität tatsächlich durchzuführen. Sie sollte damit auch andere "gleichartige" Aktivitäten durchführen können, so daß ein Verweis unterschiedlicher gleichartiger Aktivitäten an ein und dieselbe Funktionseinheit sinnvoll wird (problemseitiger Bedarf selbstredend vorausgesetzt).

Es erscheint darüber hinaus sinnvoll, Funktionseinheiten zuzulassen, welche in der Lage sind, unterschiedliche (nicht gleichartige) Aktivitäten durchzuführen, d. h. **mehrere** unterschiedliche **Dienste** anzubieten. Daraus resultiert hinsichtlich des Verweisens einer PK-Aktivität an eine Funktionseinheit die Notwendigkeit der zusätzlichen Bezeichnung eines konkreten Dienstes (unter den Dienstangeboten der FE).

Beispiel „Dienstangebot und Dienstnutzung“



- zwei Prozeßketten
  Kisten-Auftrag und Schachteln-Auftrag
- nutzen je einen Dienst der Fertigung
  Kisten bzw. Schachteln {fertigen}

Abbildung 4-6

· Die **Durchführung von Aktivitäten** obliegt den Funktionseinheiten (FEs). Der Charakter / die Art der Durchführung ist somit sinnvollerweise "in" den, "durch" die FEs geregelt, ist Teil ihrer Definition / Spezifikation.

- Einfachere, wiederkehrende Durchführungsregeln lassen sich durch Definition einer Reihe von Standard-FE-Typen berücksichtigen, wobei Parametrisierungen sowohl der Standard-Typen, als auch der von ihnen angebotenen Dienste und analog der (Prozeßketten-)Aktivitäten zur Flexibilität ihres Einsatzes beitragen.

  Im Folgenden werden für den B1-Formalismus insbesondere zwei Standard-FE-Typen vorgeschlagen, die sog. "Server"- und "Counter"-Typen. Je nach Bedarf der verschiedenen Anwendungsprojekte wird die Liste der Standards zu erweitern sein.

- Komplexere, nicht wiederkehrende Durchführungsregeln bedürfen der problemangepaßten Spezifikation im Einzelfall. Im Rahmen des B1-Formalismus werden diese Durchführungsregeln in Form von Prozeßketten erfaßt und in Prozeßkettenplänen notiert. Vgl. Abbildung 4-7.

  Konkreter gesprochen: Jede Aktivität einer Prozeßkette ist (s. o.) einem spezifischen Dienst einer spezifischen FE zugeordnet. Die Durchführung dieses Dienstes ist (innerhalb dieser FE) durch eine "niedrigere, detailliertere" Prozeßkette festgelegt (d. h. durch "niedrigere" Aktivitäten samt zugehöriger Reihenfolgevorschriften beschrieben). Im Rahmen eines ablaufenden (höheren) Prozesses ist somit die Ausführung jeder seiner Aktivitäten durch Ablauf eines spezifischen (niedrigeren) Prozesses, gemäß (niedrigerer) Prozeßkette, innerhalb der zugeordneten FE erfaßt. Die Durchführung der niedrigeren Aktivitäten (im Rahmen des Ablaufs niedrigerer Prozesse) obliegt niedrigeren FEs.

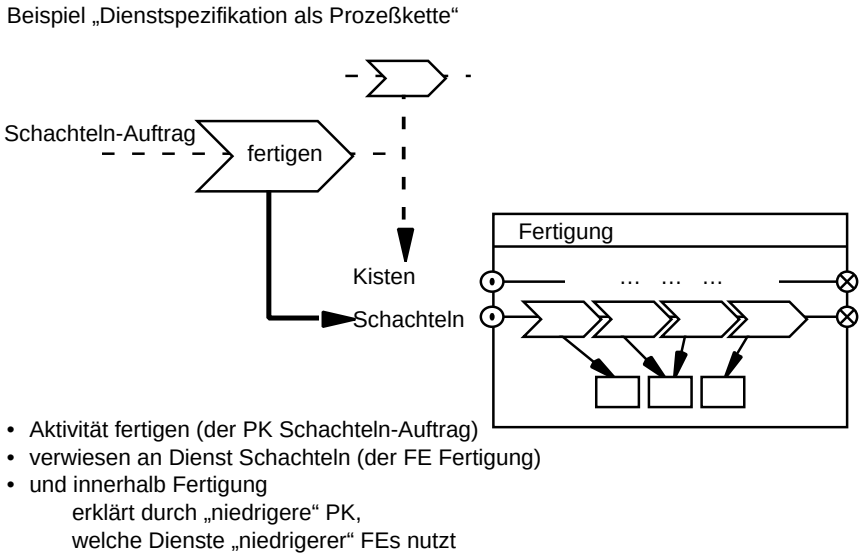


Abbildung 4-7

· Die Sicht des B1-Formalismus (wie oben skizziert) führt implizit verschiedene (gleichzeitig vorliegende) Typen der **hierarchischen Strukturierung** von Prozeßkettenplänen ein (bzw. verschiedene Typen von "Selbstähnlichkeit"), welche in gewisser Weise über die entsprechenden Möglichkeiten des A-Paradigmas hinausgehen.

- Einerseits wird jeder Prozeß in seine Aktivitäten (Teilschritte) aufgelöst, wie durch "seine" Prozeßkette festgelegt. Aktivitäten ihrerseits werden an Dienste von Funktionseinheiten zur Ausführung verwiesen, diese Ausführungen (innerhalb der Funktionseinheiten) durch Prozeßketten beschrieben. Es resultiert die weithin bekannte "**Aufruf-Hierarchie**": Prozeßketten rufen / benutzen Prozeßketten, welche wiederum Prozeßketten aufrufen / benutzen …

- Andererseits können innerhalb von Funktionseinheiten (niedrigere, im Ablauf angebotener Dienste zu nutzende) Funktionseinheiten liegen, welche ihrerseits erneut Funktionseinheiten enthalten können, … Es resultiert die gleichfalls weithin bekannte "**Enthaltensein-Hierarchie**", geeignet zur Erfassung (gewisser) struktureller Gliederungen des beschriebenen Systems.

Obwohl nicht auf den ersten Blick naheliegend, ist die Gliederungsstruktur nicht auf Baumstrukturen reduziert.

4.2 Modellierungskonstrukte des B1-Formalismus

Dieser Abschnitt stellt die Modellierungskonstrukte des B1-Formalismus im einzelnen vor. Im Hinblick auf eine zu entwickelnde graphisch/textuelle Benutzungsoberfläche ("GUI") eines Spezifikationswerkzeugs für B1-Modelle wird dabei versucht, für alle Angaben einer Spezifikation "geeignete" Orte auf einer erahnten Zeichenfläche anzugeben. Dies soll nicht implizieren, daß ein solches GUI zwingend alle Informationen gleichzeitig zeigen muß; vielmehr werden aus Übersichtlichkeitsgründen verschiedene Sichten ("views") auf ein Modell und seine Teile vorzusehen sein.

4.2.1 Allgemeines: Namen, Kommentare, Attribute, Parameter, Variablen

Eine Reihe der B1-Modellierungskonstrukte tragen **Namen** bzw. können Namen tragen. Bei Vorstellung der konkreten Konstrukte wird auf Verpflichtung bzw. Optionalität der Benennung im einzelnen hingewiesen werden. Abschnitt 4.2.6 wird Fragen der Eindeutigkeit und Sichtbarkeit (scoping) von Namen behandeln. Als Konvention gilt: Namen beginnen mit einem Buchstaben und setzen sich aus Buchstaben, Ziffern und Unterstrichen zusammen. Beispiel:

Gültiger_Name3

Verschiedene B1-Konstrukte können mit erklärenden **Kommentaren** versehen werden: Bei Vorstellung der konkreten Konstrukte wird auf diese Möglichkeit im einzelnen hingewiesen werden. Als Konvention gilt: Kommentare werden in geschweiften Klammern dargestellt. Beispiel:

{Dies ist ein Kommentar}.

Diverse B1-Konstrukte verlangen zur Festlegung ihrer konkreten Ausprägung die Setzung konstruktspezifischer **Attribute**. Die Setzung von Attributen geschieht im Verlauf der Spezifikation eines Modells; sie ist vor dem dynamischen Ablauf eines Modells (bzw. vor der Analyse eines Modells) abzuschließen; Attribute ändern ihre Werte während des Modellablaufs nicht.

Im Kontext verschiedener B1-Konstrukte ist (meist optional) der Zugriff auf Parameter und Variablen vorgesehen. Diese sind in gewohnter Weise typisiert, wobei die Menge zulässiger Datentypen noch genau zu spezifizieren sein wird. In der vorliegenden Darstellung werden die Typen REAL (reelle Zahlen), INT (ganze Zahlen), STRING (Zeichenketten), BOOL (Wahrheitswerte) und INT_VEK (Vektoren ganzer Zahlen) Verwendung finden.

Parameter dienen generell der Übergabe von Informationen; dabei wird unterschieden zwischen

- ("normalen", "Aufruf-") **Parametern**, welche eine Übergabe von Informationen während des dynamischen Ablaufs eines Modells ermöglichen, und welche ihre Werte während dieses Ablaufs geeignet ändern (können); man denke an die gewohnten Funktions- und Prozedur-Parameter der diversen Programmiersprachen;

- **Attributparametern** ("Initiierungsparametern", "Generierungsparametern"), welche der Übergabe von Informationen aus einer Analyseumgebung an ein zu analysierendes Modell dienen; Attributparameter werden vor Beginn einer Modellanalyse gesetzt; ihre Werte bleiben während des dynamischen Modellablaufs konstant; man denke an die Spezifikation eines Modell "in verschiedenen optionalen Varianten".

Der Term **Variable** wird in seiner üblichen Bedeutung verwendet. Variablen und (normale) Parameter besitzen jeweils einen konkreten Gültigkeitsbereich; eine diesbezügliche Konkretisierung erfolgt in Abschnitt 4.2.6.3. Variablen erfassen das "Gedächtnis" (den Datenraum / Zustandsraum) der diversen B1-Konstrukte.

4.2.2 Einfache Prozeßketten

Eine Prozeßkette definiert die Verhaltensvorschriften einer Familie "gleichartiger" Prozesse (vgl. 4.1). Der vorliegende B1-Formalismus erfaßt

- Aktivitäten in Form von **Prozeßkettenelementen** (PKEs),
- Reihenfolgefestlegungen in Form von **Verbindungen** und **Konnektoren**,
- Umstände der Entstehung und Beendigung von Prozessen mittels **Quellen** und **Senken**.

Kennzeichnend für die hier (zunächst) erfaßten **einfachen Prozeßketten** ist, daß ihre Prozesse einen einzelnen eindeutigen Prozeßträger (ein eindeutiges "Leistungsobjekt") besitzen. Die ablaufbeschreibende Prozeßkette weist aus diesem Grunde ein eindeutiges "erstes Element" (ein PKE oder einen Konnektor) auf sowie ein eindeutiges "letztes Element" (PKE oder Konnektor). Die Entstehung (bzw. Beendigung) von Prozessen wird dadurch ausgedrückt, daß eine Quelle mit diesem ersten Element verbunden wird (bzw. das letzte Element mit einer Senke verbunden wird).

Eine Prozeßkette als Ganzes kann mit einer Reihe von Attributen versehen werden, welche an jener Verbindung notiert werden, welche (nach Fertigstellung der Spezifikation: von einer Quelle aus) auf das erste Element der PK zuführt.

Symbol:

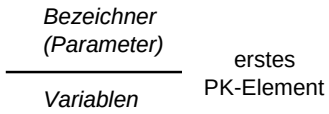


Abbildung 4-8

Attribute:

- *Bezeichner* (fallweise optional oder zwingend): Der **Name** dieser PK, zur Erhöhung der Lesbarkeit zweckmäßigerweise "sprechend" gewählt (z. B. verständliche Benennung des Gesamtvorgangs, der durch diese PK beschrieben ist);
- *Parameter* (optional): Name und Typ, ggf. Liste von Namen und Typen, von **Parametern**, welche jeden individuellen Prozeß (im Rahmen dieser Prozeßkette / Prozeßfamilie) genauer charakterisieren werden; dies sind "formale Parameter" der Prozeßkette; anläßlich der Entstehung individueller Prozesse werden diese Parameter mit "aktuellen Werten" zu belegen sein (eine Aufgabe der Quellen und anderer Konstrukte mit Quellencharakter).
- *Variablen* (optional): Name und Typ, ggf. Liste von Namen und Typen, von **Variablen**, welche jedem individuellen Prozeß dieser Prozeßkette als individueller Datenraum / Zustandsraum zur Verfügung stehen. Variablen können optional initialisiert werden. Eine Variablen-Vereinbarung

  k:INT INIT 0

  vereinbart beispielsweise eine ganzzahlige Größe *k*, welche initial den Wert 0 besitzt.

Parameter und Variablen einer PK sind Prozessen dieser PK während ihres ganzen dynamischen Ablaufs zugreifbar, wobei jeder individuelle Prozeß seinen individuellen Satz von Parametern und Variablen besitzt (und diese außerhalb dieses Prozesses nicht sichtbar sind).

Beispiel (siehe Abbildung 4-9):
Ein (im weiteren Verlauf dieser PK zu beschreibender) Beladevorgang, der jeweils bei Entstehung durch die Zahl n zu ladender Teile charakterisiert wird, und der über eine Variable z verfügt, um sich

(etwa für den Fall schrittweiser Beladung) zu "merken", wieviele der zu ladenden Teile noch zur Ladung anstehen.

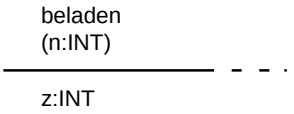


Abbildung 4-9

4.2.2.1 Das Prozeßkettenelement (PKE)

Für das PKE sind zwei Grundformen vorgesehen,

- eine Grundform zur Beschreibung von zeitverbrauchenden Aktivitäten, genauer: von Aktivitäten, welche zu ihrer Ausführung zumindest potentiell endliche (Modell-) Zeit benötigen (Aktivitäten "physischer Art"),
- eine weitere Grundform zur Beschreibung zeitloser Aktivitäten, genauer: von Aktivitäten, deren Ausführung keine (Modell-)Zeit benötigt. (Aktivitäten "logischer Art").

4.2.2.1.1 Das zeitbehaftete PKE

Symbol:

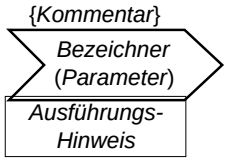


Abbildung 4-10

Attribute:

- *Kommentar* (optional): Verbale Bemerkungen zu diesem PKE, zur Erhöhung der Lesbarkeit und Verständlichkeit;
- *Bezeichner* (optional): Der Name dieses PKE, zur Erhöhung der Lesbarkeit möglichst "sprechend" gewählt (z. B. verständliche Benennung der auszuführenden Aktivität);
- *Parameter*: Wert (bzw. Ausdruck), ggf. Liste von Werten / Ausdrücken, welche die auszuführende Aktivität genauer charakterisieren (z. B. Umfang, Größe, … der Aktivität); dies sind aktuelle Parameter, welche auf die Angaben im *Ausführungshinweis* abzustimmen sind;
- *Ausführungshinweis*: Gemäß 4.1 wird eine Aktivität von einer Funktionseinheit ausgeführt, muß von einem PKE auf einen Dienst einer ausführenden FE "gezeigt werden"; aus Gründen der (graphischen) Übersichtlichkeit wird diese Zuordnung durch einen textuellen Verweis ausgedrückt, im Standardfall in der Form

  Funktionseinheit . Dienst (im Standardfall beides: Namen)

  (Zahl und Typ der Parameter der PKE-*Parameter*-Liste müssen mit denen des benutzten Dienstes übereinstimmen; dies wird im Kontext der Beschreibung von Funktionseinheiten - siehe Abschnitt 4.2.3 - verständlicher werden.)

  Zusätzlich zu diesem Standardfall werden aus Gründen der Erleichterung der Modellspezifikation eine Reihe von Sonderfällen eingeführt (werden). Einer dieser Sonderfälle ist der (recht häufig benötigten) Beschreibung von Aktivitäten gewidmet, deren Ausführung "immer und unter allen Um-

ständen" eine im voraus bekannte Zeit benötigt, also eine reine Verzögerung vorbekannter Dauer in den Ablauf eines Prozesses einfügt. Dieser Sonderfall ist durch das Schlüsselwort DELAY im *Ausführungshinweis* und der Dauer der Verzögerung (in Zeiteinheiten) als *Parameter* beschrieben.

Beispiel:

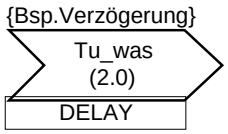


Abbildung 4-11

4.2.2.1.2 Das zeitlose PKE

Symbol:

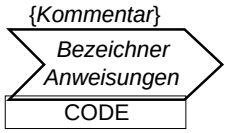


Abbildung 4-12

Attribute:

- *Kommentar* (optional): wie oben;
- *Bezeichner* (optional): wie oben;
- *Anweisungen*: Code einer "normalen" Programmiersprache (welche noch genau festzulegen sein wird); dabei Möglichkeiten des Lesens und Schreibens zugreifbarer Größen (zumindest also der Parameter und Variablen dieser PK; Zugriffsmöglichkeiten auf weitere Größen werden im weiteren Text aufscheinen); zur Erinnerung: Die Ausführung dieser Anweisungen benötigt keine (Modell-)Zeit.
- CODE: Schlüsselwort im "Hinweis"-Feld zur Charakterisierung dieses Sonderfalls.

Beispiel:

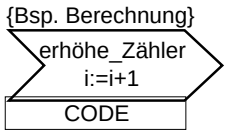


Abbildung 4-13

4.2.2.2 Verbindungen und Konnektoren

Sollen Prozesse einer PK zwei Elemente (PKEs oder Konnektoren) direkt aufeinanderfolgend (sequentiell) "besuchen", wird dies durch eine **Verbindung** der Elemente graphisch ausgedrückt; Verbindungen dienen zusätzlich der Kopplung von Quelle und erstem PKE (bzw. letztem PKE und Senke) - vgl. oben; **Konnektoren** ermöglichen die Charakterisierung alternativer, gemeinsamer, paralleler und synchronisierter Prozeßfortsetzungen.

4.2.2.2.1 Die Verbindung

Symbol:

Abbildung 4-14

Attribute: keine

(Fallweise aufscheinende Beschriftungen längs von Verbindungen charakterisieren Attribute anderer Modellierungskonstrukte, z. B. der gesamten PK, oder der verbundenen Elemente.)

Beispiel:

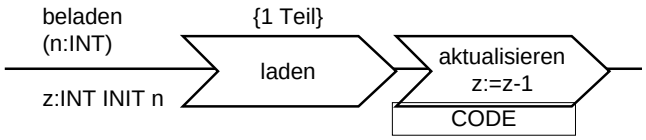


Abbildung 4-15

Im Ablauf: Laden eines ersten Teils (hier noch nicht vollständig spezifiziert), sequentiell gefolgt von einer Aktualisierung der Ladungs"restes".

4.2.2.2.2 Öffnende und Schließende ODER-Konnektoren

Öffnende ODER-Konnektoren ermöglichen eine alternative ("entweder oder") Fortsetzung von Prozessen, Schließende ODER-Konnektoren ermöglichen die Zusammenführung (und damit gemeinsame Fortsetzung) alternativer Zweige. ODER-Konnektoren sind im fertigen Modell beidseitig mit Elementen (PKEs oder Konnektoren) verbunden (sind Nachfolger vorausgehender PK-Abschnitte und Vorgänger nachfolgender PK-Abschnitte).

Symbol Öffnender ODER-Konnektor:

(zwei unterschiedliche Attributierungen, jeweils an den Ausgangsverbindungen notiert; eine Mischung der Attributierungen ist nicht zugelassen):

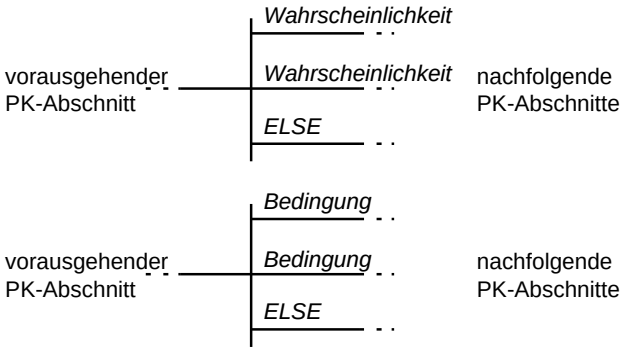


Abbildung 4-16

Attribute erster Fall:

- *Wahrscheinlichkeit*: Wert (arithmetischer Ausdruck) < 1, Wahrscheinlichkeit, mit der eine Fortsetzung des Prozesses entlang des bezeichneten Nachfolgerabschnittes erfolgt; es wird vorausgesetzt, daß die Summe aller *Wahrscheinlichkeiten* ≤ 1 ist.

· *ELSE*: optionale Notation (an maximal einem Nachfolgerabschnitt) für Fälle, in denen die Summe der expliziten *Wahrscheinlichkeiten* < 1 ist; Fortsetzung entlang des bezeichneten Nachfolgerabschnittes mit Wahrscheinlichkeit = (1 - Summe der expliziten *Wahrscheinlichkeiten*).

Attribute zweiter Fall:

· *Bedingung*: Boole'scher Ausdruck, der einen Wahrheitswert (TRUE / FALSE) liefert; Fortsetzung des Prozesses entlang des bezeichneten Nachfolgerabschnittes im Falle TRUE; es wird vorausgesetzt, daß maximal eine der *Bedingungen* zutrifft; Notierung des Boole'schen Ausdrucks über zugreifbaren Größen (zumindest Parameter und Variablen dieser PK; weitere zugreifbare Größen im Folgenden).

· *ELSE*: optionale Notation (an maximal einem Nachfolgerabschnitt); Fortsetzung entlang des bezeichneten Nachfolgerabschnittes falls alle *Bedingungen* FALSE liefern.

Symbol Schließender ODER-Konnektor:

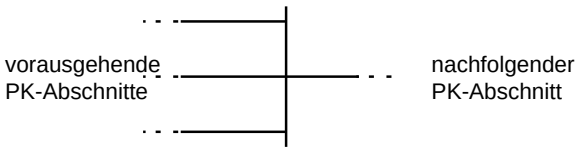


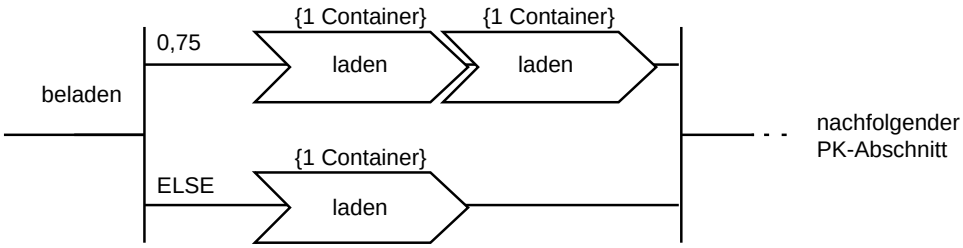
Abbildung 4-17

Prozesse jedes Vorgängerabschnitts erfahren eine Fortsetzung im Nachfolgerabschnitt.

Attribute: Keine.

Beispiel: Beladevorgang, ein oder zwei Container umfassend, die sequentiell zu laden sind. Alternative Spezifikationen:

Alternative 1:
Statistische Unterscheidung der Fälle; in 75% der Prozesse zwei Container.



Alternative 2:
Festlegung ein oder zwei Container bei Prozeßstart (Parameter n).

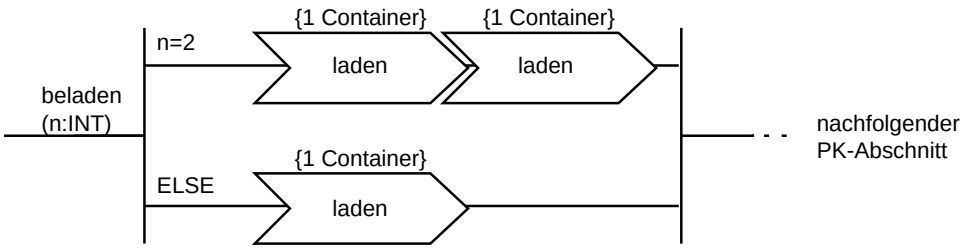


Abbildung 4-18

4.2.2.2.3 Öffnende und Schließende UND-Konnektoren

Öffnende UND-Konnektoren ermöglichen eine gleichzeitige / parallele ("sowohl als auch") Fortsetzung von Prozessen; präziser gesagt, drückt ein Öffnender UND-Konnektor aus, daß aus Sicht des Prozesses seine Fortsetzung in mehreren parallelen PK-Abschnitten erfolgt. Schließende UND-Konnektoren erlauben es, parallele PK-Abschnitte wieder "einzusammeln"; präziser gesagt, drückt ein Schließender UND-Konnektor aus, daß aus Sicht des Prozesses seine Fortsetzung erst zulässig ist, wenn spezifische PK-Abschnitte allesamt beendet sind. UND-Konnektoren sind im fertigen Modell beidseitig mit Elementen (PKEs oder Konnektoren) verbunden (sind Nachfolger vorausgehender PK-Abschnitte und Vorgänger nachfolgender PK-Abschnitte).

Symbol Öffnender UND-Konnektor:

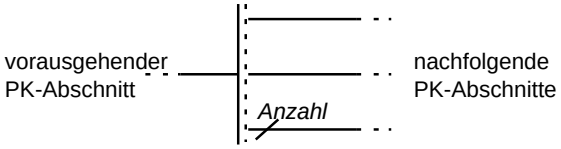


Abbildung 4-19

Attribute:

· *Anzahl* (optional): ganze Zahl (arithmetischer Ausdruck); als Kurzform für eine Menge von *Anzahl* paralleler, gleichartiger Nachfolgeabschnitte, welche mit gleicher Bedeutung explizit als *Anzahl* separater paralleler identischer Nachfolgeabschnitte notiert werden könnten. Bei Fehlen des Attributs ist für den Nachfolgeabschnitt implizit (DEFAULT-Wert) *Anzahl* = 1 angenommen.

Symbol Schließender UND-Konnektor:

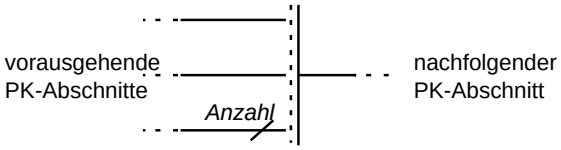


Abbildung 4-20

Attribute:

· *Anzahl* (optional): ganze Zahl (arithmetischer Ausdruck); als Kurzform für eine Menge von *Anzahl* paralleler, gleichartiger Vorgängerabschnitte, welche mit gleicher Bedeutung explizit als *Anzahl* separater paralleler identischer Vorgängerabschnitte hätten notiert werden können DEFAULT-Wert *Anzahl* = 1.

Beispiel:

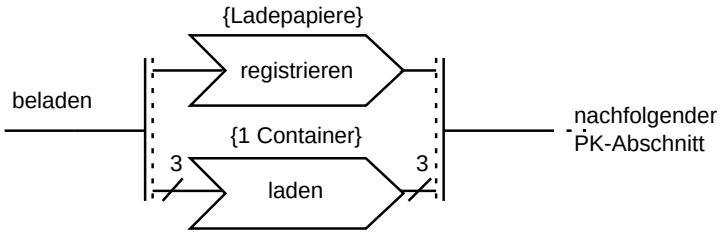


Abbildung 4-21

Im Ablauf von Prozessen *beladen* (von z. B. LKWs) sollen drei Container geladen sowie die zugehörigen Ladepapiere ausgefertigt werden. Aus Sicht des Prozesses können diese (insgesamt 4) Aktivitä-

ten simultan ablaufen ("Parallelisierung"), eine Fortsetzung ist aber nur nach Abschluß aller 4 Aktivitäten zugelassen.

Bemerkungen:
Zur Eröffnung paralleler Nachfolgerabschnitte wie auch zur Synchronisation paralleler Vorgängerabschnitte (vor weiterer Prozeßfortsetzung) sind komplexere Spezialfälle vorstellbar, die bei diesbezüglichem Bedarf der Anwendungs-Teilprojekte eingebracht werden können.

4.2.2.2.4 Gemischte Konnektoren und zusätzliche Spezifikationswünsche

Unmittelbar aufeinander folgende (direkt verbundene)

- Schließende ODER- bzw. UND-Konnektoren einerseits
- und Öffnende ODER- bzw. UND-Konnektoren andererseits

dürfen graphisch zusammengelegt werden, so daß Gemischte Konnektoren der Typen ODER/ODER, ODER/UND, UND/UND, UND/ODER entstehen (bzw. so daß ein Konnektortyp definierbar wäre, der alle Optionen umfaßt).

Symbole:

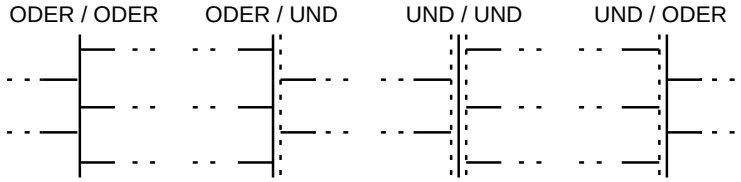


Abbildung 4-22

Die Diskussion von Konnektoren stand bisher durchgehend unter dem Vorzeichen, daß in einer Prozeßkette ("Besuchs"-)Reihenfolgevorschriften von Elementen für jeden individuellen Prozeß definiert wurden. Insbesondere beim Schließenden UND-Konnektor können aber auch andersartige Spezifikationswünsche vorliegen.

Dazu ein motivierendes Beispiel:

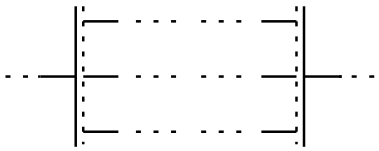


Abbildung 4-23

Gemäß bisheriger Diskussion ist hier ausgedrückt

- Ein individueller Prozeß dieser Prozeßkette ist nach einem (im Beispiel nicht detaillierten) Anfangsabschnitt in drei parallelen (hier nicht detaillierten) Abschnitten fortzusetzen.
- Erst nach Abschluß aller drei parallelen Abschnitte (genau dieses individuellen Prozesses) ist eine weitere (hier nicht detaillierte) Fortsetzung dieses individuellen Prozesses möglich.

Abweichend davon könnte der Wunsch vorliegen, die folgende Sachlage zu spezifizieren:

- Die dargestellte Prozeßkette dient als Verhaltensmuster für eine Familie individueller Prozesse; jeder von ihnen wurde individuell "gestartet", so daß potentiell mehrere Prozesse der Familie gleichzeitig existieren und voranschreiten.

- Entsprechend des skizzierten Beispiels wird sich (wie zuvor) jeder individuelle Prozeß in drei parallele Abschnitte aufspalten.

- Die Synchronisation am Schließenden UND-Konnektor ist (abweichend von zuvor) so gemeint, daß in jedem der drei parallelen Abschnitte ein Abschluß vorzuliegen hat, bevor eine Prozeßfortsetzung erfolgen kann. Dabei ist nicht verlangt, daß die drei abgeschlossenen Parallelabschnitte aus der Aufspaltung desselben individuellen Prozesses stammen.

Diese abweichende Sachlage wird im weiteren Verlauf getrennt zu diskutieren sein (s. 4.2.7). Sie berührt sowohl die geforderte Eindeutigkeit des Prozeßträgers als auch die Zugehörigkeit des UND-Konnektors zu einer Prozeßkette.

4.2.2.3 Quellen und Senken

Quellen (bzw. Senken) spezifizieren allgemein die Umstände der Entstehung (bzw. Beendigung) von Prozessen. Im hier vorliegenden Kontext der einfachen Prozeßketten sind zunächst solche Quellen und Senken von Interesse, welche die Schnittstelle zwischen spezifiziertem Modell und (nicht spezifizierter) Umgebung des Modells darstellen. In dieser Rolle beschreiben Quellen und Senken das Verhalten und die Charakteristika dieser Umgebung so weit wie erforderlich. Solche Quellen werden hier als "Unbedingte Quellen" bezeichnet. Zusätzliche Modellierungskonstrukte mit Quellen- und Senkencharakter werden im weiteren Verlauf auftreten.

4.2.2.3.1 Unbedingte Quellen

Die Unbedingte Quelle startet / erzeugt (zu vorbekannten – vom Modellablauf unabhängigen - Zeitpunkten und in festgelegter Anzahl) individuelle Prozesse der mit dieser Quelle verbundenen Prozeßkette.

Symbol:

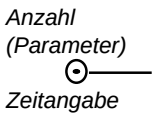


Abbildung 4-24

Attribute:

- *Anzahl*: ganzzahliger Wert (arithmetischer Ausdruck), die Anzahl der zu einem Zeitpunkt generierten individuellen Prozesse spezifizierend.

- *Parameter*: (aktuelle) Werte bzw. Ausdrücke für die (formalen) Parameter der verbundenen Prozeßkette; in Zahl und Typ auf die Parameter dieser Prozeßkette abgestimmt;

- *Zeitangabe*: In zwei alternativen Formen

  Form 1: AT t (u. U. Liste von t)

  mit REAL-wertigem t (arithmetischer Ausdruck); mit der Bedeutung, daß zum Zeitpunkt t ein individueller Prozeß der verbundenen Prozeßkette generiert wird (ggf. gemäß *Anzahl* mehrere individuelle Prozesse), mit aktuellen Parametern gemäß *Parameter*.

  Form 2: EVERY t

  mit REAL-wertigem t (arithmetischer Ausdruck); mit der Bedeutung, daß repetitiv alle t Zeiteinheiten ein individueller Prozeß der verbundenen Prozeßkette generiert wird (ggf. gemäß *Anzahl* mehrere individuelle Prozesse), mit aktuellen Parametern gemäß *Parameter*.

(Bei beiden Formen sind zusätzliche Optionen vorstellbar, die bei entsprechendem Bedarf später zu ergänzen sind.)

Ein **einfaches Beispiel**:

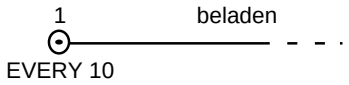


Abbildung 4-25

Eine Quelle, verbunden mit einer (unparametrisierten) Prozeßkette *beladen* (vgl. 4.2.2.2.2), startet repetitiv alle 10 Zeiteinheiten einen individuellen *beladen*-Prozeß.

Ein **komplexeres Beispiel**:

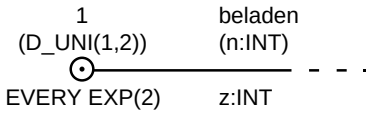


Abbildung 4-26

Eine Quelle,

- verbunden mit einer Prozeßkette *beladen* (1 ganzzahliger Parameter n; vgl. 4.2.2.2.2),
- startet repetitiv, mit exponentiell (Rate: 2/Zeiteinheit) verteilten Abständen - also mit mittlerem zeitlichem Abstand 1/2 ZE
- jeweils 1 *beladen*-Prozeß, parametrisiert mit einem Wert, der aus einer diskreten Gleichverteilung über [1,2] gezogen ist – also in der Hälfte der Fälle mit Wert 1, in der anderen Hälfte mit Wert 2.

4.2.2.3.2 Unbedingte Senken

Eine Unbedingte Senke beendet Prozesse, wenn sie im Ablauf der Prozesse erreicht wird. Sie ist mit dem (im Ablauf letzten) Element der zugehörigen (einfachen) Prozeßkette verbunden. Zur Erzielung dieses Zwecks ist es erforderlich, alle evtl. vorhandenen alternativen und / oder parallelen "Verästelungen" einer PK graphisch (mit Hilfe geeigneter Konnektoren) "zusammenzuführen". (Von der in Abschnitt 4.1 angedeuteten Möglichkeit der impliziten Beendigung von Prozessen, aufgrund der Erschöpfung ihrer Aktivitäten, wird also kein Gebrauch gemacht.)

Symbol:

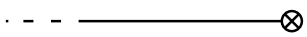


Abbildung 4-27

Attribute: Keine

4.2.3 Funktionseinheiten in externer Sicht

Die Durchführung der Aktivitäten einer Prozeßkette obliegt gemäß 4.1 geeigneten Funktionseinheiten / Organisationseinheiten (FEs). FEs können Attribute aufweisen, mit deren Setzung ihre Fähigkeit und Arbeitsweise (im Rahmen ihrer generellen Möglichkeiten) bestimmt und beeinflußt werden. Unter anderem zeigen FEs ihre Fähigkeiten (ihre "Eignung") zur Durchführung bestimmter Aktivitätsarten dadurch an, daß sie (Attribut:) einen oder mehrere benannte Dienste anbieten. Dienste können para-

metrisiert sein, so daß eine auszuführende Aktivität (über ihre "Art" hinaus) genauer charakterisierbar ist. Funktionseinheiten können typisiert werden, um so eine Wiederverwendung gleichartiger struktureller "Bausteine" zu ermöglichen. Einige grundlegende FE-Typen liegen vordefiniert vor, so die Standardtypen Server und Counter, deren Beschreibung im unmittelbaren Anschluß folgt. Andersartige FEs und FE-Typen können wie später beschrieben (vgl. 4.2.5) aus diesen Standardtypen konstruiert werden. Zur graphischen Darstellung von FEs der Standardtypen sind spezifische Symbole vorgesehen. Konstruierte FEs werden dagegen mithilfe eines diesbezüglichen Standardsymbols erfaßt.

4.2.3.1 Die Funktionseinheit vom Typ Server

Abstrakt gesprochen verwaltet ein Server einen eindimensionalen diskreten Raum, dessen Elemente für bestimmte Zeitspannen angefordert werden.

Konkreter gesprochen eignet sich ein Server zur Erfassung etwa einer arbeitsfähigen Person oder einer arbeitsbereiten Maschine / Einrichtung (vergleichbar einer "Ressource" des A-Paradigmas), oder zur Darstellung einer Gruppe arbeitsfähiger Personen / Maschinen / Einrichtungen (A-Paradigma: "Ressourcen-Pool"), sofern die Anforderungen an die arbeitsfähigen "Elemente" durch Angabe eines Arbeits-"Umfangs" voll charakterisierbar sind. Eine der Möglichkeiten zur Festlegung eines Arbeitsumfangs besteht i.allg. in Form der Angabe seiner Normdauer (seiner "Vorgabezeit"), präziser definiert der Länge des Zeitintervalls, das ein einzelnes Arbeitselement für die Verrichtung der Arbeit benötigen würde (vorausgesetzt, es wäre exklusiv dieser Arbeit zugeordnet).

Der Server ist der Modellwelt der Warteschlangensysteme und Warteschlangennetze entlehnt - auf die dortige Bedienstation / Queue (oder eben: den dortigen Server) sei als Verständnisbrücke hingewiesen.

Für Server-FEs existieren vielfältige Attributierungsmöglichkeiten, welche sehr flexible "Prägungen", Anpassungen an konkrete Einsatzzwecke erlauben. Von diesen Möglichkeiten ist im Folgenden nur eine kleine Auswahl erfaßt. Eine abschließende Auswahl kann sich an den im Problembereich auftretenden Notwendigkeiten ausrichten.

Symbol:

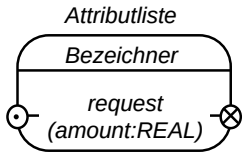


Abbildung 4-28

Attribute:

- *Bezeichner* (zwingend): Name der FE zum Zwecke ihrer Referenzierung;
- *request(amount:REAL*): Ein Server bietet standardmäßig und unveränderbar einen einzigen Dienst namens *request* an. *request* ist formal parametrisiert mit *amount*, dem Umfang der auszuführenden Tätigkeit in Arbeitseinheiten (AE), wobei selbstverständlich entschieden werden kann, daß Arbeit in Zeiteinheiten (ZE) gemessen wird, in welchem Falle der Parameter als Arbeitszeit(-Bedarf) zu interpretieren ist. Die graphische Kennzeichnung des Dienstes in einer "Quellen-Senken-Klammer" geschieht in Analogie zur Spezifikation konstruierter Funktionseinheiten (vgl. dort);
- *Attributliste* : Liste prägender Attribute, sämtlich in der Form

  SCHLÜSSEL = wert

  angegeben; all diese Attribute besitzen DEFAULT-Werte, die nicht explizit gesetzt werden müssen.

Derzeit vorgesehen sind die Attribute / Schlüssel

- *Bedien-Geschwindigkeit*:

  SPEED = <positiv reell> DEFAULT: 1

  Beschreibt die potentiell nötige Umrechnung des geforderten Arbeitsumfangs, von parameterseitigen Arbeitseinheiten (AE) in Zeiteinheiten (ZE), Dimension also AE/ZE. Wurde entschieden, daß der Arbeitsumfang direkt in ZE angegeben wird, ist offensichtlich keine Umrechnung nötig und SPEED=1 die adäquate Setzung. (Modellierungstechnisch kann der SPEED-Wert auch zu anderen Zwecken hilfreich eingesetzt werden.)

- *Bedien-Disziplin*:

  DIS = FCFS | PS | IS DEFAULT: FCFS

  womit eine der möglichen Strategien / Disziplinen der Bedienung gleichzeitig vorliegender Beauftragungen ausgewählt wird:

  FCFS steht für nicht unterbrechende Bedienung in der Reihenfolge der Beauftragungszeitpunkte; im Zusammenhang mit der Angabe der *zeitlichen Kapazität* (s. u.) schreiten bis zu CAP-Wert Bedienungsvorgänge gleichzeitig voran, wobei jeder einzelne eine Bedienfortschritt von 1 ZE je ZE erzielt; stehen mehr als CAP-Wert Vorgänge an, müssen diese auf Bedienung warten.

  PS steht für gleichzeitige Bedienung aller (im Server) anstehenden Bedienungsvorgänge; im Zusammenhang mit der Angabe der *zeitlichen Kapazität* (s. u.) erzielt bei n anstehenden Vorgängen jeder einzelne einen Bedienfortschritt von CAP-Wert /n ZE je ZE.

  IS steht für gleichzeitige Bedienung aller (im Server) anstehenden Bedienungsvorgänge; unabhängig von der Zahl anstehender Vorgänge und unabhängig erzielt jeder einzelne einen Bedienfortschritt von 1 ZE je ZE.

- *Bedien-Kapazität* (zeitliche Kapazität):

  CAP = <positiv ganz>DEFAULT: 1

  Beschreibt dem Sinne nach die Anzahl der im Server zur Verfügung stehenden, potentiell simultan aktiven Arbeitseinheiten (Arbeitsplätze / Arbeitsvorrichtungen); formaler gesehen, setzt CAP die maximale gesamte Bearbeitungsrate im Server auf CAP-Wert ZE je ZE.

Weitere Attribute (etwa Bedienung nach Prioritäten, räumliche Grenzkapazität, etc.) sind denkbar und können bei Bedarf zugefügt werden.

Beispiel 1:

Für die Ladetätigkeiten der Beispiele gemäß Abbildung 4-18 bzw. Abbildung 4-21 könnte ein Pool von Gabelstaplern zur Verfügung stehen. Dieser Pool wäre im einfachsten Falle durch eine Server-FE *stapler* erfaßbar, dessen Attributierung festzulegen hätte,

· wieviele Gabelstapler insgesamt verfügbar sind;

  z. B. CAP=2 für den Fall, daß 2 Gabelstapler vorhanden wären;

· nach welchen Regeln potentielle Bearbeitungskonflikte zwischen mehreren simultan vorliegenden Aufträgen aufzulösen sind;

  z. B. DIS=FCFS für den Fall, daß jede einmal begonnene Bearbeitung ohne Unterbrechung vollständig durchzuführen wäre, sowie daß bis zu CAP-Wert Bearbeitungen simultan durch-

führbar wären, darüber hinaus anstehende Bearbeitungen aber auf eine freiwerdende Bearbeitungseinheit zu warten hätten, wo bei Freiwerden einer Bearbeitungseinheit der Auftrag mit der längsten Wartezeit gegenüber später eingetroffenen Vorrang hätte;

- wie der bei Auftragserteilung angegebene Parameterwert des genutzten *request*-Dienstes hinsichtlich notwendiger Bearbeitungszeit zu interpretieren ist, wobei "je nach Geschmack" verschiedene Optionen bestehen;

  z. B. SPEED=1 für den Fall, daß der Dienstaufruf direkt mit der Vorgabezeit der Tätigkeit für das Laden eines Containers parametrisiert wäre – also etwa Parameterwert 5 für eine Vorgabezeit von 5 min;

  z. B. SPEED=0.2 für den Fall, daß der Dienstaufruf mit der Anzahl zu ladender Container parametrisiert wäre – 0.2 Container je Minute resultierten dann ebenfalls in 5 min je Container.

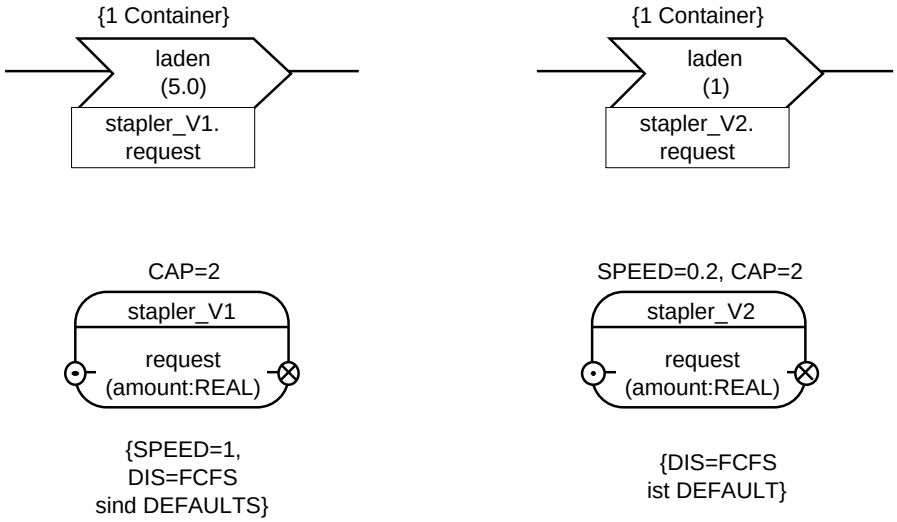


Abbildung 4-29

Abbildung 4-29 zeigt für beide Optionen der *request*-Parametrisierung jeweils das PK-Element einer Dienst-Anforderung "Ladung eines Containers" zusammen mit der externen Sicht einer beauftragten Server-FE. In Konkretisierung der schematischen Angaben zum *Ausführungshinweis* in Abschnitt 4.2.2.1.1 und im Vorgriff auf Abschnitt 4.2.4 (Prozeßkettenpläne) ist auch der Verweis der auszuführenden Aktivität an den angebotenen FE-Dienst explizit ausgedrückt.

Anmerkung:
Im Rahmen des Beispiels der Abbildung 4-21 (simultane Anforderung von 3 Container-Beladungen) ergibt sich demnach bei der gewählten Attributierung der genutzten Server-FE eine minimale Abwicklungszeit des Parallel-Konstrukts von 2·5 = 10 min, da ja maximal 2 Beladungsvorgänge tatsächlich parallel abgewickelt werden und der dritte auf die Beendigung der ersten beiden zu warten hat (dieser Minimalwert wird natürlich nur dann erreicht, falls die genutzte Server-FE nicht weitere simultan vorliegende Bearbeitungen abzuwickeln hat und zusätzlich die Ausfertigung der Ladepapiere nicht mehr als 10 min in Anspruch nimmt).

Beispiel 2:

Eine Server-FE mit einer Bedienkapazität CAP="unendlich" könnte offensichtlich beliebig viele Aufträge simultan ausführen; da sich allerdings "unendlich" schlecht als reelle Zahl notieren läßt, ist dieser Fall durch die spezielle Bediendisziplin DIS=IS (kurz für "Infinite Server") erfaßt, wobei ein etwa angegebener CAP-Wert unerheblich ist. Tritt zur Attributierung DIS=IS noch SPEED=1 hinzu (der DEFAULT-Fall), dann erhält man eine Server-FE, die für jede Anforderung (unabhängig von etwa

vorliegenden anderen simultanen Anforderungen) genau die Zeit benötigt, die im Parameterwert der Anforderung angegeben ist. Diese FE bewirkt somit eine reine Verzögerung im Ablauf des anfordernden Prozesses. Ihre Nutzung ist identisch mit dem Spezialfall des DELAY-Schlüsselworts im Verweisteil des anfordernden PKEs (vgl. Abschnitt 4.2.2.1.1), das sich damit als bequeme Abkürzung im Rahmen der hier vorliegenden allgemeinen Fälle erweist.

4.2.3.2 Die Funktionseinheit vom Typ Counter

Abstrakt gesprochen verwaltet ein Counter einen mehrdimensionalen diskreten Raum, dessen Elemente explizit und für unbestimmte Zeitspannen angefordert und explizit freigegeben werden.

Konkreter gesprochen eignet sich ein Counter zur Erfassung etwa eines Lagers, eines Puffers, allgemein eines begrenzten "Platzes", aber auch zur Beschreibung einer begrenzten Zahl von Arbeitsmitteln, Transportmitteln etc., sofern die Anforderungen an die benötigten (bzw. nicht mehr benötigten) "Elemente" durch Angabe eines Anforderungs- (bzw. Freigabe-) "Umfangs" (Zahl von Elementen) voll charakterisierbar sind. (Vgl. auch hier mit dem Ressourcen-Pool des A-Paradigmas). Modellierungstechnisch sind mit dem Counter zwei häufig auftretende Problembereiche erfaßt: Das "Einfügen in begrenzten Raum / Belegen von begrenztem Raum" sowie das "Entnehmen aus begrenzt vorhandener Menge".

Counter-FEs besitzen Attribute, welche eine Anpassung an konkrete Einsatzzwecke erlauben. Auch hier ließe sich die Attributierung gemäß der im Problembereich auftretenden Notwendigkeiten erweitern.

Symbol:

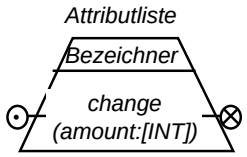


Abbildung 4-30

Attribute:

- *Bezeichner* (zwingend): Name der FE zum Zwecke ihrer Referenzierung;

- *change(amount:[INT]*): Ein Counter bietet standardmäßig und unveränderbar einen Dienst namens *change* an. *change* ist formal parametrisiert mit *amount*, einem Vektor von INTEGER-Werten, welcher der Übergabe der Anzahl geforderter (bzw. freizugebender) Elemente in den eingestellten Dimensionen (vgl. *Counter-Dimension*) dient; dabei gelten positive Anzahlen als Anforderung, negative als Freigabe; die graphische Kennzeichnung des Dienstes in "Quellen-Senken-Klammer" geschieht wieder analog zur Spezifikation konstruierter Funktionseinheiten;

- *Attributliste* : Liste prägender Attribute, angegeben in der Form

  SCHLÜSSEL = wert

  Derzeit vorgesehen sind die Attribute / Schlüssel

  - *Counter-Dimension*:

    Implizites Attribut; wird aus dem Initalisierungsattribut abgeleitet und kann nicht explizit gesetzt werden. Legt die Dimensionalität des verwalteten Raumes fest. Wohl in den meisten Anwendungsfällen wird Dimension = 1 sein.

- *Counter-Initialisierung*:

  INIT = [INT,INT,…]

  Beschreibt die initial vorhandene Belegung (Zahl von Einheiten) in den vorhandenen Dimensionen. Setzt durch die Anzahl der angegebenen Vektor-Positionen implizit die Dimensionalität des Counters.

- *Belegungs-Untergrenzen*:

  MIN = [INT,INT,…]

  Beschreibt die minimal zulässige Belegung (Zahl von Einheiten) in den vorhandenen Dimensionen. Funktionserklärung direkt im Anschluß.

- *Belegungs-Obergrenzen*:

  MAX = [INT,INT,…]

  Beschreibt die maximal zulässige Belegung (Zahl von Einheiten) in den vorhandenen Dimensionen. Funktionserklärung direkt im Anschluß.

- *Wartedisziplin*:

  DIS = RANDOM | {zusätzliche Optionen noch festzulegen} DEFAULT: RANDOM

  womit eine der möglichen Strategien / Disziplinen der Auflösung von Wartesituationen bei gleichzeitig vorliegenden (erfüllbaren) Forderungen ausgewählt wird:

  RANDOM steht für (gleichverteilt) zufällige Berücksichtigung einer Anforderung aus der Menge jener wartenden Anforderungen, welche (gemäß Belegungszustand und Grenzen) durchführbar sind.

Funktionserklärung Belegungs-Initialisierung und -Grenzen:

Sei einführend ein eindimensionaler Counter angenommen, etwa zur Erfassung eines Lagers, dessen räumliche Kapazität in gleichartigen "Plätzen" angegeben wird, und in das (bzgl. Lagerung) gleichartige "Güter" eingelagert (bzw. aus dem Güter entnommen) werden können, die jeweils einen Platz des Lagers belegen (bzw. freigeben).

Das Lager sei zu Anfang des Betriebs "leer", spezifiziert durch die Attributierung INIT=[0]. Die Lagerkapazität sei realistischerweise endlich angenommen; sie betrage n_max Plätze, spezifiziert durch MAX=[n_max]. Als sinnvolle Minimalbelegung des Lagers könnte "leer" gefordert sein, spezifiziert durch MIN=[0]. Der Counter "merkt sich" im dynamischen Ablauf (intern) seinen Füllungszustand n_momentan. Er ändert seinen Füllungszustand aufgrund von (an ihn verwiesenen) Aktivitäten der Art *change([n_änderung])* gemäß n_momentan:=n_momentan+n_änderung (erhöht seinen Füllungszustand also bei positivem n_änderung, erniedrigt ihn bei negativem n_änderung). Er ändert seinen Füllungszustand allerdings nur dann, wenn dies bzgl. der gesetzten Belegungsgrenzen erlaubt ist; würde die angeforderte *change*-Aktivität Unter- oder Obergrenzen verletzen, wird sie nicht ausgeführt – der anfordernde Prozeß "bleibt stecken", wartet auf einen (späteren) Zeitpunkt, zu dem die Aktivität ohne Verletzung der Grenzen durchgeführt werden kann.

Mehrdimensionale Counter arbeiten analog, in mehreren Dimensionen. So könnte etwa ein Counter der Dimension 2 ein Lager erfassen, in dem 2 Arten von Gütern auf 2 unterschiedlichen Arten von Plätzen unterzubringen sind, mit jeweils ihrer Initial-, Minimal- und Maximalbelegung, und mit der Möglichkeit, mittels 2-dimensionaligem *change([n1,n2])* eine simultane Veränderung beider Lagerplatzarten herbeizuführen. Wesentlich: Die Aktivität wird entweder insgesamt ausgeführt oder

wird (bei drohender Verletzung auch nur einer Unter- oder Obergrenze) insgesamt zu warten haben.

Beispiel:

Ein Lager werde von Prozessen einer PK *einlagern* beschickt, während Prozesse einer PK *auslagern* dem Lager Teile entnehmen. In einer Minimalversion sei nur ein Typ von Teilen vorgesehen. Eine Counter-FE *lager* werde genutzt, um die reine Kapazitäts- und Füllungsverwaltung des Lagers zu erfassen (s. Abbildung 4-31).

Die Attributierung von *lager* hätte festzulegen:

- wieviele Teile zu Beginn des Betriebs im Lager weilen;

  z. B. INIT=[500] für den Fall einer Initialbelegung von 500;

  (mit gleichzeitiger Festlegung der Dimension von lager als "1", aufgrund der Annahme lediglich eines Typs von Lagerteilen);

- welche Minimalzahl von Teilen im Lager toleriert werden;

  z. B. MIN=[0] für den Fall daß das Lager völlig geleert werden dürfte;

- welche Maximalzahl von Teilen das Lager aufnehmen kann;

  z. B. MIN=[1000] für den Fall einer Kapazität von 1000;

- nach welchen Regeln unter mehreren Prozessen, die auf eine Änderung des Lagerzustands warten (also einlagernde Prozesse auf benötigten Lagerplatz bei "zu vollem" Lager, auslagernde Prozesse auf Verfügbarkeit von Teilen bei "zu leerem" Lager), bei erfolgter Änderung ein als nächstes zu berücksichtigender Prozeß ausgewählt wird;

  z. B. (momentan einzige vorgesehene Möglichkeit und DEFAULT) DIS=RANDOM für den Fall zufälliger Auswahl;

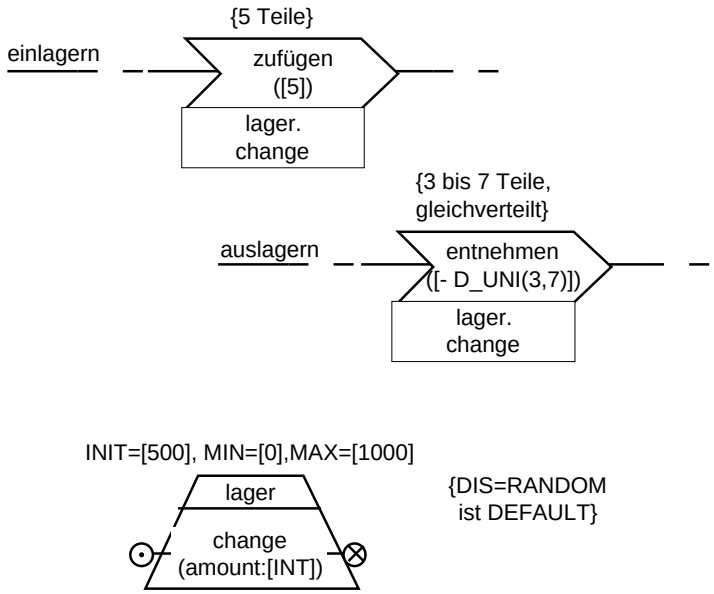


Abbildung 4-31

Abbildung 4-31 zeigt die wesentlichen Ausschnitte der beiden erwähnten Prozeßketten *einlagern* und *auslagern* in Form der Aktivitäten *zufügen* und *entnehmen*. Zur Parametrisierung der Aktivitäten vgl. deren Kommentarfelder. In Konkretisierung der schematischen Angaben zum *Ausführungshinweis* in Abschnitt 4.2.2.1.1 und im Vorgriff auf Abschnitt 4.2.4 (Prozeßkettenpläne) sind wieder die Verweise der auszuführenden Aktivitäten an den angebotenen FE-Dienst explizit ausgedrückt; man beachte, daß in der Basisform des Counters ein einziger Dienst *change* angeboten ist, der für positive Parameterwerte ein Füllen, für negative Werte ein Entleeren vornimmt (entweder zum konkreten Zeitpunkt der Anforderung oder später, nach etwa notwendigem Warten).

4.2.3.3 Konstruierte Funktionseinheiten

Wie in der Einleitung zu 4.2.3 bereits erwähnt, erlaubt der B1-Formalismus, selbst konstruierte Funktionseinheiten (neben den Standard-FEs) einzusetzen. Über die Konstruktion von FEs gibt Abschnitt 4.2.5 näheren Aufschluß. Mit der Erstellung und Verwendung selbst konstruierter FEs wird die Stufe hierarchischer Modelle erreicht werden. Die externe Darstellung konstruierter FEs (und damit die Sicht auf ihre Verwendung) bedient sich folgenden Standardsymbols und seiner Attributierungen. (In Abschnitt 4.2.7.2 werden zusätzliche Varianten hinzukommen.)

Symbol:

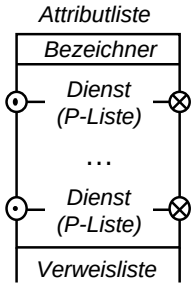


Abbildung 4-32

Attribute:

- *Bezeichner* (zwingend): Name der FE zum Zwecke ihrer Referenzierung;

- *Dienst(P-Liste*): Charakterisierung eines oder mehrerer von dieser FE (gemäß ihrer Konstruktion) angebotener Dienste. *Dienst* ist der Bezeichner (Name) eines Dienstes, der seine Referenzierung erlaubt. *P-Liste* ist eine Parameterliste dieses Dienstes (gemäß seiner Konstruktion). Aufgelistet werden Name und Typ aller Parameter in der Form *Name:Typ*. Bei Verwendung eines Dienstes (d. h. bei Verweisung einer Aktivität einer Prozeßkette an diesen Dienst) ist eine zur P-Liste konsistente Liste aktueller Parameter zu übergeben.

  (Anmerkung: Man vergleiche die obigen Standard-FEs Server und Counter, bei denen genau nach dieser Vorschrift verfahren wurde.)

- *Attributliste*: : Liste prägender Attribute dieser FE, sämtlich in der Form

  SCHLÜSSEL = wert

  angegeben; die angebbaren bzw. anzugebenden Attribute richten sich nach den diesbezüglichen Festlegungen bei Konstruktion der FE; als SCHLÜSSEL findet der bei Konstruktion festgelegte Attributname Verwendung.

- *Verweisliste*: Wird erst bei Konstruktion komplizierterer hierarchischer Modelle (und gemäß deren spezifischer Konstruktion) Verwendung finden; kann hier noch nicht klar beschrieben werden. Im Zweifelsfall: leer. Vgl. aber Abschnitt 4.2.8.1.

Zu Beispielen vgl. anschließenden Abschnitt 4.2.4.

4.2.4 Prozeßkettenpläne

Mit der Definition von Prozeßkettenplänen wird in der vorliegenden Beschreibung des B1-Formalismus erstmals das Niveau erreicht, die Konstrukte des Formalismus derart zusammenfügen zu können, daß alle wesentlichen Informationen analysierbarer (allerdings hier: "flacher", nicht hierarchischer) Modelle vorliegen. Die B1-Spezifikation wird erst in 4.2.5 so erweitert werden, daß auch hierarchische Modelle beschreibbar werden. In diesem Sinne konzentriert sich 4.2.4 ausschließlich auf flache Modelle (und setzt daher die bereits skizzierten konstruierten FEs nicht ein).

Prozeßkettenpläne des B1-Formalismus enthalten:

- die Beschreibung einer oder mehrerer **Prozeßketten**,
- Spezifikationen jener **Funktionseinheiten**, die mit der Ausführung von Aktivitäten dieser Prozeßketten beauftragt werden sollen,
- sämtliche Verweis-Informationen, d. h. sämtliche "**Bindungen**" von PKEs an FE-Dienste,
- (optional:) die Vereinbarung von **Variablen**, d. h. eines für einen Prozeßkettenplan insgesamt zugreifbaren Datenraums / Zustandsraums,
- **weitere Informationen**, die hauptsächlich dem Modellierungskomfort dienen und daher erst im weiteren Verlauf eingeführt werden.

Das Beispiel der Abbildung 4-33 zeigt die prinzipielle graphische Erscheinungsform eines PK-Plans; nur wenige Einzelheiten sind als Erklärungsgrundlage konkretisiert; enthalten sind insbesondere:

- Skizzen zweier Prozeßketten, *PK_1* und *PK_n*
  (die skizzierten Quellen und Senken bedürfen noch einer Vervollständigung, welche je nach Einsatz unterschiedlich ausfallen wird - vgl. 4.2.5.1 bzw. 4.2.5.2)
- mit jeweils zwei (aus den gesamten PKs herausgegriffenen) PKEs zur Erfassung der Aktivitäten
  - *bearbeiten* und *verpacken* in PK_1,
  - *bearbeiten* und *verpacken* in PK_n,

  (unterschiedliche Kommentare zu Zwecken der Unterscheidung)
- drei Funktionseinheiten:
  - *drehbank*, eine Server-FE mit DEFAULT-Parametrisierung (DIS=FCFS, CAP=1, …) zur Erfassung der Vorstellung einer einzelnen Drehbank samt Bediener, an der Beauftragungen jeweils vollständig, in der Reihenfolge ihrer Ankunft, bearbeitet werden;
  - *fräse*, ein Server mit DEFAULT-Parametrisierung, und Sachlage analog zur Drehbank;
  - *packerei*, ein Server, parametrisiert mit Bedienstrategie PS und Bedienkapazität 5, zur Erfassung der (groben, aber u. U. hinreichenden) Vorstellung einer Arbeitsgruppe von 5 Bearbeitern, die sich um alle anstehenden Verpackungsaufträge zugleich, "irgendwie gleichmäßig", bemühen;
- ein Bereich ***Variablen*** (im Beispiel nicht genutzt), der wie gesagt die Möglichkeit eröffnet, einen für alle Prozeßketten dieses Prozeßkettenplans gemeinsam zugreifbaren Datenraum / Zustandsraum einzurichten; die Variablenvereinbarungen erfolgen in derselben syntaktischen Form wie die der Vereinbarungen der Prozeßketten-Variablen (vgl. 4.2.2) und können wie dort optional initialisiert werden; (in Wiederholung:) eine Variablen-Vereinbarung

k:INT INIT 0

vereinbart eine ganzzahlige Größe *k*, welche initial den Wert 0 besitzt;

· die expliziten Durchführungs-Verweise (von PKEs an FE-Dienste):

{PKE_1.1}-*bearbeiten* der PKE_1 an *request*-Dienst der *drehbank*-FE, …, …,
{PKE_n.2}-*verpacken* an *packerei.request*;

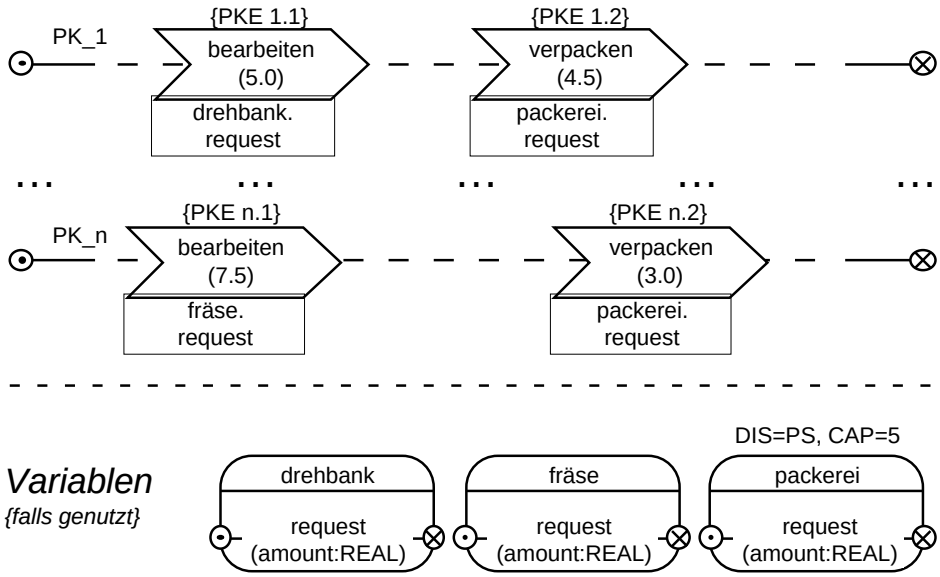


Abbildung 4-33

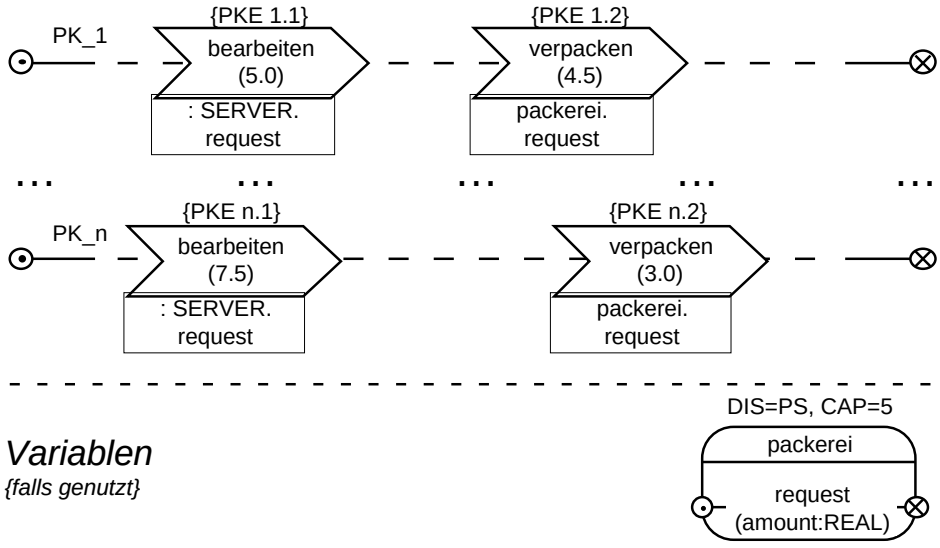


Abbildung 4-34

Um eine weitestgehende Ähnlichkeit zum A-Paradigma zu schaffen (in dem Prozeßkettenelemente und ausführende Akteure deutlich stärker miteinander "verknüpft" sind als in der vorliegenden

B1-Sicht), ist vorgesehen, eine "A-ähnlichere" Notation in den Fällen zuzulassen, in denen dies konfliktfrei möglich ist. Dabei handelt es sich insbesondere um Fälle, in denen eine agierende Einheit ("Funktionseinheit") exklusiv für die Aktivität eines einzelnen Prozeßkettenelementes zur Verfügung steht, diese agierende Einheit also sonst (will heißen: für die Aktivitäten anderer PKEs) nicht benötigt wird.

Ist in der PK-Plan-Skizze der Abbildung 4-33 *drehbank* exklusiv für PKE_1.1 zugreifbar, und *fräse* exklusiv für PKE_n.1 verfügbar, liegt dieser Fall vor. Er liegt keinesfalls vor für *packerei*, die ja sowohl von PKE_1.2 als auch von PKE_n.2 beauftragt wird, so daß es zu wechselseitigen Beeinflussungen dieser Aktivitäten kommt.

Die abkürzende Notation sieht in diesem Fall die Form gemäß Abbildung 4-34 vor. Ausgedrückt ist, daß sowohl PKE_1.1 als auch PKE_n.1 je eine FE vom Typ SERVER exklusiv "besitzen" (falls erforderlich, könnten auch diese exklusiv nutzbaren FEs noch mit Namen versehen werden).

4.2.5 Konstruierte Funktionseinheiten (in interner Sicht) und Prozeßkettenmodelle

Die in Abschnitt 4.2.4 skizzierten Prozeßkettenpläne bilden den Kern der Beschreibung sowohl konstruierter Funktionseinheiten, als auch gesamter Prozeßkettenmodelle. Die vorgeschlagenen Erweiterungen von PK-Plänen sind im Folgenden getrennt beschrieben, jeweils angereichert mit ein wenig "Modellierungskomfort". Insgesamt wird in diesem Abschnitt ein Beschreibungsniveau erreicht, in dem auch hierarchische Modelle spezifizierbar sind, wobei die hierarchischen Beziehungen sowohl "Aufruf-" als auch "Enthaltensein"-Relationen abdecken.

4.2.5.1 Die Konstruktion von Funktionseinheiten

Die Grundidee der Konstruktion von Funktionseinheiten besteht daraus,

- Prozeßkettenpläne gemäß 4.2.4 geeignet "einzukapseln",
- und dafür zu sorgen, daß einzelne Prozeßketten des eingekapselten PK-Plans "von außen" aktivierbar / aufrufbar sind.

Im Normfalle werden dabei

- die bisher "expliziten" Quellen des PK-Plans (welche individuelle Prozesse dieses PK-Plans, zu quellenseitig bestimmten Zeitpunkten, starten)

  zu "virtuellen" Quellen (welche in der Umgebung, außerhalb der FE initiierte Anstöße / Aktivierungen / Aufrufe in den Start interner Prozesse umsetzen),

- die bisher "expliziten" Senken des PK-Plans (welche den Anschluß individueller Prozesse dieses PK-Plans bewirken)

  zu "virtuellen" Senken (welche eine "Rückmeldung" der Beendigung interner Prozesse an die aufrufende Umgebung bewirken).

In der Folge werden (über diesen Normfall hinaus) komplexere Fälle auftreten.

Das Vorgehen bei der Konstruktion von Funktionseinheiten sei auf Basis des Beispiels aus 4.2.4 skizziert. Konkret sei aus der dortigen Prozeßkette eine *fabrik* zu erstellen, die zwei unterschiedliche Produkte *produkt_A* und *produkt_B* zu fertigen in der Lage ist, und solche Fertigungsvorgänge auf Anforderung einer Umgebung leistet. Der interne Fertigungsvorgang für *produkt_A* sei durch die Prozeßkette *PK_1* des Beispiels beschreiben, jener für *produkt_B* durch Prozeßkette *PK_n*.

Die Beschreibung von *fabrik* (d. h. die interne Beschreibung einer konstruierten Funktionseinheit) geschieht in folgender Form:

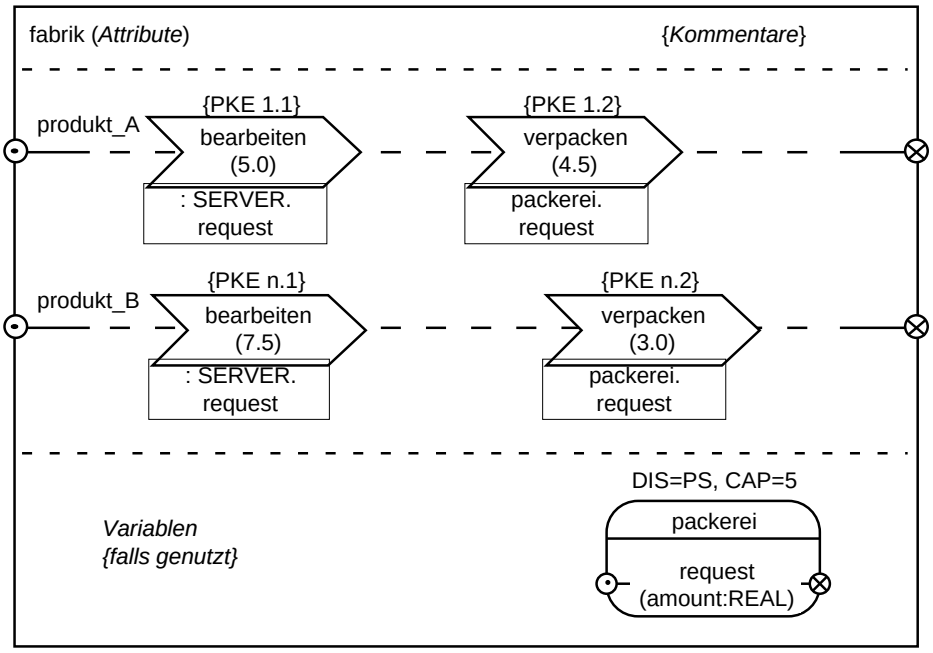


Abbildung 4-35

fabrik kann Aufträge der Art *produkt_A* (fertigen) und *produkt_B* (fertigen) aus einer Umgebung entgegennehmen, ausführen und das Ende der Ausführung an die Umgebung rückmelden. *fabrik* ist eine konstruierte Funktionseinheit in Form eines gekapselten Prozeßkettenplans. Als "Modellierungskomfort" hinzugekommen ist hier neu:

- *Attribute*: zur Ermöglichung der Beschreibung einer gewissen "Familie" ähnlicher Funktionseinheits-Varianten, aus denen (im Verlauf der Modellspezifikation, vor Modellanalyse) genau eine Variante herausgegriffen wird (beispielsweise könnte *fabrik* mit einer Größe *pack_n*, der Anzahl der Arbeiter von *packerei*, attributiert sein, um so Analyseserien über verschiedenen Werten von *pack_n* zu ermöglichen);

- *Kommentare*: als Möglichkeit, erklärende Texte zur konstruierten FE einzubringen.

Der Vollständigkeit halber sei angefügt, daß sich extern, (innerhalb eines "umgebenden" Prozeßkettenplans, der *fabrik* aufruft / benutzt; vgl. auch 4.2.5.2) *fabrik* (gemäß 0) darstellt in der Form:

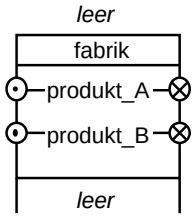


Abbildung 4-36

mit leeren (hier unbenutzten) Attributierungs- und Verweisteilen, unter expliziter Angabe der zur Referenzierung benötigten Namen, einerseits der FE selbst, andererseits ihrer verwendbaren Dienste.

4.2.5.2 Flache und hierarchische Prozeßkettenmodelle

Ein Prozeßkettenmodell (eine Modelleinheit, welche nach Abschluß ihrer Spezifikation einer Modellanalyse zugeführt werden kann) ist, liegt (bis auf abrundende Komfortangaben) mit einem Pro-

zeßkettenplan bereits vor. Man könnte ebenso sagen, daß eine konstruierte Funktionseinheit **ohne** Dienstexport (ohne virtuelle Quellen und Senken) ein Prozeßkettenmodell ausmacht.

So entsteht aus dem Prozeßkettenplan des Abschnitts 4.2.4 leicht das (flache, d. h. nichthierarchische) Prozeßkettenmodell gemäß Abbildung 4-37.

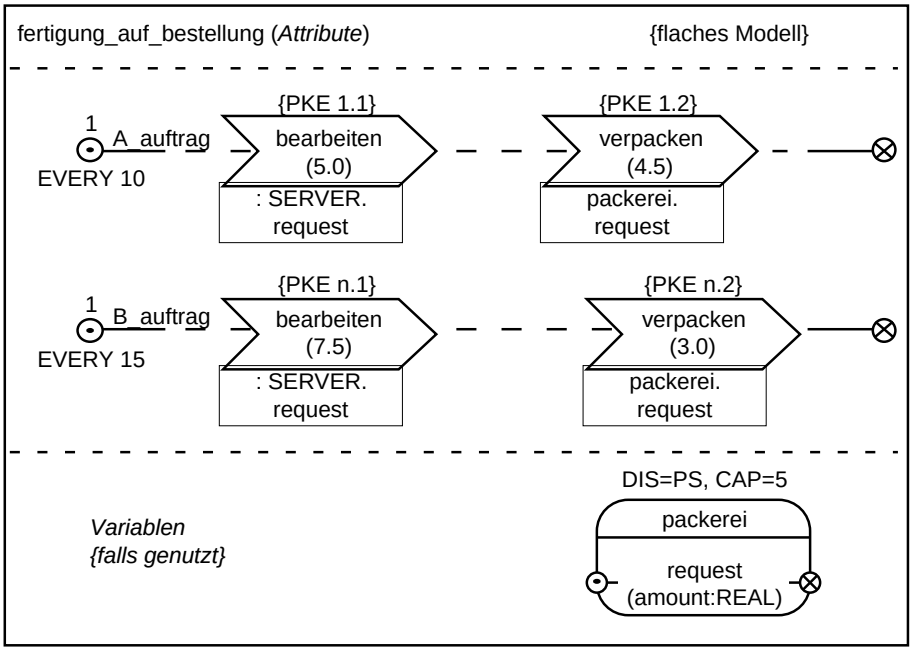


Abbildung 4-37

In diesem Modell namens *fertigung_auf_bestellung* (u. U. zu Zwecken der Variantenhandhabung mit Attributen versehen) werden die beiden Fertigungslinien (vormals *produkt_A* und *produkt_B* genannt) direkt mit Aufträgen beschickt: Alle 10 ZE "entsteht" (Erfassung der Umgebung) ein neuer Prozeß der PK *A_auftrag*, alle 15 ZE ein neuer *B_auftrag*; die Aufträge werden in den zugehörigen Prozeßketten bearbeitet und "verschwinden" (Erfassung der Umgebung) in den abschließenden Quellen.

Ein gleichbedeutendes hierarchisches Modell läßt sich spezifizieren unter Verwendung der in 4.2.5.1 spezifizierten konstruierten FE *fabrik*. Man erhält die Form der Abbildung 4-38.

In Kommentierung:

- In regelmäßigen Abständen entstehen Aufträge bzgl. Produkten A und B. Mit der Fertigung wird die Funktionseinheit *fabrik* beauftragt. Nach Fertigstellung der Produkte werden sie, in Erweiterung obigen flachen Modells, (einzeln) ausgeliefert, was angenommenerweise in jeweils festen Zeitspannen (der Dauer 20.0 bzw. 18.5 ZE), unbeeinflußt vom sonstigen Geschehen, möglich ist.

- Der Ablauf der Fertigung der Produkte ist (in "selbstähnlicher" Weise) innerhalb der Funktionseinheit *fabrik* erklärt (vgl. 4.2.5.1).

- Es wäre offensichtlich leicht möglich, eine weitere Funktionseinheit (z. B. namens *spedition*) zu spezifizieren, welche die Vorgänge des Transports von Produkten in größerem Detail erfassen würde, und welcher die Transportaufträge zu übergeben wären, völlig analog zur Übergabe der Fertigungsaufträge an *fabrik*.

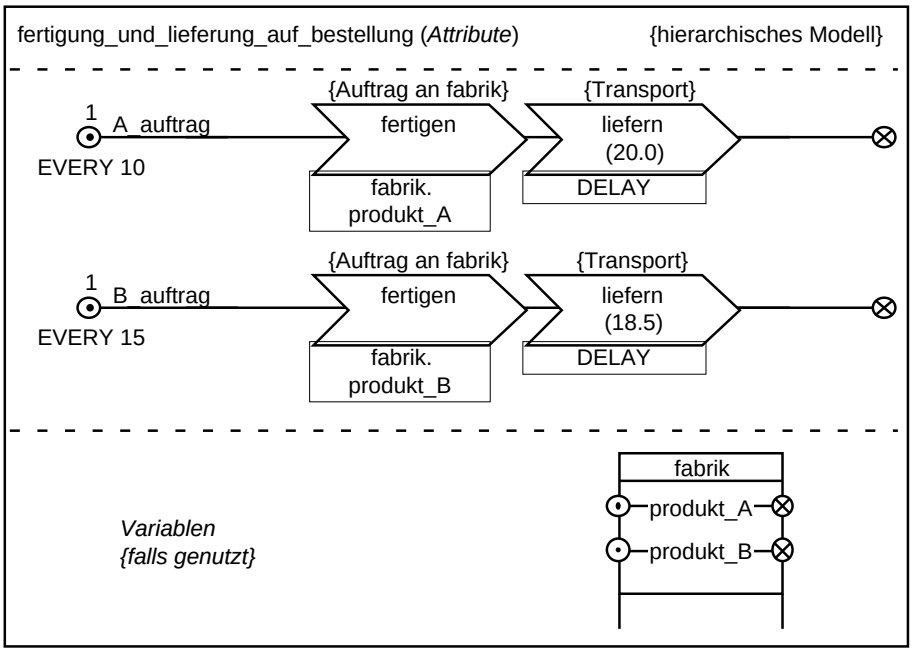


Abbildung 4-38

4.2.6 Attribute, Parameter und Variablen

Attribute, Parameter und Variablen waren schon in Abschnitt 4.2.1 überblicksartig eingeführt worden. Auf höherer Kenntnisstufe bzgl. des B1-Formalismus folgen hier einige Wiederholungen und Ergänzungen.

4.2.6.1 Attribute

Eine Reihe von Konstrukten des B1-Formalismus besitzt initial gewisse Freiheitsgrade bzw. erlaubt es, anläßlich ihrer Konstruktion bestimmte Freiheitsgrade vorzusehen (so daß initial eine ganze Familie "ähnlicher" Konstrukte vorliegt). Diese Freiheitsgrade müssen vor Abschluß der Spezifikation eines Modells konkret ausgefüllt werden (so daß letztlich eine konkrete "Ausprägung" der initial vorgesehenen Familie vorliegt), um nachfolgenden Analyseschritten ein eindeutig bestimmtes Modell zuführen zu können. In diesem Kontext dienen sog. Attributparameter von B1-Konstrukten der Sichtbarmachung von Freiheitsgraden, die Möglichkeiten der wertmäßigen Belegung von Attributparametern der Setzung von Attributen und damit der Ausfüllung der Freiheitsgrade. Explizit erwähnt wurden in den vorangegangenen Abschnitten

- die Attribute des FE-Typs Server (vgl. 4.2.3.1), mithilfe derer eine konkrete Funktionseinheit dieses Typs einen Namen erhielt und ihr Bediengeschwindigkeit, Bediendisziplin und Bedienkapazität zugeordnet werden konnten;

- die Attribute des FE-Typs Counter (vgl. 4.2.3.2), mithilfe derer eine konkrete Funktionseinheit dieses Typs einen Namen erhielt und ihr Dimensionalität, initiale Belegung, Ober- und Untergrenzen der Belegung sowie Wartedisziplin zugeordnet werden konnten;

- die Attributparameter konstruierter Funktionseinheiten (vgl. 0 und 4.2.5.1) und jene von Prozeßkettenmodellen (vgl. 4.2.5.2), über deren Existenz und Bedeutung bei Erstellung dieser Konstrukte freizügig entschieden werden konnte.

Für die Setzung der (Werte der) Attribute stehen zwei Optionen zur Verfügung

- Eine explizite Setzung kann bei Spezifikation des attributbehafteten Konstruktes vorgenommen werden (vgl. Abbildung 4-33, wo die FEs *drehbank*, *fräse* und *packerei* explizite Attributwerte bei Spezifikation erhielten, bzw. ihre Attributwerte auf den DEFAULT-Werten belassen wurden).

- Eine parametrisierte Setzung kann durch Einführung und Nutzung von sog. Attributparametern geschehen, wodurch insbesondere auch die Möglichkeit geschaffen ist, mehreren (wechselseitig wertabhängigen) Attributen in automatisierter Weise wechselseitig konsistente Werte zuzuweisen.

In Wiederholung: Die Setzung von Attributen eines Modells ist (zwingend) vor Analyse dieses Modells abzuschließen. Für eine Analyse (bzw. während eines Analyse"laufs" bei dynamischer, insbesondere simulativer Analyse) sind die gesetzten Attributwerte konstant (unveränderbar).

Seien als Beispiel für die Einrichtung und Nutzung von Attributparametern die Beispiele aus Abschnitt 4.2.5.1 wieder aufgenommen und für den hier vorliegenden Zweck (unter Elimination hier uninteressanter Details) konkretisiert. Abbildung 4-39 zeigt erneut die konstruierte FE *fabrik*, hier versehen mit einem Attributparameter *packer*, dem Sinne nach für die "Anzahl von Packer-Arbeitskräften" der in *fabrik* vorhandenen und genutzten Server-FE *packerei*. Die Attributierung von *packerei* weist als zu setzenden CAP-Wert den Wert dieses Attributparameters aus. Abbildung 4-40 zeigt erneut die Umgebung von *fabrik* unter Setzung ihres Attributs *packer* auf den Wert eines Attributparameters *pack_kapazität*. Der Wert von *pack_kapazität* wird (in Wiederholung des Übergabevorgangs auf nächsthöherer Ebene) aus der Attributierung des Modells *fertigung_und_lieferung_auf_bestellung* bezogen.

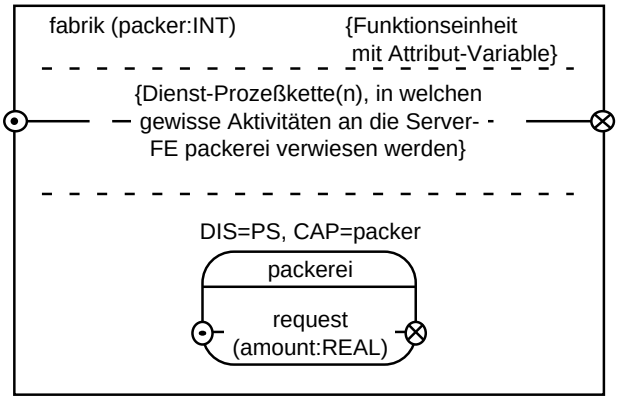


Abbildung 4-39

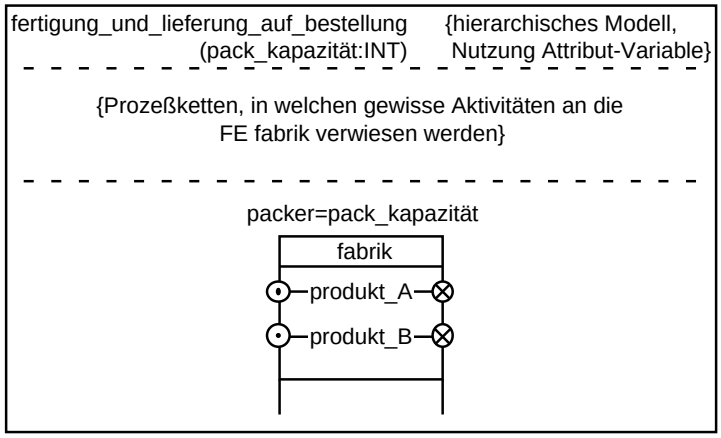


Abbildung 4-40

Eine letztendliche Nutzung des Konzepts der Attributparameter muß in das noch zu erstellende Konzept des Experimentierens mit B1-Modellen eingebunden werden. Ziel ist, in diesem Konzept Expe-

rimente und Experimentserien festlegen zu können etwa der Form (abgestimmt auf das vorliegende Beispiel)

> Analysiere *fertigung_und_lieferung_auf_bestellung*,
> hinsichtlich des Ergebnisses "mittlere Gesamtzeit eines Fertigungs- + Lieferungs-Auftrags",
> für die Fälle von 5, 10 und 15 verfügbarer Packer.

Die Sichtbarkeit von Attributen einer Funktionseinheit bzw. eines Modells (bzw. entsprechender FE- und Modell-Typen), und damit die Nutzbarkeit von Attributwerten, ist auf die jeweilige syntaktische Vereinbarungseinheit beschränkt; beispielsweise kann (vgl. Abbildung 4-40) *pack_kapazität* in den Prozeßketten von *fertigung_und_lieferung_auf_bestellung* genutzt werden (unter Einschluß der Initialisierung von Parametern dieser Prozeßketten), gleichfalls zur Attributierung eingeschlossener Funktionseinheiten (wie im Beispiel für die FE *fabrik* auch durchgeführt) und (analogerweise) zur Initialisierung von Variablen der FE *fabrik* (im Beispiel nicht durchgeführt), nicht aber "außerhalb" von *fertigung_und_lieferung_auf_bestellung*, und auch nicht "innerhalb" eingeschlossener Funktionseinheiten (im Beispiel also nicht innerhalb *fabrik*).

4.2.6.2 Parameter

Die Prozeßketten des B1-Formalismus (vgl. 4.2.2) sind optionalerweise parametrisiert. Diese Parameter sind "Aufruf"-Parameter im Sinne der Funktions- und Prozedurparameter üblicher Programmiersprachen. In der Denkweise dynamischer Analysen (insbesondere also der Simulation) erhalten diesbezügliche "formale" Parameter ihre "aktuellen" Werte anläßlich eines "Aufrufs", im Rahmen des B1-Formalismus also anläßlich der Inkarnierung eines individuellen Prozesses einer parametrisierten Prozeßkette.

Der B1-Formalismus sieht in diesem Kontext sowohl "Eingabeparameter" vor (der Übergabe von Informationen aus der aufrufenden Umgebung einer Prozeßkette an einen inkarnierten Prozeß dienend), als auch "Ausgabeparameter" (der Übergabe von Informationen aus einem - abgeschlossenen - Prozeß an die Umgebung einer Prozeßkette, also der Übergabe von Resultaten dienend). Der (Ein- oder Ausgabe-) Charakter von Parametern wird im B1-Formalismus syntaktisch sichtbar gemacht in der Form

> *Liste_von (P_Name:P_Typ) –> Liste_von (P_Name:P_Typ)*

mit (links des Pfeiles) der Liste der Eingabeparameter inkl. ihres Typs und (rechts des Pfeiles) der Liste der Ausgabeparameter inkl. ihres Typs. Innerhalb der Prozeßkette erfolgt die Setzung des Wertes eines Ausgabeparameter mittels einer üblichen Zuweisungsoperators (in einem PKE des CODE-Typs). Die Sichtbarkeit von Prozeßkettenparametern entspricht jener von Prozeßkettenvariablen (vgl. 4.2.6.3).

4.2.6.3 Variablen

Prozeßketten (vgl. 4.2.2) und Prozeßkettenpläne (vgl. 4.2.4) - und folglich auch Funktionseinheiten und Modelle - boten die Möglichkeit zur Definition eigener Datenräume / Zustandsräume auf dem Wege der Vereinbarung von Variablen. Hinsichtlich der Sichtbarkeit (und damit Verwendbarkeit) von Variablen gilt:

- Die Sichtbarkeit der Variablen einer Prozeßkette ist auf die syntaktische Ebene dieser Prozeßkette beschränkt. Damit sind Variablen und Parameter einer Prozeßkette nicht in anderen Prozeßketten zugreifbar und auch nicht in Funktionseinheiten, an welche Aktivitäten der Prozeßkette verwiesen werden (sofern letzteres gewünscht ist, ist eine explizite Parameterübergabe erforderlich).

- Die Sichtbarkeit der Variablen eines Prozeßkettenplans ist auf die syntaktische Ebene dieses Prozeßkettenplans (d. h. auf die der jeweiligen Funktionseinheit / des jeweiligen Modells) beschränkt. Damit sind Variablen und Parameter eines Prozeßkettenplans in allen Prozeßketten dieses

PK-Plans zugreifbar, nicht aber innerhalb von (in diesem PK-Plan etwa enthaltenen) Funktionseinheiten.

4.2.7 Zusammengesetzte Prozeßketten

Bisher betrachtete Prozeßketten (vgl. 4.2.2) waren auf die Beschreibung einzelner eindeutiger Prozeßträger (einzelner eindeutiger Leistungsobjekte) zugeschnitten. Sie wurden als "einfache Prozeßketten" bezeichnet. Einfache Prozeßketten sind formal dadurch ausgezeichnet, daß sie ein eindeutiges "erstes Element" und ein eindeutiges "letztes Element" (einen eindeutigen Anfang und ein eindeutiges Ende) besitzen.

In Abschnitt 3 war bei Vorstellung diverser Anwendungsfälle bereits darauf hingewiesen worden, daß es praktisch notwendig erscheint, auch die Koexistenz (und wechselseitige Beeinflussung) unterschiedlicher Prozeßträger in Prozeßkettenmodellen erfassen zu können. In den vorangehenden Abschnitten war auch verschiedentlich auf technische Einschränkungen hingewiesen, denen einfache Prozeßketten unterliegen.

Als zusätzliche Motivation für die in diesem Abschnitt erfolgende Einführung zusammengesetzter Prozeßketten sei etwa an die typische Problematik erinnert, die bei Modellierung eines Verteillagers vorliegt, bei dem Transporte unterschiedlicher Zusammensetzung (Produktmix), auf unterschiedlichen Ladungsträgern (Containern, Paletten), auf unterschiedlichen Transportmitteln (LKWs, Zügen, Schiffen) eintreffen und von dem Transporte unterschiedlicher Zusammensetzung, auf unterschiedlichen Ladungsträgern, auf unterschiedlichen Transportmitteln ausgehen. Je nach Interessenlage, und unter Umständen durchaus gemischt, wird das Interesse des Modellierers Einlieferungsaufträgen, Auslieferungsaufträgen, Transporteinheiten, Ladungsträgern, Produktgruppen oder auch einzelnen Produkten als Leistungsobjekten (und damit in prozeßorientierter Auffassung als Prozeßträgern) gelten.

Hinsichtlich Koexistenz und wechselseitiger Beeinflussung unterschiedlicher Prozeßträger erlauben Prozeßkettenpläne (auf Basis einfacher Prozeßketten) bisher bereits auszudrücken:

- die Tatsache der Koexistenz in Form der Möglichkeit, mehrere Prozeßketten (für naheliegenderweise unterschiedliche Prozeßträger) im Rahmen eines Prozeßkettenplans (als Kern sowohl eines Modells als auch einer konstruierten Funktionseinheit) zu spezifizieren;

- wechselseitige Ablaufbeeinflussungen in Form der Möglichkeit, Aktivitäten unterschiedlicher Prozeßketten an identische Funktionseinheiten zur Ausführung zu verweisen und somit wechselseitige "Behinderungen" bei konkurrierender Bearbeitung nachzuvollziehen bis "hinunter" zu den Standard-FEs der Server- und Counter-Typen, deren Funktionalität auf die dynamische Erfassung genau solcher Behinderungen konzentriert ist;

- wechselseitigen Informationsaustausch in Form der Möglichkeit, aus den Aktivitäten unterschiedlicher Prozeßketten auf einen gemeinsamen Datenraum / Zustandsraum zugreifen zu können (nämlich den des beherbergenden Modells bzw. der beherbergenden Funktionseinheit - vgl. *Variablen* und *Attribute* der Abschnitte 4.2.4 und 4.2.5).

Schlecht ausdrückbar dagegen ist bisher:

- die direkte Synchronisation der Aktivitäten von Prozeßträgern unterschiedlicher Prozeßketten, etwa in der Form: Ein individueller Prozeßträger aus Prozeßkette A kann einen spezifischen Punkt in seinem Ablauf erst "überwinden" (d. h. kann, nach Ablauf einer bestimmten seiner Aktivitäten, erst dann seine nächste Aktivität in Angriff nehmen), wenn ein Prozeßträger aus (einer anderen) Prozeßkette B in seinem Ablauf einen spezifischen Punkt "überwunden" hat (in seinem Ablauf eine bestimmte seiner Aktivitäten abgeschlossen hat). Im A-Paradigma sind die sog. "Zeit-Konnektoren" dieser Problemlage gewidmet. Im (bisherigen) B1-Formalismus kann diese Problemlage bisher allenfalls indirekt, unter Einsatz von Counter-FEs, beschrieben werden.

Und praktisch nicht ausdrückbar sind bisher:

- das synchronisierte Entstehen von Prozeßträgern / der synchronisierte Start von Prozessen bzw. Prozeßabschnitten

  (**nicht** wie bisher zu vorbekannten Zeitpunkten, vgl. die Unbedingten Quellen des Abschnitts 4.2.2; und auch **nicht** als synchrone Einschachtelung von hierarchisch tieferen Prozeßabschnitten, vgl. die Dienstaufrufe des Abschnitts 4.2.3, sondern:)

  etwa in der Form: Wenn ein individueller Prozeßträger aus Prozeßkette A einen spezifischen Punkt seines Ablaufs erreicht hat, dann (neben anderen Folgen dieses Erreichens) startet auch ein Individuum der Prozeßkette B.

  Dies ist die typische Sachlage bei Vorgängen des Entladens, Depalettierens, Auspackens.

- das synchronisierte Verschwinden von Prozeßträgern / die synchronisierte Beendigung von Prozessen bzw. Prozeßabschnitten

  (**nicht** wie bisher synchron zum Ablauf des individuellen Prozesses, vgl. die Unbedingten Senken des Abschnitts 4.2.2; und auch **nicht** zu Ende der synchronen Einschachtelung hierarchisch tieferer Prozeßabschnitte, vgl. die Dienstaufrufe des Abschnitts 4.2.3, sondern:)

  etwa in der Form: Wenn ein individueller Prozeßträger aus Prozeßkette A einen spezifischen Punkt seines (A-)Ablaufs erreicht hat, und auch ein Individuum einer (anderen) Prozeßkette B einen spezifischen Punkt seines (B-)Ablaufs erreicht, dann existiert keine Fortsetzung für den Prozeßträger aus A (verschwindet der Prozeßträger aus A / endet der individuelle A-Prozeß).

  Dies ist die typische Sachlage bei Vorgängen des Beladens, Palettierens, Packens.

Das A-Paradigma setzt gelegentlich (Öffnende bzw. Schließende) UND-Konnektoren zur Erfassung der letztgenannten beiden Problemlagen ein. Es handelt sich dabei allerdings darstellungsmäßig ("zeichnerisch") um dieselben Symbole wie die (im vorliegenden Kontext) in Abschnitt 4.2.2 beschriebenen, so daß die beiden (deutlich unterschiedlichen) Bedeutungen schlecht separierbar (und daher schlecht in eine automatische Modellanalyse umsetzbar) sind.

(NB: Obige Schilderung konzentrierte sich auf eine Art prozeß- und prozeßketten-übergreifenden UND-Konnektor. Aus Analogie-Überlegungen heraus ließe sich auch eine Art prozeßübergreifender ODER-Konnektor motivieren. Andererseits haben unsere bisherigen Kontakte mit den Anwendungsprojekten bisher keine überzeugende praktische Notwendigkeit für einen derartigen Konnektortyp erbracht, so daß diese Linie im Moment nicht weiter verfolgt wird.)

4.2.7.1 Prozeßkettenkonnektoren (PK-Konnektoren)

Der skizzierten Mehrdeutigkeit von Konnektoren wird hier mit dem Vorschlag der Einführung eines zusätzlichen Konnektor-Symbols begegnet. Die einzuführenden **Prozeßkettenkonnektoren** ähneln in der Darstellungsform den UND-Konnektoren, sind aber von diesen unterscheidbar. Prozeßkettenkonnektoren sind, grob gesprochen, mit dem Ziel entwickelt, Anfänge bzw. Enden von einfachen Prozeßketten zu synchronisieren. In der folgenden Darstellung wird von einem (Meta-) Symbol für einfache Prozeßketten Gebrauch gemacht, das **nicht** den B1-Modellspezifikations-Symbolen zugehört, sondern allein der Erklärung dient: Dieses Symbol

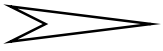


Abbildung 4-41

steht für eine einfache Prozeßkette, ein Konstrukt (entsprechend Abschnitt 4.2.2) bestehend aus PKEs, UND- und ODER-Konnektoren sowie deren Verbindungen (aber ohne Quellen und Senken), mit genau einem Anfangs- und genau einem Ende-Element.

Im Folgenden wird ein Öffnender und ein Schließender Prozeßkettenkonnektor definiert, wobei der Öffnende der Problematik synchronisierter Prozeßentstehungen, der Schließende jener synchronisierter Prozeßbeendigungen gewidmet ist.

Öffnender PK-Konnektor, symbolisch:

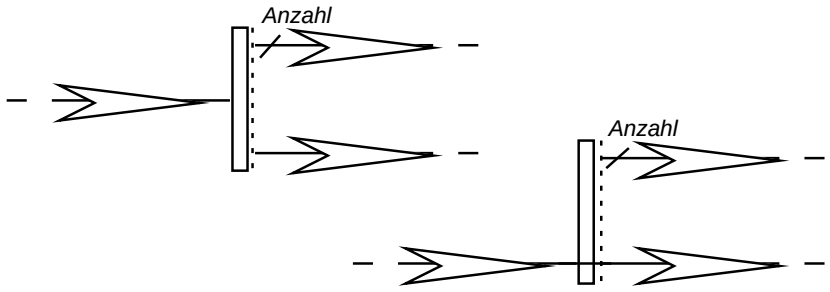


Abbildung 4-42

In beiden angebotenen Formen wird ausgedrückt, daß nach jeder Beendigung eines Prozesses der "linken" (einfachen) Prozeßkette Prozesse der "rechten" (einfachen) Prozeßketten simultan beginnen. In beiden Formen sind mehr als (wie hier skizziert) zwei parallele rechte Prozeßketten zulässig. In beiden Formen existieren als

Attribute:

· *Anzahl* (optional): ganze Zahl (arithmetischer Ausdruck); als Notation für den Start einer Menge von *Anzahl* paralleler Prozesse gemäß bezeichneter rechter Prozeßkette; DEFAULT: *Anzahl* = 1.

Der Unterschied der beiden Formen besteht darin, daß die Form links oben anzeigt, daß alle rechten Prozesse neu starten, während in der Form rechts unten (mittels durchgezogener Verbindung) angezeigt wird, daß der linke (vergangene) Prozeß sich in einem rechten (zukünftigen) Prozeß fortsetzt.

Man beachte, daß mit dem Öffnenden Prozeßkettenkonnektor ein Konstrukt eingeführt wird, das Quellencharakter besitzt. In dieser Hinsicht zeigt ein Vergleich mit der Definition einfacher Prozeßketten (vgl. 4.2.2), daß obige Notation u. U. nicht vollständig ist. Einfache Prozeßketten sind (optionalerweise) benannt, parameterbehaftet und mit Variablen versehen, wobei die spezifische Wahl der Optionen aus der Spezifikation der jeweiligen Prozeßkette hervorgeht, die Setzung von Parametern aber der jeweiligen Quelle obliegt. Die vollständige Form der Spezifikation des Öffnenden Prozeßkettenkonnektors wird daher festgelegt zu:

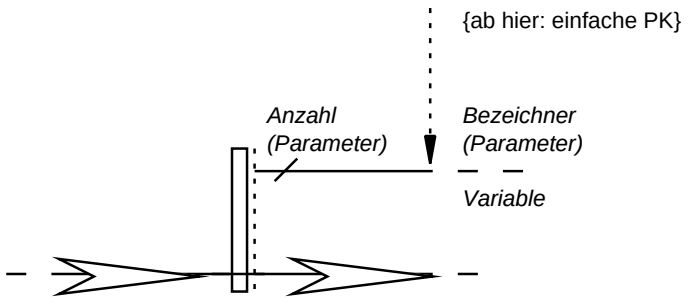


Abbildung 4-43

Eine rechte PK, deren Prozesse als Fortsetzung von Prozessen der linken PK gekennzeichnet sind, benötigt diese Vervollständigung nicht (hier setzen sich individuelle Prozesse fort, deren Parametrisierung bereits "zuvor" geschah und die ihren Datenraum bereits besitzen und samt Variablenwerten be-

halten). Bei allen neu einsetzenden rechten Prozessen sind ggf. (vgl. Quelle, 4.2.2.3.1) Parameterwerte zu setzen in Form der (dem jeweiligen PK-Konnektor-Ausgang zugeordneten) zusätzlichen Attribute:

· *Parameter*: (aktuelle) Werte (Ausdrücke) für die (formalen) Parameter der verbundenen Prozeßkette; in Zahl und Typ auf die Parameter dieser Prozeßkette abgestimmt;

Schließender PK-Konnektor, symbolisch:

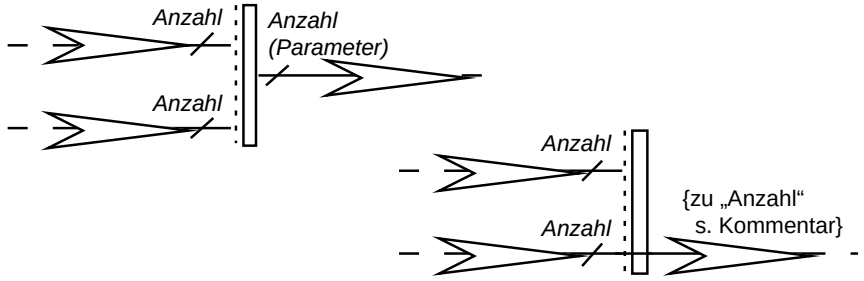


Abbildung 4-44

In beiden Formen wird ausgedrückt, daß nach der Beendigung spezifischer Anzahlen von Prozessen der "linken" Prozeßketten eine spezifische Anzahl von Prozessen der "rechten" Prozeßkette simultan beginnen. In beiden Formen sind mehr als (wie hier skizziert) zwei parallele linke Prozeßketten zulässig. In beiden Formen existieren als

Attribute:

· *Anzahl* (links, optional): ganze Zahl (arithmetischer Ausdruck); als Bedingung dafür, wieviele parallele Prozesse gemäß bezeichneter linker Prozeßkette beendigt sein müssen, um eine rechte Fortsetzung zu ermöglichen; DEFAULT-Wert: *Anzahl* = 1.

· *Anzahl* (rechts, optional): ganze Zahl (arithmetischer Ausdruck); als Festlegung dafür, wieviele Prozesse gemäß bezeichneter rechter Prozeßkette gestartet werden; DEFAULT-Wert: *Anzahl* = 1.

Die Form links oben zeigt an, daß alle rechten Prozesse (der einen rechten PK) neu starten, während in der Form rechts unten (mittels durchgezogener Verbindung) angezeigt wird, daß Prozesse einer der linken PKs sich in rechten Prozessen fortsetzen. Die obere linke Form benötigt daher ggf. (s. oben)

zusätzliche Attribute:

· *Parameter* (rechts): Werte (Ausdrücke) für die Parameter der verbundenen Prozeßkette; in Zahl und Typ auf die Parameter dieser Prozeßkette abgestimmt;

wohingegen diese Notwendigkeit bei der rechten unteren Form entfällt; auch ist bei dieser Fortsetzungsform eine *Anzahl*-Angabe nicht vorgesehen (Prozeß-Individuen in der links aufscheinenden *Anzahl* werden fortgesetzt).

Man beachte, daß mit dem Schließenden Prozeßkettenkonnektor ein Konstrukt eingeführt wird, das Senkencharakter besitzt.

Gemischte PK-Konnektoren, welche sowohl schließende (d. h. synchronisierende) als auch öffnende (d. h. parallelisierende) Charakteristika gemeinsam aufweisen, sind zugelassen. Sie besitzen sowohl Senken- als auch Quellencharakter.

Gemischter PK-Konnektor, symbolisch:

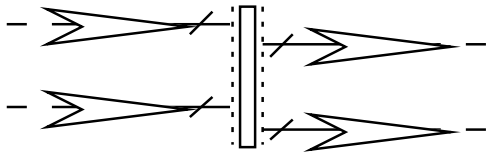


Abbildung 4-45

Attribute in Kombination der Angaben der nicht gemischten PK-Konnektoren. Als Zusatz gilt, daß Prozesse mehrerer der linken Prozeßketten eine direkte Fortsetzung in Prozessen rechter (mittels durchgezogener Verbindungen zugeordneter) Prozeßketten finden dürfen. Ein Konstrukt, das ausschließlich solche direkten Fortsetzungen enthält, ist i. w. identisch zum Zeitkonnektor des A-Paradigmas.

4.2.7.2 Bedingte Quellen und Senken

Die Einführung der Prozeßkettenkonnektoren, obzwar problemseitig motiviert, führt zu zusätzlichem Erklärungsbedarf bei hierarchischen Prozeßkettenmodellen. Enthält nämlich eine konstruierte Funktionseinheit als Bestandteil einer Prozeßkette

- einen Öffnenden Prozeßkettenkonnektor, dann können innerhalb der FE, während des dynamischen Ablaufs eines aus der Umgebung der FE angestoßenen Prozesses, neue Prozesse entstehen, welche (zumindest potentiell) in einer virtuelle Senke enden, dem Sinne nach also einer Fortsetzung in der FE-Umgebung bedürfen - obwohl das Erreichen der virtuellen Senke nicht (wie in den bisher betrachteten Fällen) als Rückmeldung eines umgebungsseitig angeforderten Dienstes interpretierbar ist;

- einen Schließenden Prozeßkettenkonnektor, dann kann ein aus der Umgebung der FE angestoßener Prozeß innerhalb der FE, an einem bestimmten Punkt seines dynamischen Ablaufs, beendet werden, ohne eine virtuelle Senke erreicht zu haben – also ohne sich (wie in den bisher betrachteten Fällen) umgebungsseitig zurückzumelden.

Die hier einzuführenden Bedingten Quellen und Senken widmen sich der Beseitigung dieser Schwierigkeiten.

Bedingte Quelle (symbolisch):

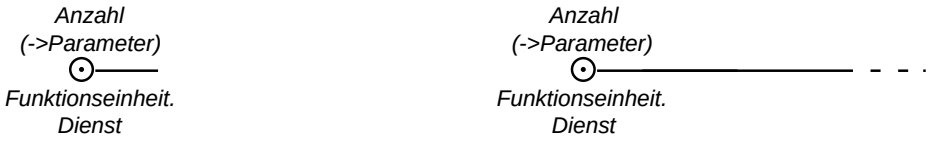


Abbildung 4-46

Links ist die Bedingte Quelle für sich dargestellt, rechts (der Deutlichkeit halber) die Bedingte Quelle, verbunden mit einer Prozeßkette (vgl. 4.2.2).

Die Darstellung symbolisiert, daß aus einem Dienst einer Funktionseinheit (*Funktionseinheit.Dienst*) heraus angezeigte Beendigungen FE-interner Prozesse in den Start der mit der Quelle verbundenen Prozeßkette (im Beispiel mit Namen entsprechend *Bezeichner*) umgesetzt werden. Diese Prozesse entstehen in einer Vielfachheit gemäß *Anzahl*. Die Setzung der PK-*Parameter* (falls vorgesehen) geschieht entsprechend der Resultat-*Parameter* der abgelaufenen internen Prozesse.

Voraussetzung des Einsatzes der Bedingten Quelle ist die Existenz einer entsprechenden

Funktionseinheit mit Quellenfunktionalität

symbolisch, externe Sicht:

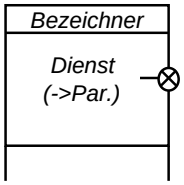


Abbildung 4-47

eine entsprechende (mögliche) konstruierte FE mit Quellenfunktionalität hier als

Beispiel, externe Sicht:

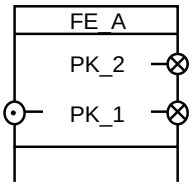


Abbildung 4-48

Beispiel, interne Sicht:

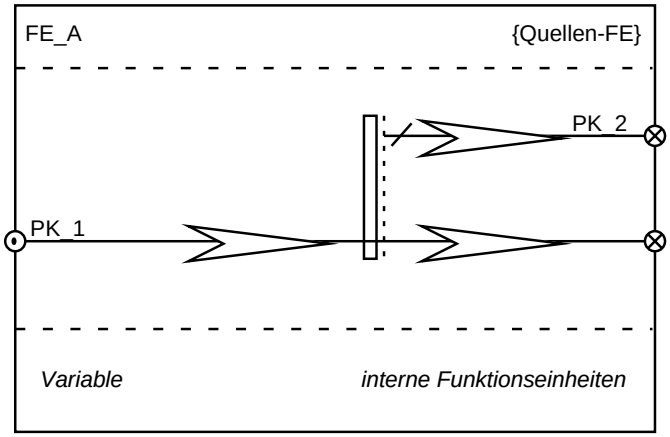


Abbildung 4-49

Während jedes Aufrufs des Dienstes mit PK_1 entsteht am Prozeßkettenkonnektor ein Prozeß gemäß Prozeßkette mit PK_*2*, dessen (interne) Beendigung für die Umgebung sichtbar ist und dort als Quellenereignis fungiert. Eine Parameterübergabe ist im Beispiel nicht vorgesehen.

Bedingte Senke (symbolisch, externe Sicht):

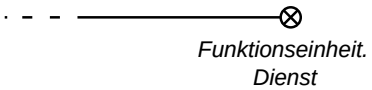


Abbildung 4-50

Die Darstellung symbolisiert, daß ein Dienst einer Funktionseinheit (*Funktionseinheit.Dienst*) in Anspruch genommen wird, welcher einen FE-internen Prozeß anstößt, wobei dieser Prozeß innerhalb der FE endet, ohne eine diesbezügliche Anzeige nach außen zu geben. Im Unterschied zur Unbedingten Quelle, bei deren Erreichen Prozesse der mit der Senke verbundenen Prozeßkette unmittelbar enden,

dauern hier die von einem solchen Prozeß verursachten Vorgängen "noch eine Weile an", ohne daß dies auf der Aufrufebene sichtbar und meßbar wäre.

Voraussetzung des Einsatzes der Bedingten Senke ist die Existenz einer entsprechenden

Funktionseinheit mit Senkenfunktionalität

symbolisch, externe Sicht:

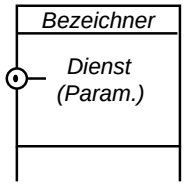


Abbildung 4-51

eine entsprechende (mögliche) konstruierte FE mit Senkenfunktionalität hier als

Beispiel, externe Sicht:

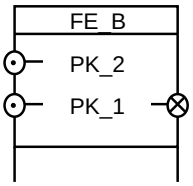


Abbildung 4-52

Beispiel, interne Sicht:

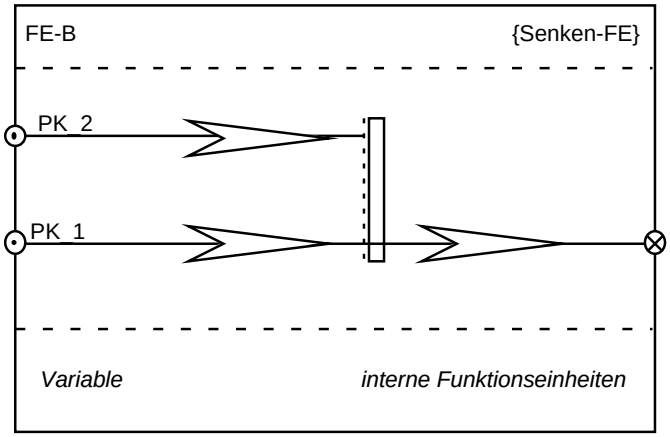


Abbildung 4-53

Jeder Aufruf des Dienstes PK_2 endet am Prozeßkettenkonnektor, synchron mit dem Erreichen dieses Prozeßkettenkonnektors seitens einen Prozesses gemäß Prozeßkette PK_1. Der Zeitpunkt der Beendigung von PK_2-Prozessen ist in der Umgebung nicht sichtbar.

4.2.8 Über hierarchische Beziehungen

Überblicksabschnitt 4.1 wies kursorisch auf die Möglichkeiten hierarchischer Strukturierung von Modellen im Rahmen des B1-Formalismus hin - die folgenden Abschnitte stellten schrittweise verschiedene Einzelaspekte dieser Möglichkeiten vor. Insgesamt wurde zwischen zwei unterschiedlichen (gleichzeitig vorliegenden) hierarchischen Relationen unterschieden,

· einerseits der weithin bekannten funktionalen **Aufruf-Relation**, welche in der Form unterstützt ist, daß

  - die Definition einer Prozeßkette diese in eine Menge von Aktivitäten (dargestellt durch PK-Elemente) zerlegt, sowie Reihenfolgevorschriften über den Aktivitäten (dargestellt durch Verbindungen und Konnektoren) festlegt; der B1-Formalismus interpretiert diese Festlegungen im Ablauf von Prozessen (für jeden individuellen Prozeß dieser Prozeßkette) als Forderung, die genannten Aktivitäten de facto auszuführen / durchzuführen, und sie in der festgelegten Abfolge auszuführen,

  - durchzuführende Aktivitäten zum Zwecke ihrer Ausführung an (spezifische Dienste von) Funktionseinheiten verwiesen werden; die Erklärung der Arbeitsweise von Funktionseinheiten (bei Ausführung angeforderter Dienste) erfolgt ("selbstähnlicherweise") durch den Diensten zugeordnete Prozeßketten, mit ihren Aktivitäten, Reihenfolgevorschriften und Ausführungsverweisen.

  In dynamischer Sicht ist dies genau die gewohnte Aufrufhierarchie üblicher Programmiersprachen (mit dem "ungewohnten" Nebeneffekt während des Ablaufes verstreichender, genau definierter Modellzeit).

· andererseits der gleichfalls weithin bekannten strukturellen **Enthaltensein-Relation**, welche in der Form unterstützt ist, daß

  - Funktionseinheiten (und auf "oberster" Ebene: Modelle) bei ihren Tätigkeiten (nämlich der Durchführung von Prozessen) die Fähigkeiten von "ihren" (d. h. in ihnen enthaltenen, in ihrer Regie und Verantwortung liegenden) Funktionseinheiten in Anspruch nehmen (nämlich diese zur Ausführung von Aktivitäten anhalten),

  - wobei die enthaltenen Funktionseinheiten auf wiederum ("selbstähnlicherweise") in ihnen enthaltene Funktionseinheiten rekurrieren (bis hin zu den "untersten", nicht mehr aufgelösten Ebenen der Standard-Funktionseinheiten).

  In struktureller Sicht ist dies genau die gewohnte Enthaltenseinhierarchie blockstrukturierter Programmiersprachen.

Die modelltechnische Nutzung der Hierarchisierungsoptionen in Form der Festlegung einer spezifischen Modellstruktur ist dem Anwender freigestellt und kann auf seine Modellierungsziele abgestellt erfolgen. Eine problemnahe (und daher empfehlenswerte) Art der Nutzung bestünde darin, strukturelle Aspekte des zu erfassenden Realsystems in der gewählten Modellstruktur widerzuspiegeln.

Beispielsweise könnte ein Firmenverbund,

> welcher eine Reihe von Firmen umfaßt, wo jede der Firmen in eine Reihe von Abteilungen gegliedert ist, jede Abteilung eine Reihe von Arbeitsstätten umschließt, jede Arbeitsstätte ihre Basis-Ressourcen aufweist,

in einem B1-Modell erfaßt werden,

> welches (unter Nutzung der Enthaltensein-Hierarchie:) eine entsprechende Menge von Firmen-FEs enthält, deren jede ihre spezifischen Abteilungs-FEs enthält, deren jede ihre Arbeitsstätten-FEs enthält, deren jede entsprechende Ressourcen-FEs enthält,

sowie gleichzeitig die in diesem Firmenverbund offensichtlich abzuwickelnden Anfrage- und Auftrags-Vorgänge zwischen globaleren Organisationseinheiten als Auftraggeber und ihren jeweiligen Auftragnehmern (unter Nutzung der Aufruf-Hierarchie:) durch entsprechende Ausführungsverweise von Aktivitäten an Dienste enthaltener FEs erfaßt werden.

Die Gesamtstruktur eines B1-Modells ist bzgl. der Enthaltensein-Relation offensichtlich streng baumförmig, bzgl. der Aufruf-Relation durch die baumförmige Enthaltensein-Relation deutlich eingeschränkt. Das in Abschnitt 4.1 abschließend aufgestellte Versprechen, die Modell-Gliederungsstruktur sei "nicht auf Baumstrukturen reduziert", ist somit bisher nicht eingelöst.

4.2.8.1 Gemeinsam genutzte Funktionseinheiten

Das Konzept der "gemeinsam genutzten Funktionseinheiten" dient dazu, gewisse Einschränkungen der Aufruf-Relation aufzuheben, wie sie durch die streng baumförmige Struktur der Enthaltensein-Relation bedingt sind. Zu diesem Zweck wird innerhalb von FEs die Möglichkeit geschaffen, Aktivitäten enthaltener Prozeßketten (nicht nur an Dienste enthaltener FEs, sondern auch:) an Dienste "unbekannter", "externer" FEs zu verweisen.

In einem Beispielsszenario sei im Betriebsablauf einer *Firma_1* eine Transporttätigkeit erforderlich, die (zumindest optionalerweise) als Fremdauftrag vergeben werden soll, etwa an einen Transportdienstleister, dessen Identität aber nicht vorab festgelegt ist. Der B1-Formalismus bietet dazu (vgl. Abbildung 4-54) die Möglichkeit,

- den internen FEs einer Funktionseinheit (hier der *Firma_1*) eine Pseudo-FE *EXTERNAL* zur Seite zu stellen,
- bei dieser die Verfügbarkeit bestimmter Dienste vorauszusetzen (hier: *transport* und *sonstwas*)
- und diese Dienste zu nutzen (letzteres in der üblichen Weise des Verweisens von Aktivitäten an *EXTERNAL*-Dienste, hier bei *bewegen* in beiden enthaltenen Prozeßketten *PK_1* und *PK_2*).

Die Symbolik des "unten offen" der FE *Firma_1* soll die Tatsache ihrer Ergänzungsnotwendigkeit in bezug auf genutzte Dienste signalisieren.

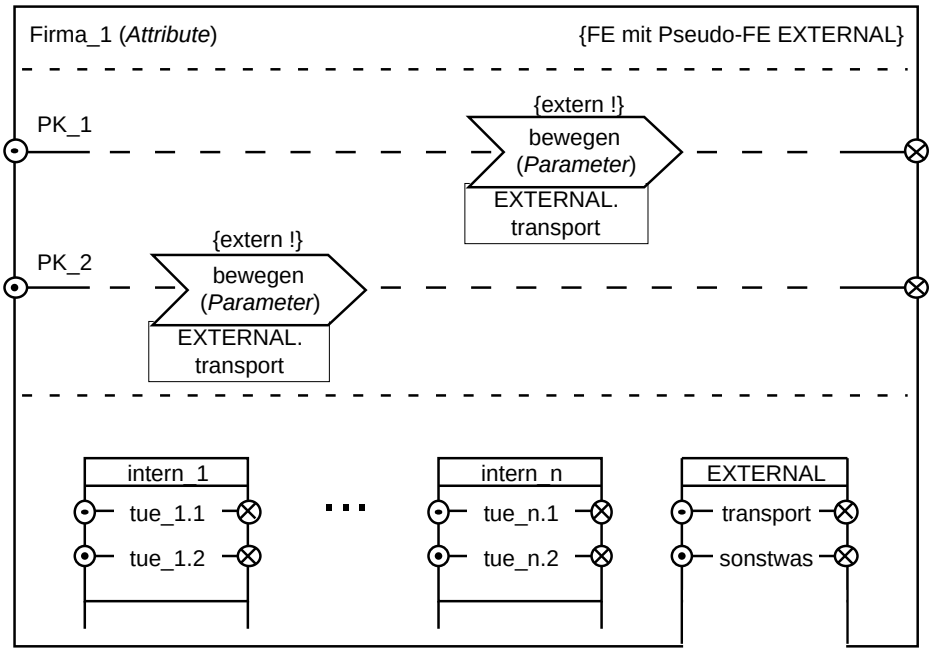


Abbildung 4-54

In der externen Sicht von *Fabrik_1* (vgl. Abbildung 4-55) wird diese Ergänzungsnotwendigkeit erneut sichtbar, hier in einer Form, die es erlaubt, die Ergänzung sinnvoll vorzunehmen. Konkret handelt es sich dabei um die (bisher nicht erklärten) Verweislisten konstruierter Funktionseinheiten (vgl. Abschnitt 4.2.3.3 bzw. Abbildung 4-32 und spätere).

Um im Beispielsszenario zu bleiben, sei angenommen, *Firma_1* sei in einem Modell *Firmen_und_Speditionen* enthalten, das auch eine Speditions-FE *Sped_A* enthält, deren angebotener Transport-)Dienst *t_auftrag* für die Transporte genutzt werden soll. Diese Entscheidung ist als Teil der Verweisliste (von *Firma_1*) mittels des Schlüsselworts TO ausgedrückt (vgl. Abbildung 4-55). Darüber hinaus sind weitere Firmen skizziert (welche alle Transportunterstützung "von außen" benötigen könnten) und weitere Speditionen eingezeichnet (welche alle Transportdienste anbieten könnten).

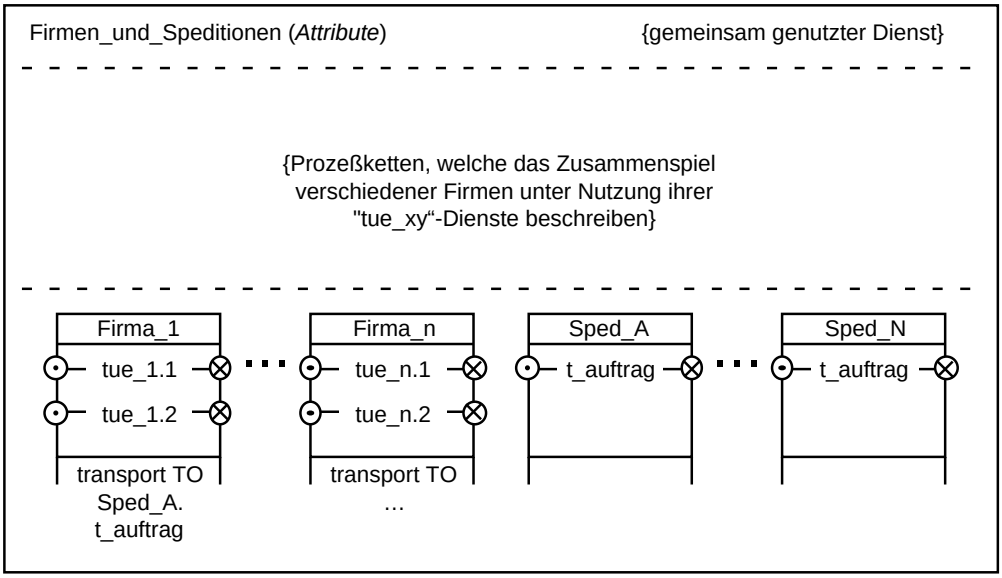


Abbildung 4-55

Die Entscheidungen darüber, welche der Firmen sich welcher der Speditionen bedienen, kann freizügig spezifiziert werden. Insbesondere läßt sich festlegen, daß sowohl *Firma_1* als auch *Firma_n* sich für ihre Transporte derselben *Sped_A* bedienen; diese ist "gemeinsam genutzt", ohne in *Firma_1* oder *Firma_n* enthalten zu sein: Die beschränkenden Auswirkungen der strikten Baumstruktur der Enthaltenseins-Relation auf die Aufruf-Relation sind weitgehend beseitigt.

Bei Nutzung der Option der gemeinsam genutzten Funktionseinheiten ist darauf zu achten, daß die Aufruf-Relation nicht zirkulär ("rekursiv") wird.

4.2.9 Ausblick

Bei Entwurf eines Modellformalismus, der die Prüfung seiner Akzeptanzfähigkeit, und in der Folge seine Realisierung, noch vor sich hat, werden sinnvollerweise nicht alle Details bis ins letzte festgelegt. Offen sind beim vorliegenden Entwurf insbesondere

- eine ganze Reihe von Syntaxfragen,
- Entscheidungen über das "Ingangsetzen" / Starten von Quellen-Einheiten,
- Mechanismen des "Typisierens" konstruierter Funktionseinheiten,
- das Zusammenspiel mit einer komfortablen "Analyse-Umgebung",
- die Frage der Nutzbarkeit des vorgelegten Attributierungs-Konzepts für Optimierungsaufgaben,

Es besteht die Absicht, diese (und potentiell andere) "Lücken" in Zusammenarbeit mit anwendenden Teilprojekten, und in Berücksichtigung ihrer spezifischen Modellierungsaufgaben, zu füllen.

5 Anwendungsbeispiele – erneut betrachtet

In diesem Kapitel werden die Anwendungsbeispiele aus Kapitel 3 erneut betrachtet und mit den Modellierungskonstrukten aus Kapitel 4 beschrieben. Neben der prinzipiellen Modellierbarkeit der Beispiele mittels des B1-Paradigmas soll dieses Kapitel auch die Modellierbarkeit problematischer Systemcharakteristika aufzeigen.

5.1 Güterverteilzentrum (GVZ)

Im folgenden wird das Beispiel des Güterverteilzentrums aufgegriffen und mittels unseres in Kapitel 4 definierten B1-Modellformalismus modelliert.

Abstrakt betrachtet, läßt sich das GVZ als Funktionseinheit ansehen, welches Dienstleistungen (Dienste) für Züge und LKWs anbietet. Wie aus der Beschreibung in Kapitel 3 ersichtlich sind die Abläufe innerhalb des GVZ relativ komplexer Natur. Daher wollen wir das Modell schrittweise entwickeln und zuerst den einfacheren Fall der KV-affinen Verkehre betrachten und annehmen, daß Huckepack-Verkehre nicht ankommen und sich auch nicht auf dem Zug befinden. In unserem Formalismus sieht eine mögliche Beschreibung einer abstrakten Sicht des GVZ wie folgt aus

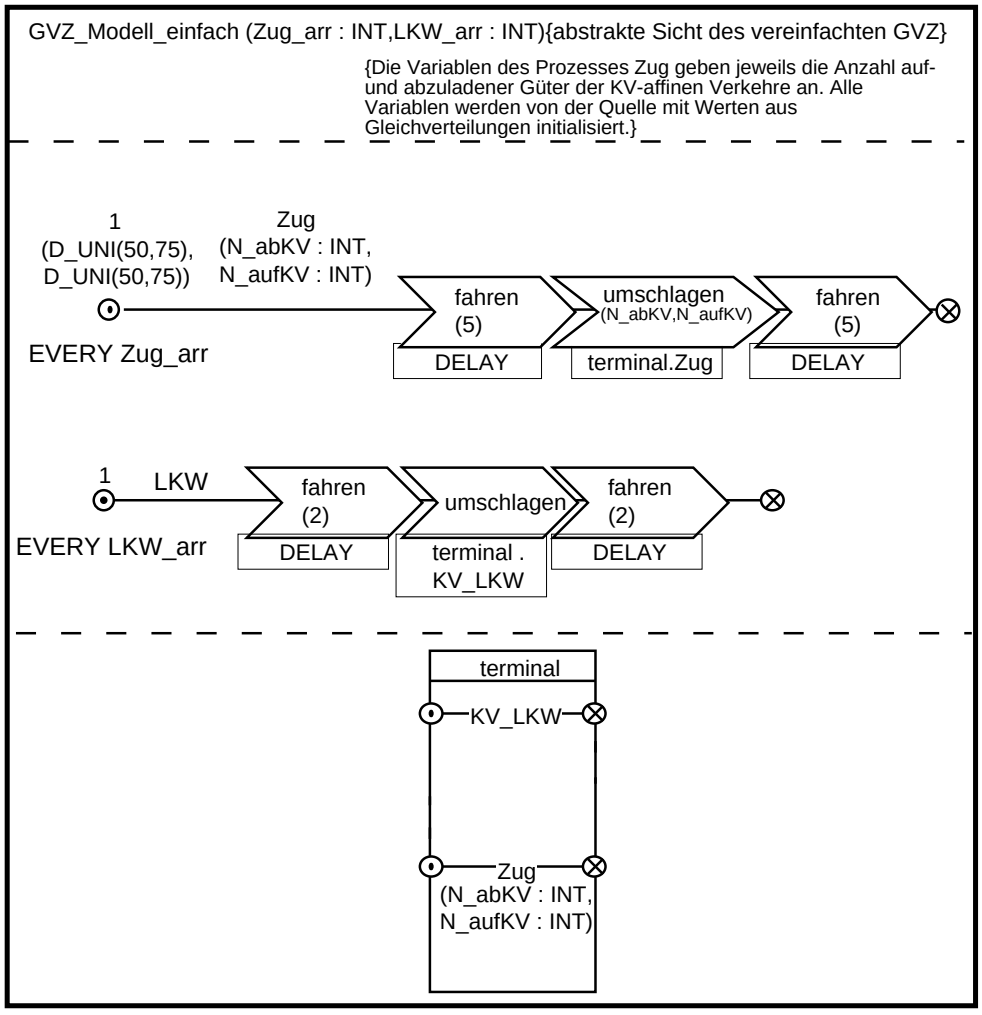


Abbildung 5-1 Abstrakte Sicht der KV-affinen Verkehre

Die gesamte Modellbeschreibung ist in drei Teile eingeteilt, welche durch die gestrichelten Linien angedeutet werden. Zuoberst ist der Name des Modells (*GVZ_Modell_einfach*) angegeben, in der

Mitte eine Menge von einfachen Prozeßketten und im unteren Teil Funktionseinheiten (hier nur *terminal*) auf deren Dienste sich die Aktivitäten der einzelnen Prozeßkettenelemente abstützen.

Auf den Namen des Modells im obersten Teil der Beschreibung folgt die Angabe zweier Attribute vom Typ *INT* (*Zug_arr, LKW_arr*). Diese Attribute müssen gesetzt werden, um ein ausführbares Modell zu beschreiben. Die o. g. Attribute werden zur Spezifikation der unbedingten Quellen benutzt. Beispielsweise wäre *GVZ_Modell(480,10)* eine mögliche Setzung, welche angibt, daß alle 480 Zeiteinheiten (ZE) ein Zug ankommt und alle 10 ZE ein LKW.

Die Ankünfte von Zügen und LKWs werden durch Quellen modelliert, die in gewissen zeitlichen Abständen Prozesse vom Typ *Zug* und *LKW* generieren. Diese Prozesse sind beschrieben durch die identischen Sequenzen *fahren-umschlagen-fahren*. Jeder Prozeß vom Typ *Zug* besitzt 2 Variablen vom Typ *INT*, die bei Erzeugung eines Prozesses (alle *Zug_arr* Zeiteinheiten) aus Gleichverteilungen gesetzt werden. So wird beispielsweise der Variablen *N_abKV* gleichverteilt ein Wert zwischen 50 und 75 zugewiesen. Die beiden Variablen spezifizieren die Ladung bzw. die Aufnahmekapazität eines Zuges. Zum Beispiel bezeichnet *N_abKV* die auf dem Zug befindliche Anzahl abzuladender KV-affiner Güter. Die Variablen *N_abKV, N_aufKV* eines Prozesses vom Typ *Zug* werden innerhalb der Funktionseinheit *terminal* zur Modellierung der Ent- und Beladevorgänge benutzt. Diese Tatsache ist im Modell an zwei Stellen ersichtlich. Die Aktivität *umschlagen* des Prozeßmusters *Zug* stützt sich auf den Dienst Zug der Funktionseinheit *terminal* ab und übergibt hierzu die Werte der beiden Variablen als Parameter an diesen Dienst. Ferner wird in der externen Sicht der Funktionseinheit *terminal* die notwendige Angabe der beiden Parameter spezifiziert. Analog ist zur Behandlung der LKWs mit KV-affinen Gütern der Dienst *KV_LKW* vorgesehen. Dieser Dienst wird ohne Angabe von Parametern genutzt.

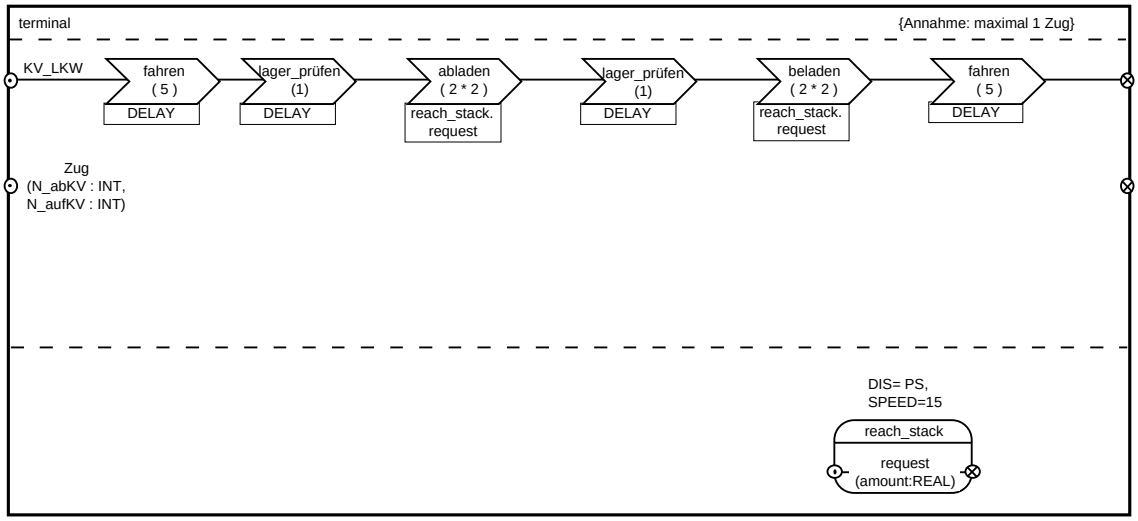


Abbildung 5-2 Der Dienst KV_LKW innerhalb der Funktionseinheit terminal

In der Funktionseinheit *terminal* werden die angesprochenen Dienste nun genauer spezifiziert (vgl. Abbildung 5-2). Betrachten wir zunächst den KV-affinen Verkehr und überlegen uns welche Funktionseinheiten innerhalb der Funktionseinheit *terminal* benötigt werden, um ein Be- und Entladen des KV-affinen Verkehrs zu bewerkstelligen. Im wesentlichen sind hier die Reach-Stacker (spezielle Gabelstapler) zu nennen, die die Container von den LKWs bzw. auf diese heben.[4] Insgesamt könnte eine Modellierung des KV-affinen Verkehrs innerhalb des *terminal* durch den Ausschnitt aus Abbildung 5-2 beschrieben werden. Der Dienstaufruf *KV_LKW* bewirkt, daß der aufrufende Prozeß nacheinander die dargestellten Tätigkeiten durchführt. Die Erbringung einiger dieser Tätigkeiten wird wiederum durch das Dienstangebot der Reach-Stacker (einer internen Funktionseinheit des *terminal*) ermöglicht. Die Reach-Stacker werden hier vereinfachend durch eine geeignet parametrisierte Stan-

[4] Die Reach-Stacker bringen die abgeladen Container nachfolgend in ein Lager bzw. holen Container aus diesem Lager zwecks Beladung. In der Modellierung könnte man das Lager als Funktionseinheit innerhalb der Reach-Stacker ansehen. Um die Beschreibung nicht zu verkomplizieren, wollen wir auf eine solche Verfeinerung verzichten.

dard-Funktionseinheit *reach_stack*, einem Server, dargestellt, dessen Dienst *request* jeweils zum Be- und Entladen (Tätigkeiten *abladen* und *beladen*) genutzt wird. Dabei wird angenommen, daß alle Aufträge, die an die Funktionseinheit *reach_stack* verwiesen werden von dieser gleichzeitig mit evtl. verminderter Kapazität bedient werden (*DIS=PS*) und daß eine Auftragseinheit in 1/15 ZE bearbeitet werden kann (*SPEED=15*). Bei entsprechender Wahl der Basiseinheiten kann die „SPEED“ z. B. die Anzahl der Gabelstapler bezeichnen. Der Dienst der Funktionseinheit *reach_stack* wird von den Prozessen bei Ausführung der Tätigkeiten *abladen* und *beladen* genutzt und mit konstanten Parameterwerten (hier ein konstanter arithmetischer Ausdruck) aufgerufen. Bei den Tätigkeiten *fahren* und *lager_prüfen* handelt es sich um reine Verzögerungen. Die numerischen Werte sind willkürlich gewählt worden.

Die Aktivitäten des Zuges innerhalb der Funktionseinheit *terminal* sind in Abbildung 5-3 beschrieben, wobei angenommen wurde, daß das Be- und Entladen eines KV-affinen Gutes 2 Zeiteinheiten benötigt. Die entsprechenden Aktivitäten stützen sich ebenfalls auf das Dienstangebot der Gabelstapler (SERVER *reach_stack*) ab. Da die jeweiligen Aktivitäten parallel (nebenläufig) zu den Aktivitäten der LKWs ablaufen, kann es hier zu einer konkurrierenden Nutzung der Standard-Funktionseinheit *reach_stack* kommen, so daß sich die Beendigung der jeweiligen Aktivitäten verzögern kann.

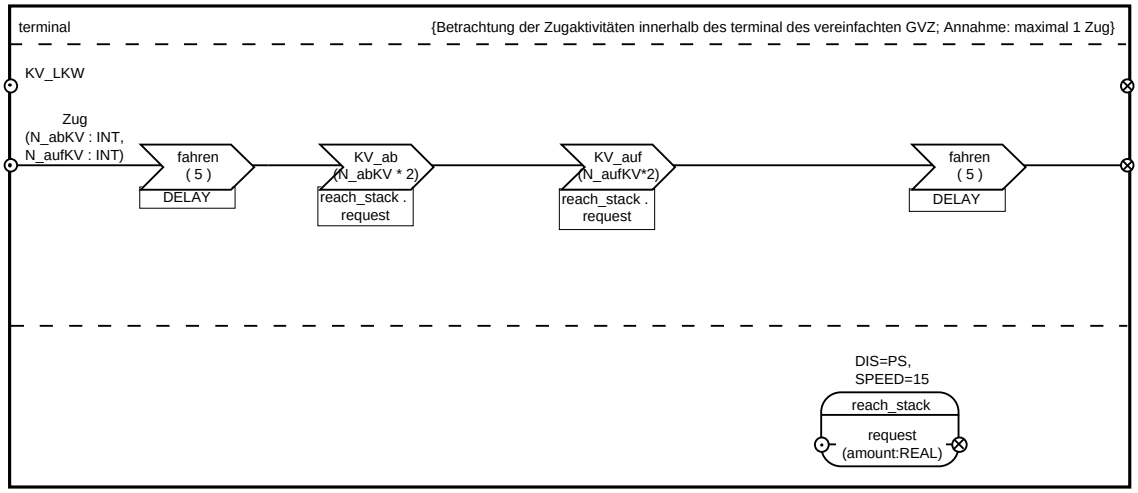


Abbildung 5-3 Dienst Zug des vereinfachten GVZ

Wir wollen nun zusätzlich auch die Huckepack-Verkehre betrachten. Die abstrakte Sicht des GVZ ändert sich, siehe Abbildung 5-4. Neben den Prozeßketten *Zug* und *LKW* sind jetzt auch die Prozeßketten *Huck_arrival* und *Huck_departure* beschrieben.

Auf den Namen des Modells im obersten Teil der Beschreibung folgt jetzt die Angabe dreier Attribute vom Typ *INT* (*Zug_arr*, *LKW_arr*, *Huck_arr*). Die Ankünfte von Zügen und LKWs werden durch Quellen modelliert, die in gewissen zeitlichen Abständen Prozesse vom Typ *Zug* und *LKW* generieren. Jeder Prozeß vom Typ *Zug* besitzt nun 4 Variablen vom Typ *INT*, die bei Erzeugung eines Prozesses aus unterschiedlichen Gleichverteilungen gesetzt werden. So wird beispielsweise der Variablen *N_abKV* gleichverteilt ein Wert zwischen 50 und 75 zugewiesen und *N_abHu* ein Wert zwischen 20 und 30. Die 4 Variablen spezifizieren die Ladung bzw. die Aufnahmekapazität eines Zuges.

In Abbildung 5-4 ist auch ein Dienst *Huck_ab* eingezeichnet, der nicht von Zügen oder LKWs genutzt werden kann. Dieser spezielle Dienst deutet an, daß innerhalb der Funktionseinheit *terminal* Prozesse vom Typ *Huck_ab* generiert werden (nämlich beim Abladen von Huckepack-Verkehr), welche bei Beendigung die Generierung von Prozessen an der Quelle hervorrufen. Diese generierten Prozesse führen anschließend Aktivitäten der Prozeßkette *Huck_departure* durch.

Die externe Sicht der Funktionseinheit *terminal* zeigt weiterhin, daß die Nutzung des Dienstes *Huck_an* den aufrufenden Prozeß innerhalb der Funktionseinheit beendet. Hiermit werden die ankommenden Huckepack-LKWs modelliert, welche im GVZ auf einen Zug verladen werden. Sinnvol-

lerweise erfolgt die Nutzung dieses Dienstes als letzte Aktivität einer Prozeßkette, daher ist der entsprechende Ausführungshinweis an der Senke der Prozeßkette *Huck_arrival* plaziert.

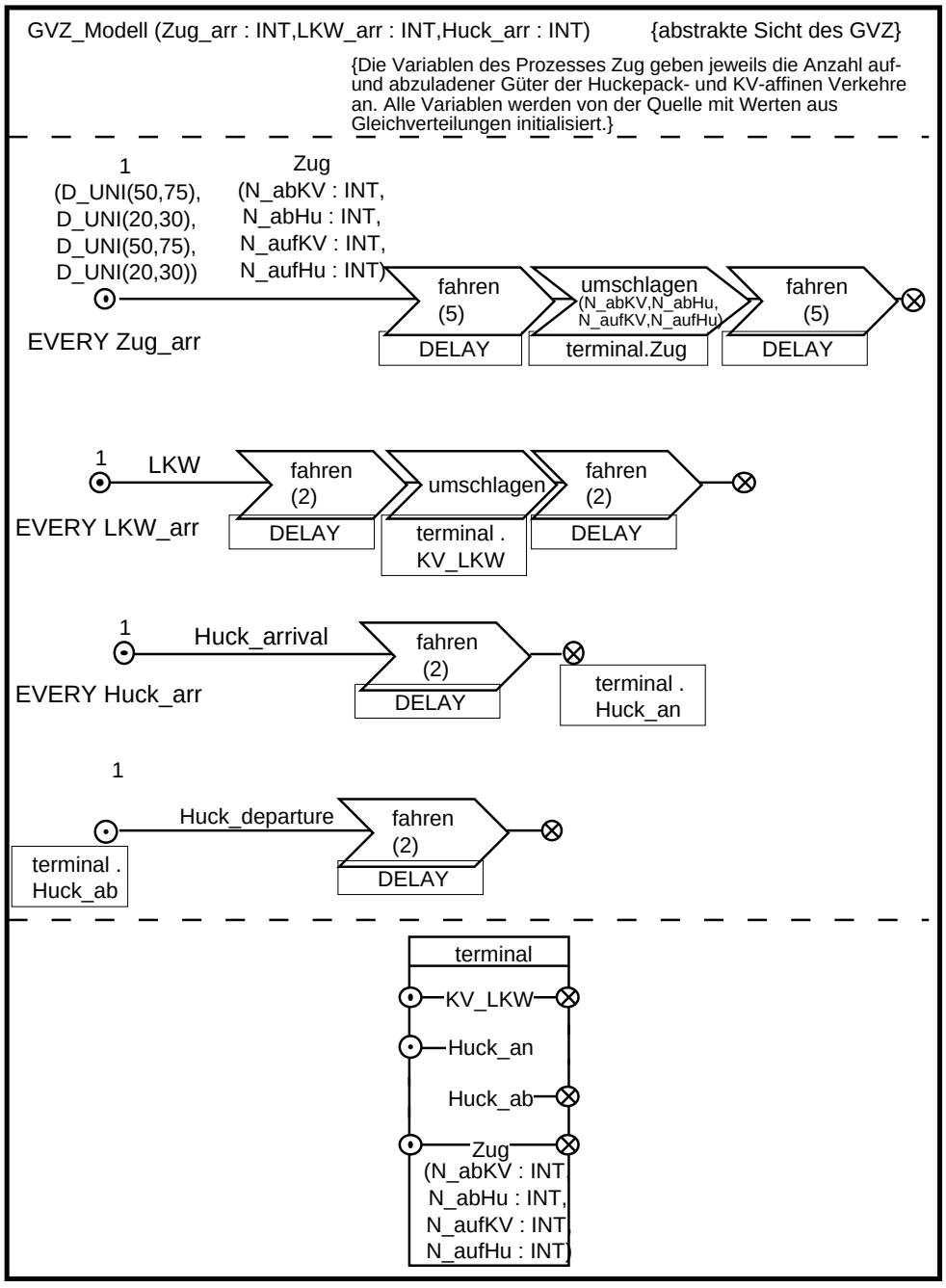


Abbildung 5-4: Abstrakte Sicht des GVZ

In der Funktionseinheit *terminal* werden die angesprochenen Dienste nun genauer spezifiziert (vgl. Abbildung 5-5). Der KV-affine Verkehr läßt sich bzgl. der LKWs genauso modellieren, da er nicht durch die Aktivitäten der Huckepack-LKWs beeinflußt wird.

Die Beschreibung der Huckepack-Verkehre und ihre Interaktion mit dem Zug sind dagegen komplexer. Zuerst müssen alle Huckepack-LKWs abgeladen werden und erst danach dürfen die entsprechenden Beladevorgänge beginnen. Da die Aktivitäten eines Huckepack-LKWs und eines Zuges durch verschiedene Prozeßketten dargestellt werden, müssen ihre Aktivitäten geeignet synchronisiert werden. Mittel zur Beschreibung von Synchronisationen sind Konnektoren und Counter (vgl. Kapitel 4).

In unserem Modell werden beide Beschreibungsmittel eingesetzt. Insgesamt werden 3 Counter verwendet (*Huck_ab, Huck_control, Huck_auf*):

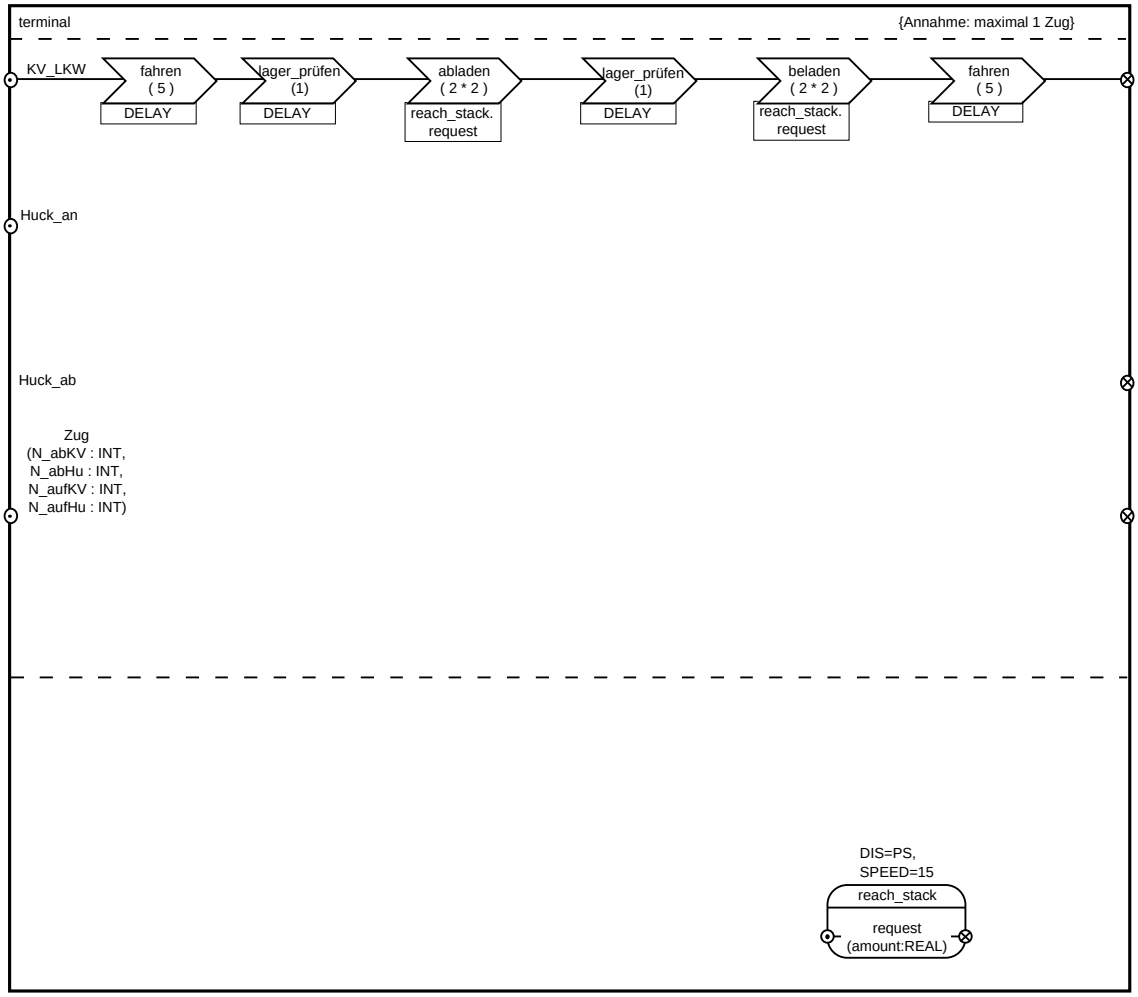


Abbildung 5-5: KV-affiner Verkehr innerhalb des GVZ

Der Counter *Huck_ab* zählt die Anzahl abzuladender LKWs. Jeder LKWs erhöht diesen Counter um 1. Der Zug versucht diesen Counter um die Anzahl abzuladender Huckepack-LKWs zu erniedrigen, was erst nach erfolgter Abladung aller LKWs ausgeführt werden kann. Hierdurch wird im Modell sichergestellt, daß ein Zug erst nach vollständiger Entladung der Huckepack-LKWs mit den entsprechenden Beladevorgängen beginnt.

Huck_control steuert den Start der Beladung von Huckepack-LKWs. Wenn der Zug seine Entladevorgänge beendet hat, setzt er diesen Counter auf einen Wert, der seiner Aufnahmekapazität bzgl. Huckepack-LKWs entspricht. Jeder aufzuladende Huckepack-LKW muß diesen Counter um 1 erniedrigen, bevor er zur Beladung auf den Zug zugelassen wird. Erst nach Zulassung darf ein LKW zum Beladen vorfahren. Kann ein LKW den Counter nicht erniedrigen, so muß er warten, was modellseitig durch ein „Steckenbleiben“ des Prozesses im Counter realisiert wird (vgl. Kapitel 4).

Huck_auf steuert das Ende der Beladung. Der letzte LKW erhöht diesen Counter um 1 und gibt somit dem Zug, der versucht diesen Counter zu erniedrigen, die Möglichkeit mit seinen Aktivitäten fortzufahren und aus dem GVZ herauszufahren.

Ankommender Huckepack-Verkehr nutzt den Dienst *Huck_an* des *terminal* (Abbildung 5-6). Dieser Huckepack-Verkehr soll auf einen Zug aufgeladen werden und mit diesem später das GVZ verlassen. Das GVZ, genauer die Funktionseinheit *terminal*, ist also eine Senke für diesen ankommenden Ver-

kehr. In Abbildung 5-4 ist dies bereits durch das fehlende Senkensymbol für den Dienst *Huck_an* angedeutet. Es wird damit in der externen Sicht bereits festgelegt, daß ein aufrufender Prozeß innerhalb der Funktionseinheit endet. In der internen Sicht (siehe Abbildung 5-6) wird dieses durch die fehlende Senke am rechten Rand dargestellt. Die einzelnen Aktivitäten die bei Nutzung des Dienstes *Huck_an* ausgeführt werden, stützen sich wiederum auf Standard-Funktionseinheiten ab. Neben dem Server *Kran* werden jetzt auch die o. g. Counter genutzt. Im Counter *Hu_control* wird mitgezählt, wieviel LKWs noch zwecks Beladung auf den Zug vorfahren dürfen. Wenn dem LKW die Beladung gestattet wird, wird nachfolgend die komponentenlokale Variable *n_huck* erhöht, die anzeigt, wieviel LKWs zur Beladung auf den Zug derzeit akzeptiert wurden. Zur Modellierung der „Zulassung“ von LKWs zur Beladung wurde ein Counter und nicht eine weitere komponentenlokale Variable verwendet, da hierdurch direkt das Warten des Huckepack-Verkehrs ausgedrückt werden kann: wird ein LKW nicht zum Beladen zugelassen, so drückt sich dies im Modell dadurch aus, daß der Diensteaufruf *Hu_control.change* vorerst nicht endet (Wert des Counters kann nicht erniedrigt werden, da bereits auf Null) und die Anforderung im Counter „wartet“ bis der Wert des Counters von einer anderen Aktivität entsprechend erhöht wird (vgl. Abbildung 5-7).

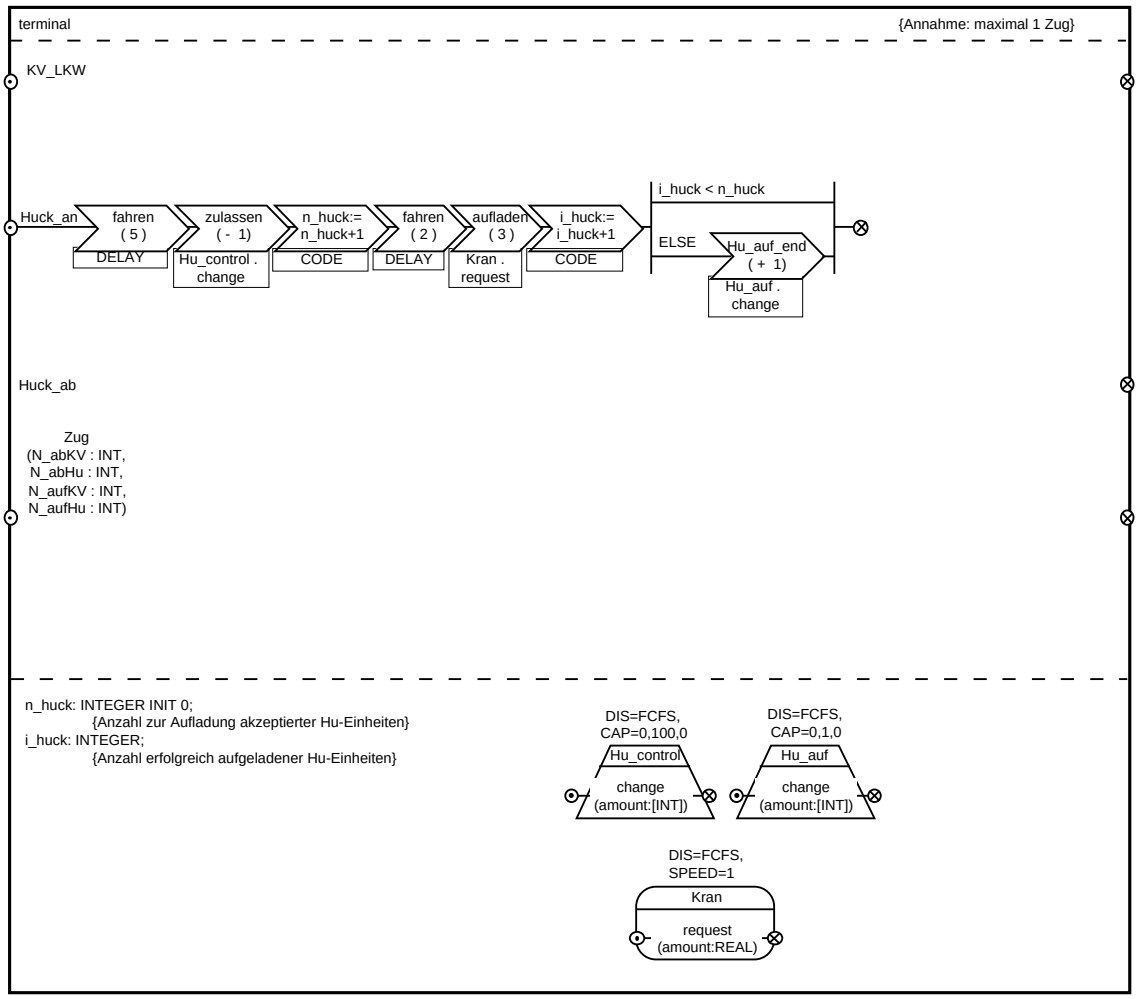


Abbildung 5-6: Ausschnitt aus terminal mit Dienst Huck_an

Wie wir in Kapitel 3.1 festgestellt hatten, war ein weiteres Problem die Modellierung der genauen Abfolge der Tätigkeiten beim Umschlagen des Huckepack-Verkehrs. Das Problem liegt darin, die korrekte Zeitdauer für das Umschlagen zu modellieren und zusätzlich die einzelnen abgeladenen LKWs, die direkt nach Abladen ihr „Eigenleben“ beginnen und unabhängig voneinander den GVZ-Bereich verlassen. Im Modell (siehe Abbildung 5-6) wird dies durch den Counter *Hu_auf* in Zusammenhang mit dem ODER-Konnektor beschrieben. Sind alle zur Beladung zugelassenen LKWs auch aufgeladen worden (Bedingung *i_huck* < *n_huck* ist nicht mehr erfüllt), so erhöht der letzte auf-

zuladende LKW den Wert des Counters *Hu_auf*. Dieser Counter dient dem Prozeß *Zug* (siehe Abbildung 5-7) als Information, daß der Beladevorgang abgeschlossen ist und er nachfolgend das GVZ verlassen kann, was durch das Beenden der Aktivitäten des Dienstes *Zug* modelliert ist. Die Anzahl der zur Aufladung akzeptierten LKWs (*n_huck*) und die Anzahl der erfolgreich aufgeladenen LKWs (*i_huck*) sind hier als komponentenlokale Variablen modelliert worden und nicht als Counter, da Wartesituationen bei entsprechenden Werteänderungen nicht auftreten sollen.

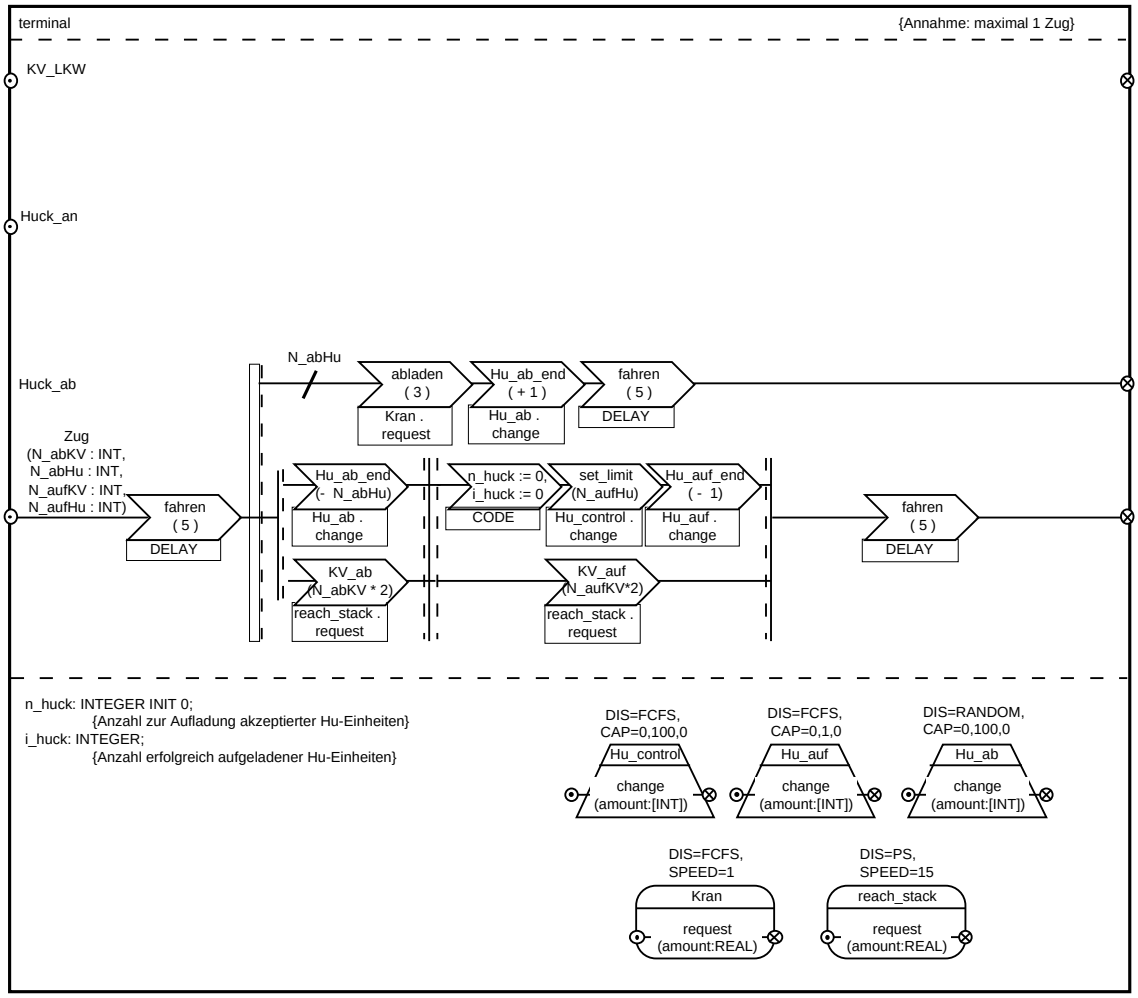


Abbildung 5-7: Dienst Zug inkl. Abladen des Huckepack-Verkehrs (Dienstergebnis Huck_ab)

Abbildung 5-7 zeigt in der internen Sicht des *terminal* die Aktivitäten der Dienste *Huck_ab* und *Zug*. Der Dienst *Zug* wird von Prozessen genutzt, die Variablen besitzen (*N_abKV,...,N_aufHu*), welche die Anzahl der jeweils auf- und abzuladenden Huckepack- und KV-Verkehre spezifizieren. Die Werte dieser prozeßlokalen Variablen werden innerhalb der Funktionseinheit benutzt, um Umschlagszeiten zu bestimmen, Zustände interner Funktionseinheiten zu setzen und Prozesse vom Typ *Huck_ab* zu instanziieren. Nach Einfahrt des Zuges (Aktivität *fahren*) werden mehrere Prozesse gleichzeitig gestartet, die nachfolgend nebenläufig (parallel) zueinander ablaufen können. Zum einen sind dies die bereits angesprochenen abzuladenden LKWs (*N_abHu* Prozesse vom Typ *Huck_ab*), die einzeln um den Dienst *request* der Funktionseinheit *Kran* konkurrieren und nach erfolgter Abladung durch Erhöhen des Wertes des Counters *Hu_ab* mithalten, wieviel LKWs aktuell abgeladen wurden: Der Wert des Counters *Hu_ab* wird vom Zug durch die Aktivität *Hu_ab_end* entsprechend der Anzahl abzuladender LKWs erniedrigt. Diese Anforderung an den Counter *Hu_ab* kann erst dann ausgeführt werden, wenn der Counter einen Wert größer oder gleich *N_abHu* besitzt, andernfalls würde das angegebene Minimum unterschritten werden. Die Aktivität *Hu_ab_end* ist somit erst dann beendet, wenn der gesamte Huckepack-Verkehr abgeladen wurde. Parallel zu diesen Aktivitäten werden die KV-affinen Güter abgeladen (*KV_ab*). Ist der gesamte Entladevorgang beendet, werden entsprechende Aktivitäten

der Beladung gestartet, wobei das Aufladen des Huckepack-Verkehrs und der KV-affinen Güter ebenfalls parallel erfolgt. In der Teilprozeßkette, welche den Huckepack-Verkehr betrifft, werden dabei zunächst die „Zählvariablen“ *n_huck* und *i_huck* initialisiert und die Anzahl aufzuladender Einheiten im Counter *Hu_control* abgelegt (*set_limit*), so daß eventuell noch nicht beendete Aktivitäten *zulassen* der Prozesse vom Typ *Huck_an* beendet werden können. Danach wird die Aktivität *Hu_auf_end* gestartet, die erst dann beendet werden kann, wenn der letzte LKW des Huckepack-Verkehrs aufgeladen ist, da der Counter *Hu_auf*, vgl. Abbildung 5-6) von diesem entsprechend erhöht wird. Nach Beendigung beider Aktivitäten (Huckepack-Verkehr und KV-affine Güter aufladen), setzt der Prozeß *Zug* mit der Aktivität *fahren* seinen Ablauf fort.

Alle oben gezeigten Ausschnitte der Funktionseinheit *terminal* sind in Abbildung 5-8 zusammengefaßt. Diese Darstellung modelliert das gesamte Verhalten des in Kapitel 3.1 beschriebenen GVZ in einer Form, daß aus dieser Beschreibung automatisch eine maschinengestützte Analyse möglich ist. Der Leser beachte, daß man bei dieser Analyse dann auf eventuelle Unzulänglichkeiten oder Fehler im Modell stoßen könnte. Beispielsweise setzt die Modellierung von Zählvariablen (*n_huck, i_huck*) voraus, daß sich maximal ein Zug innerhalb des GVZ befinden darf, was durch eine geeignete Abfolge von Aufrufen des Dienstes Zug sicherzustellen wäre, da andernfalls andere „Züge“ auf den gleichen Variablen „operieren“ würden. Des weiteren würde man bei der Modellanalyse feststellen, daß ein Zug das GVZ erst dann verläßt, wenn mindestens ein Huckepack-LKW aufgeladen wurde[5], was allerdings systemseitig durchaus sinnvoll sein könnte.

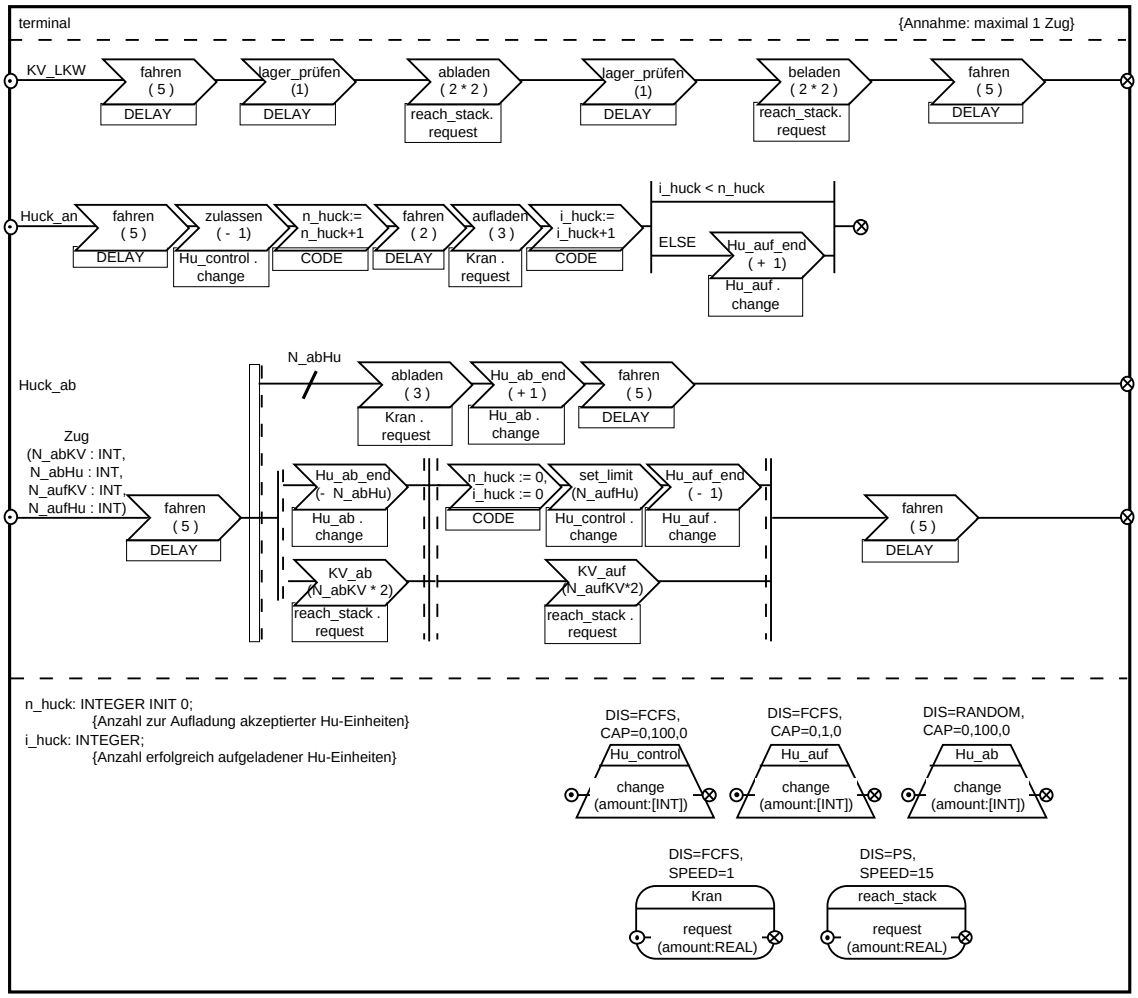


Abbildung 5-8: Funktionseinheit terminal (komplette Darstellung)

[5] Falls kein Huckepack-LKW auf Beladung warten würde, würde der Counter *Huck_auf* nicht erhöht werden und somit könnte der Prozeß der den Dienst *Zug* nutzt diesen nicht erniedrigen.

5.2 Redistributionsnetz

Hauptsächlich interessiert uns bei dem Redistributionsnetz aus Kapitel 3.2 der Fluß der Leistungsobjekte durch das Netz und die sich hierdurch ergebenden mittleren Bestände an den jeweiligen Orten. Vereinfacht betrachtet, läßt sich dieser Fluß durch die potentiellen Routen der Leistungsobjekte durch das Netz beschreiben, wobei für die einzelnen Routen in erster Betrachtung Häufigkeiten oder Wahrscheinlichkeiten angenommen werden können.[6] Ein entsprechendes Modell zeigt Abbildung 5-9: . Da (in diesem speziellen System!) alle Zyklen des Systems den Ort 3 enthalten, ist es naheliegend diesen Ort als Start- und Endpunkt einer sequentiellen Darstellung zu wählen. Nachdem ein Behälter eine gewisse Zeit im Orte 3 verweilt hat und z. B. produzierte Güter aufnimmt (*prod_3*)[7], wird probabilistisch (Wahrscheinlichkeit h_{31}) der nächste Besuchsort ausgewählt. Analog wird ein Behälter am Orte 1 mit Wahrscheinlichkeit h_{15} nach 5 weitertransportiert oder gelangt mit Wahrscheinlichkeit $1\text{-}h_{15}$ nach 4 (siehe *ELSE*-Übergang). Nach Festlegung des Quellenverhaltens und Angabe entsprechender Verweilzeiten an den Orten und Transportzeiten zwischen diesen, läßt sich das Modell bereits analysieren. Der wesentliche Vorteil dieser vereinfachten Modellbeschreibung ist, daß sehr effiziente Algorithmen aus der Warteschlangen-Theorie eingesetzt werden können, die z. B. eine rechnergestützte Ermittlung der mittleren Bestände innerhalb Sekundenbruchteilen erlaubt. Hierdurch ist es für den Konstrukteur eines solchen Netzes möglich, erste Dimensionierungsgrößen beispielsweise für unterschiedliche Quellparameter interaktiv zu ermitteln.

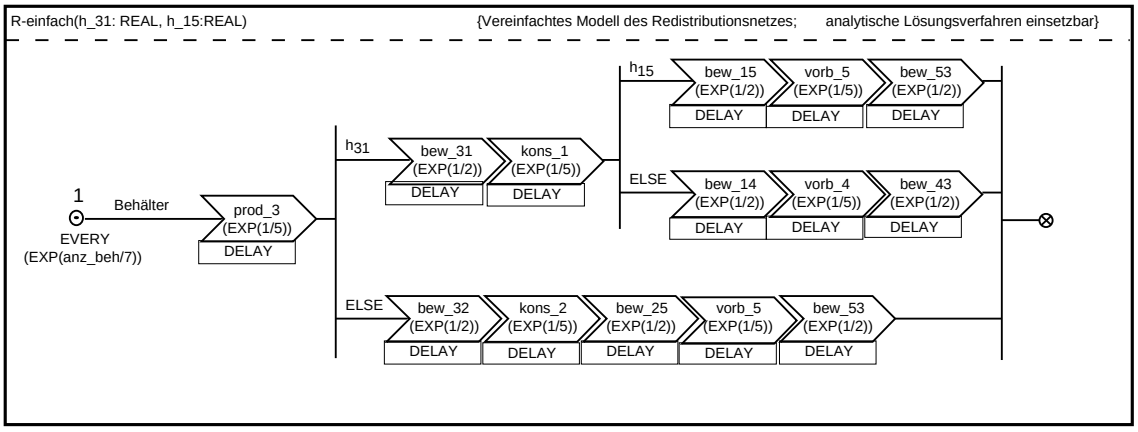


Abbildung 5-9: Effizient lösbares Redistributionsmodell

Das vereinfachte Modell in Abbildung 5-9 berücksichtigt noch nicht die vorgegebenen Produktionsdaten, welche Grundlage zur Bestimmung der Anzahl zu transportierender Behälter sind. Anhand dieser Daten und der Information über die Anzahl bereits transportierter Behälter muß entschieden werden, wieviel leere Behälter zusätzlich zu transportieren sind. Hierbei sind allerdings einige Freiheitsgrade gegeben, da durch Lagerung einer kleineren oder größeren Anzahl von Leerbehältern die Transportmengen über mehrere Wochen durchaus unterschiedlich verteilt werden können.

In Abbildung 5-10 wird der in Abbildung 5-9 probabilistisch beschriebene Fluß der Behälter nun durch eine Lenkungsebene gesteuert, die mit den gleichen Modellierungselementen beschrieben wird, wie die zu steuernde Materialflußebene. In der Lenkungsebene werden komponentenlokale Variablen (hier n_{31}, n_{15}, z_{31}, z_{15}) gesetzt, die indirekt über die Abfrage an den Konnektoren die Wahl der nächst aufzusuchenden Orte bestimmt. Das Setzen dieser Variablen erfolgt immer für die nächste Periode. Nachfolgend werden die jeweiligen Produktionsdaten ermittelt (a_{10}, ..., a_{52})[8] und danach entschieden wieviel Behälter in der nächsten Periode zu transportieren sind. Die in der Lenkungsebene angegebe-

[6] Die jeweiligen Werte können beispielsweise realen Messungen entnommen werden. Falls diese nicht verfügbar sind, können durch Parametervariation der Wahrscheinlichkeitswerte die Abhängigkeiten zwischen mittleren Beständen und Anzahl zu transportierender Behältern ermittelt werden.

[7] *prod_3* deutet die Produktion im Orte 3 an, *kons_i* konsumieren im Orte i, *vorb_j* vorbereiten im Orte j und *bew_ij* das Bewegen eines Behälters vom Orte i zum Orte j.

[8] Da die maximale Zeit um einen Behälter vom Ort 3 zu einem anderen Ort zu transportieren in diesem Beispiel 2 beträgt, reicht eine „Vorausschau“ um zwei Perioden. Dabei wird angenommen, daß i.allg. genügend Behälter im Orte 3 lagern.

nen Tätigkeiten {*ermittle...}* und *{entscheide...}* müßten noch entsprechend präzisiert werden, um ein automatisch analysierbares Modell zu erhalten. Zum Abschluß werden in der Lenkungsebene noch die Zustände zweier Counter (*exit_3* und *exit_1*) mit der Anzahl der Behälter gesetzt, die die Orte 3 und 1 in der jeweiligen Periode verlassen dürfen. Die Beschreibung der Behälteraktivitäten wird durch die oben genannten Modellemente gesteuert. Neben der bereits erwähnten Abfrage der komponentenlokalen Variablen (n_{31}, n_{15}, z_{31}, z_{15}) zur „Flußsteuerung“ werden Behälter, welche die Orte 3 bzw. 1 verlassen wollen, gegebenenfalls durch die Counter *exit_3* bzw. *exit_1* zurückgehalten, sofern die Anzahl zu transportierender Behälter bereits erreicht worden ist.

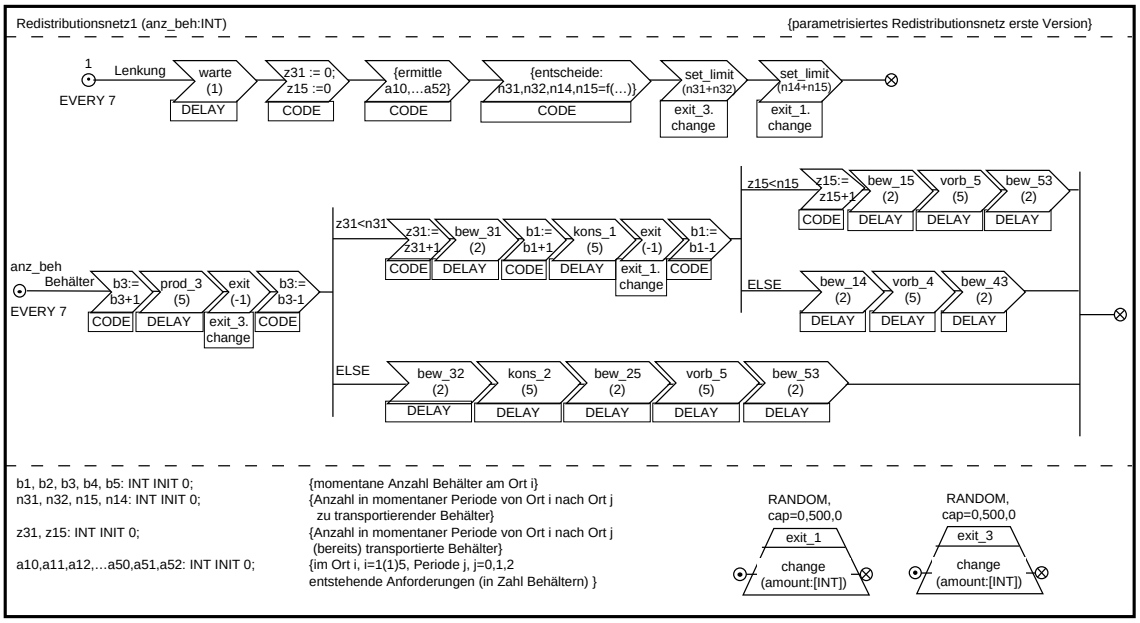


Abbildung 5-10: Detailliertes Modell des Redistributionsnetzes

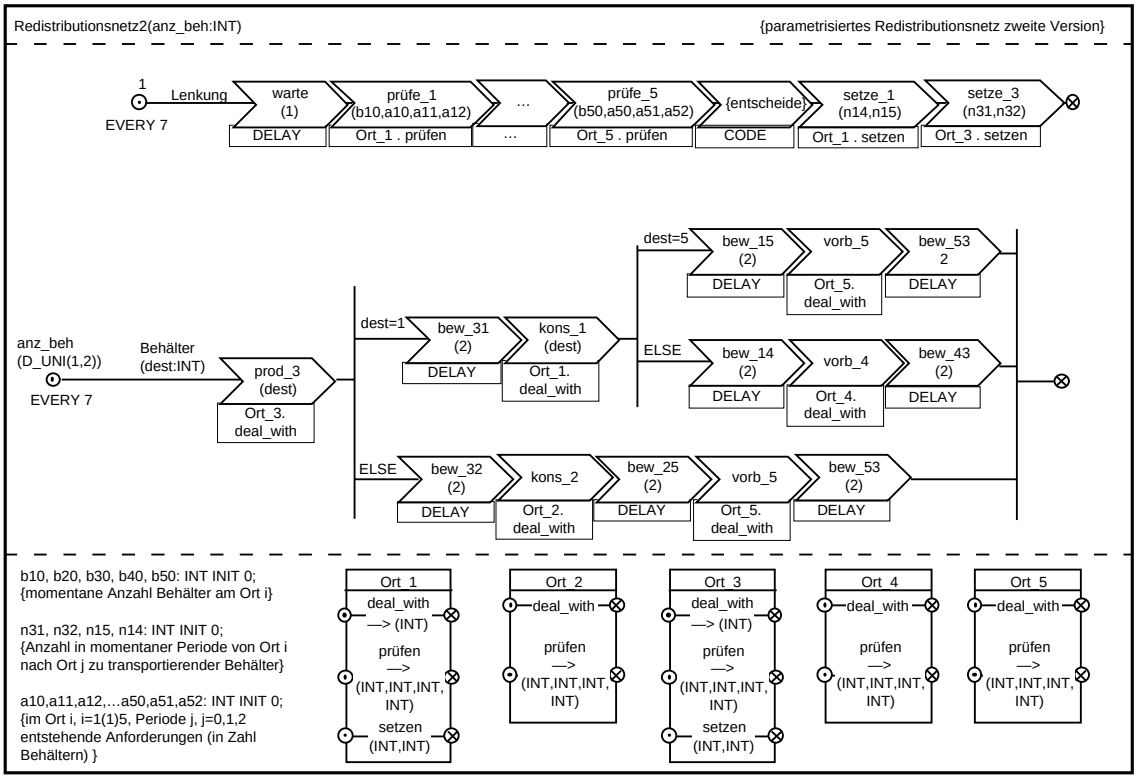


Abbildung 5-11: Verfeinertes Modell des Redistributionsnetzes

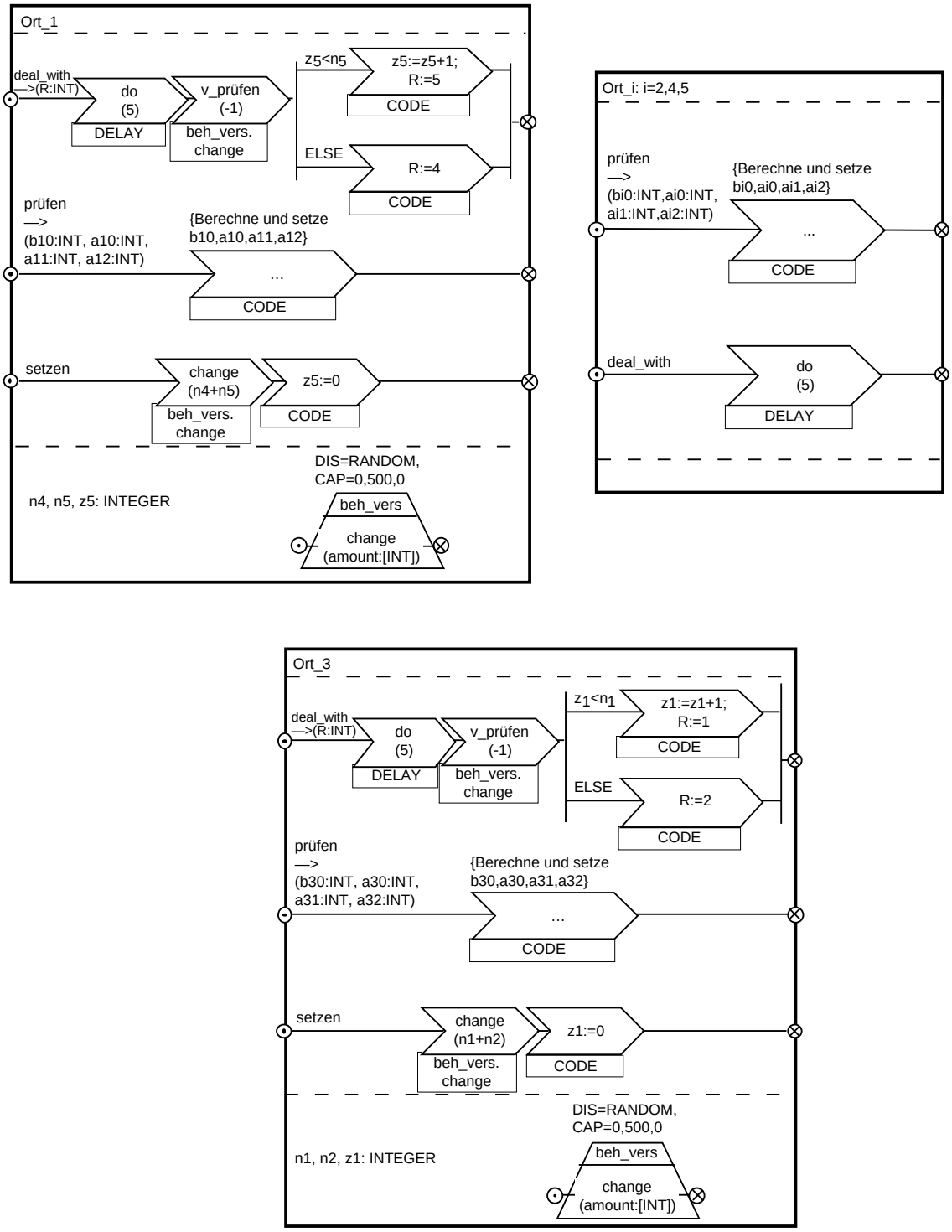


Abbildung 5-12: Verfeinerung innerer Funktionseinheiten des Redistributionsnetz-Modell aus Abbildung 5-11

In Abbildung 5-11 und Abbildung 5-12 wird das Modell aus Abbildung 5-10 weiter verfeinert, indem die an den jeweiligen Orten stattfindenden Aktivitäten genauer beschrieben werden. Wie in Abbildung 5-11 zu sehen, ändert sich die Struktur des Modells auf dieser Ebene nur unwesentlich, wodurch ein wesentlicher Vorteil einer geeigneten hierarchischen Modellierungsweise deutlich wird. In diesem (etwas realistischeren) Modell wird davon ausgegangen, daß die Information über die benötigten Behältermengen und derzeitigen Behälterbestände in den jeweiligen Orten lokal vorliegen und der Lenkungsebene über den Dienst *prüfen* zur Verfügung gestellt werden. Mittels des Dienstes *setzen*

werden innerhalb der Funktionseinheiten *Ort_3* und *Ort_1* die Counter *beh_vers*[9] gesetzt. Diese Counter übernehmen jeweils die Funktionen der Counter *exit_3* und *exit_1* (vgl. Abbildung 5-10). Der zuvor nur durch eine Verzögerung dargestellte Vorgang des Verweilens des Behälters an einem Orte (z. B. *prod_3*) wird jetzt durch den Dienst *deal_with* modelliert, wobei zusätzlich die Entscheidung ob ein Behälter weiterzutransportieren ist (*v_prüfen*) und wohin, ebenfalls lokal im jeweiligen Ort vorgenommen wird.

Daß Dienstaufrufe (siehe Abbildung 5-12) Ergebnisse ermitteln, ist beispielsweise an der Signatur ersichtlich, z. B. *deal_with -> (INT)*. In der internen Sicht wird das Ergebnis, der jeweils angegebenen Ergebnisvariablen zugewiesen (*R* in Abbildung 5-12). Man beachte, daß die Ergebnisse mehrdimensional sein dürfen, wie z. B. bei Dienst prüfen, der den Bestand und die in den nächsten Perioden benötigte Anzahl an Behältern zurückliefert. Wie Abbildung 5-12 zeigt sollen die Funktionseinheiten der Orte 2, 4 und 5 gleich sein. Dies deutet eine mögliche Erweiterung des B1-Paradigmas an, nämlich die Einführung von Typen von Funktionseinheiten.

5.3 Beschaffung

In diesem Kapitel soll das Beispiel aus Kapitel 3.3 anhand des im Kapitel 4 definierten B1-Formalismus modelliert werden.

Der *Hersteller* läßt sich als Modell betrachten, also als FE die keine Dienste anbietet. Das Modell beinhaltet eine Prozeßkette (*herstellen*). Die Abbildung 5-13 zeigt die Notation des Modells.

Einmal täglich (alle 1440 Minuten) wird ein Prozeß vom Typ *herstellen* generiert. Zu jedem dieser Prozesse gehören die Prozeßvariablen *Bedarf* (für den am siebtfolgenden Tag ermittelten Bedarf), *Bestellmenge* (für die Menge, der für den siebtfolgenden Tag bestellten A-Teile), *Gutteile* (für den Anteil an A-Teilen, die ordnungsgemäß geliefert wurden) und *Liefermenge* (für die tatsächlich angelieferte A-Teil-Menge).

Um festhalten zu können, wie hoch der Gesamtbedarf aller laufenden *herstellen*-Prozesse ist, die sich zwischen Bedarfsermittlung und Fertigteilproduktion befinden, ist eine Variable des Modells *GBedarf* (Gesamtbedarf) definiert. Ebenso gibt es eine Variable des Modells für die Gesamtbestellmenge (*GBestellmenge*), mit der festgehalten wird, wieviele A-Teile aktuell von allen laufenden *herstellen*-Prozessen beim Lieferanten bestellt, aber noch nicht geliefert worden sind.

Außerdem ist ein Lager für die A-Teile deklariert, das minimal 0 und maximal 1600 Teile lagert und initial mit 0 Teilen startet. Die Variable *LBestand* speichert den aktuellen Lagerbestand.

herstellen nutzt den Dienst *produzieren* der Funktionseinheit *Lieferant*. Standard-FEs vom FE-Typ SERVER sind die *Annahme*, deren Kapazität eine bestimmte Anzahl an Personal (*Auslader*) umfaßt, die *Fertigung*, die eine Anzahl an *Maschinen* anbietet sowie die *Fertigungsprüfung*, die eine bestimmte Anzahl an *Prüfer* umfaßt. *Auslader*, *Maschinen* und *Prüfer* sind Attributparameter der FE *Hersteller*. Alle drei Standard-FEs arbeiten mit der Reihenfolgedisziplin PS (Processor Sharing – die Ressourcen werden also jeweils auf die Aufträge verteilt).

Jeder *herstellen*-Prozeß startet mit einer Aktivität, die den *Bedarf* für den siebtfolgenden Tag festlegt. Dieser wird hier durch eine Gleichverteilung zwischen den Werten 750 und 850 Stück beschrieben. Dieser *Bedarf* wird in einer zweiten Aktivität auf den Gesamtbedarf (*GBedarf*) addiert.

Anschließend findet eine Aufspaltung in zwei parallele Zweige statt. Im oberen Zweig wird die Bestellung, Lieferung und Prüfung der A-Teile behandelt, im unteren die Produktion der Fertigteile.

[9] Behälter verschicken

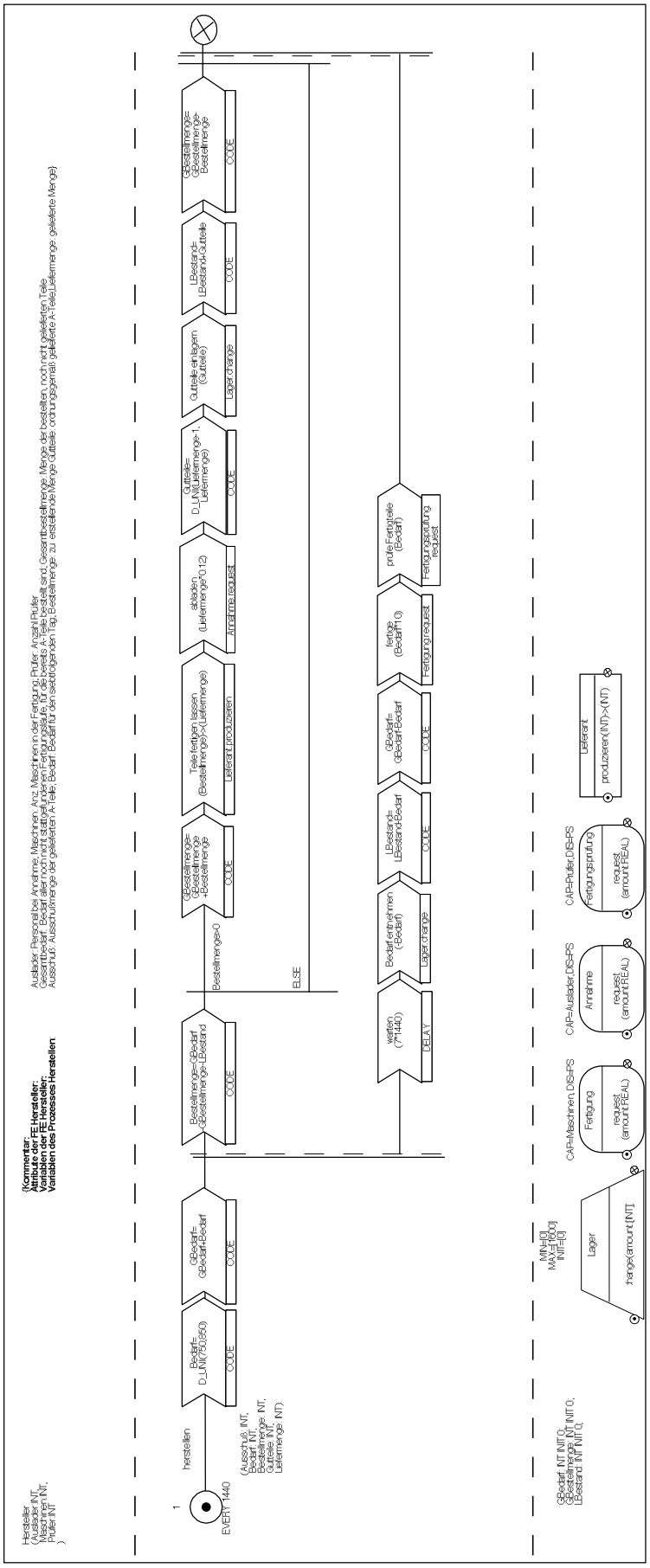


Abbildung 5-13: Modellbeschreibung der Funktionseinheit Hersteller

Als erstes wird im oberen Zweig die *Bestellmenge* ermittelt. Dazu wird vom aktuellen Gesamtbedarf (*GBedarf*) die Menge aller bereits bestellten, aber noch nicht gelieferten A-Teile sowie der aktuelle Lagerbestand (*LBestand*) abgezogen. Ist das Ergebnis kleiner oder gleich 0, so wird nichts nachbestellt. Ansonsten wird die *Bestellmenge* auf die Gesamtbestellmenge (*GBestellmenge*) addiert und der *Lieferant* durch Nutzen seiner Prozeßkette *produzieren* damit beauftragt, die benötigte Anzahl A-Teile zu liefern (die *Bestellmenge* wird dazu übergeben und eine *Liefermenge* wird als Rückgabewert erwartet). Geliefert wird jedoch nicht notwendigerweise die bestellte Menge (*Bestellmenge*), sondern eine evtl. davon leicht abweichende *Liefermenge*. Die gelieferten Teile werden durch die *Annahme* abgeladen (Dauer 0,12 Minuten je Teil). Die anschließende Teileprüfung in der FE *Fertigungsprüfung* ermittelt die Anzahl der ordnungsgemäßen A-Teile (*Gutteile*). Im Modell wird angenommen, daß sich in den Lieferungen zu 50% 1 fehlerhaftes und zu 50% kein fehlerhaftes Teil befindet. Die ordnungsgemäßen A-Teile werden eingelagert und die Gesamtbestellmenge (*GBestellmenge*) um die für diese Lieferung bestellte Menge (*Bestellmenge*) vermindert. (Die Differenzmenge zwischen *Bestellmenge* und *Gutteile* wird bei der Bestellmengenermittlung des *herstellen*-Prozesses aufgefangen, der als nächstes gestartet wird.)

Im unteren Zweig wird zunächst 7 Tage = 7*1440min gewartet, da die in einem *herstellen*-Prozeß betrachtete A-Teile-Menge für die Produktion des siebtfolgenden Tages gedacht ist. Nach diesen 7 Tagen wird die zur Tagesproduktion benötigte Menge an A-Teilen (*Bedarf*) dem Lager entnommen. Sollte zum Beginn dieser Produktion die Lieferung der benötigten Teile noch nicht oder nicht vollständig erfolgt sein, so wird versucht werden, diese dennoch dem Lager zu entnehmen (für den Fall, daß sich trotz der Lieferprobleme aktuell genügend A-Teile im Lager befinden). Solange sich nicht die benötigte Menge entnehmen läßt, wird in dieser Aktivität gewartet, bis dies möglich ist. (Durch diesen Mechanismus findet über den Lagerbestand eine Synchronisation zwischen dem oberen und dem unteren Zweig dieses Prozeßtypen statt.) Ein hier wartender Prozeß zieht vom Gesamtbedarf (*GBedarf*) seinen eigenen *Bedarf* nicht ab. Ist der Rückstand im Lager durch eine unvollständige Lieferung begründet, so ist jedoch die Gesamtbestellmenge (*GBestellmenge*) verringert. In einem solchen Fall findet ein anderer, später gestarteter *herstellen*-Prozeß, der während der Wartezeit eines Prozesses auf ausreichende Lagerfülle seine *Bestellmenge* ermittelt, einen erhöhten Gesamtbedarf und eine niedrige Gesamtbestellmenge vor und erhöht daher seine *Bestellmenge* dementsprechend.

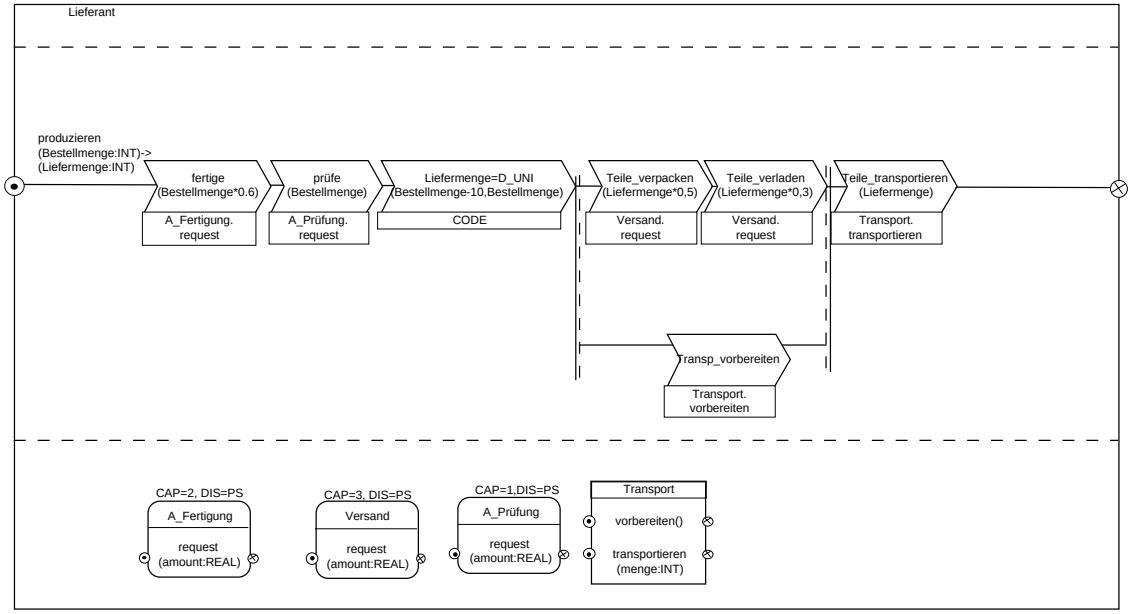


Abbildung 5-14: Modellbeschreibung der Funktionseinheit Lieferant

Nach der erfolgreichen Entnahme der benötigten A-Teile aus dem Lager wird der Gesamtbedarf um den Tagesbedarf vermindert und anschließend die *Fertigung* mit der Produktion der Fertigteile beauftragt. Dabei sind für jedes zu verarbeitende A-Teil 10 Minuten Bearbeitungszeit angesetzt. Anschließend wird parallel einerseits die *Fertigungsprüfung* mit der Prüfung der Fertigteile beauftragt (Dauer:

1 Minute je verarbeitetem A-Teil) und nach der Prüfung die Fertigteile gelagert sowie andererseits das Leergut für die A-Teile zunächst gelagert und dann zurückgegeben.

Nach Abschluß des oberen und des unteren Zweiges ist der *herstellen*-Prozeß beendet.

Die beim Ablauf des *herstellen*-Prozesses genutzte FE *Lieferant* (vgl. Abbildung 5-14) bietet einen Dienst (*produzieren*) an. Dieser Dienst übernimmt einen Wert für die *Bestellmenge* vom aufrufenden Prozeß und definiert für jeden *produzieren*-Prozeß einen Ergebnisparameter *Liefermenge*, der die tatsächlich nach der Produktion zur Verfügung stehende und ausgelieferte A-Teile-Menge erfaßt. *Liefermenge* ist ein Ergebnisparameter des *produzieren*-Prozesses vom Typ INT. Von der Prozeßkette jedes *produzieren*-Prozesses aus können zwei verschiedene Dienste (*vorbereiten* und *transportieren)* der Funktionseinheit *Transport* angestoßen werden. Als Standard-FEs stehen die *A-Fertigung*, die *A-Prüfung* und der *Versand* bereit, die jeweils über eigenes Personal verfügen und alle Anfragen jeweils nach PS-Strategie abarbeiten.

Jeder *produzieren*-Prozeß beginnt mit der Fertigung der bestellten Menge. Mit dieser Aufgabe wird die *A-Fertigung* beauftragt. Dabei sind je Teil 0,6 Minuten angesetzt. Die gefertigten Teile werden durch die *A-Prüfung* geprüft (Dauer: 1 Minute je Teil). Nach der Prüfung ergibt sich die *Liefermenge*, die in diesem Beispiel maximal um 10 A-Teile niedriger sein kann als die *Bestellmenge* (alle möglichen Werte sind dabei gleichverteilt). Während die Funktionseinheit *Transport* beauftragt wird, den Transport vorzubereiten, werden durch den *Versand* die ordnungsgemäßen A-Teile verpackt (0,5 Minuten je Teil) sowie anschließend auf ein Transportmittel verladen (0,3 Minuten je Teil).

Nach Abschluß der Transportplanung und des Verpackens und Verladens wird die FE *Transport* damit beauftragt, den Transport der *Liefermenge* (diese wird an den aufrufenden Prozeß übergeben) durchzuführen. Damit ist der *produzieren*-Prozeß beendet.

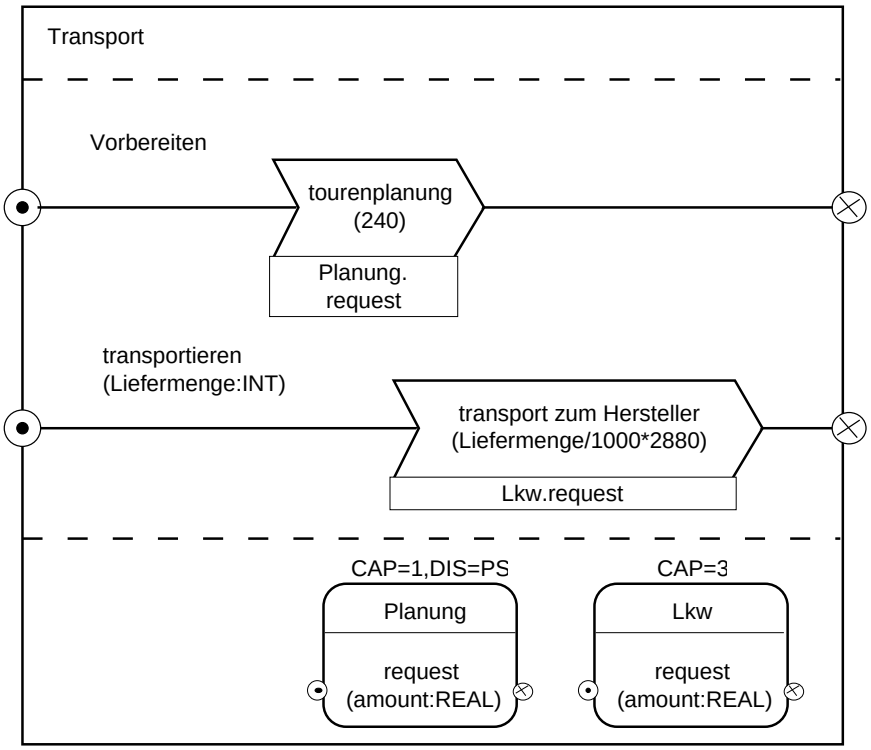


Abbildung 5-15: Modellbeschreibung der Funktionseinheit Transport

Die verwendete Funktionseinheit *Transport* (vgl. Abbildung 5-15) bietet zwei Dienste (v*orbereiten* und *transportieren*) an. Des weiteren sind die Standard-FEs *Planung* und *Lkw* definiert. Die *Planung* arbeitet mit der Kapazität 1 (dies entspricht der Vorstellung, daß es nur einen Planungsverantwortlichen gibt) sowie nach der Reihenfolgedisziplin PS. In der Funktionseinheit *Lkw* steht eine gewisse

Anzahl an LKWs zur Verfügung, mit denen die ihnen übertragenen Aufgaben durchgeführt werden. Als Reihenfolgedisziplin ist hier FCFS (First Come First Served – also in der Ankunftsreihenfolge) gewählt. Es wird angenommen, daß zu jedem LKW dauerhaft ein Fahrer gehört, so daß diese nicht gesondert im Modell aufgeführt werden müssen.

Ein *vorbereiten*-Prozeß beauftragt die *Planung* mit der Tourenplanung, für die 4 Stunden (240 Minuten) angesetzt sind. Anschließend ist der Prozeß beendet.

Ein *transportieren*-Prozeß beauftragt die Funktionseinheit *Lkw* mit dem Transport der durch den anstoßenden Prozeß übergebenden Liefermenge zum Hersteller. Als Dauer sind dafür je 2 Tage je 1000 angefangenen Einheiten angesetzt. Nach dieser Aktivität ist der Prozeß beendet.

Anzumerken ist, daß eine Einbeziehung des Leergutes der A-Teile in das Modell dadurch möglich ist, daß ihr Bestand beim Lieferanten über ein Counter-Element gezählt wird. Dieser Counter würde dann bei der Verpackung der A-Teile erniedrigt und nach der Rücklieferung des Leergutes entsprechend erhöht.

5.4 Distribution

Dieser Abschnitt greift die beiden Beispielszenarien aus Abschnitt 3.4 wieder auf und entwirft zugehörige Modelle im Rahmen des B1-Formalismus. Dem Charakter der Szenarien entsprechend, wird in 5.4.1 ein weithin vollständiges Modell angestrebt, während in 5.4.2 der Referenzmodell-Charakter erhalten bleibt, so daß vollständige Modelle erst nach Ergänzung um konkrete Parameterwerte etc. entstehen würden.

5.4.1 Anwendungsbeispiel der Distribution

Das Anwendungsbeispiel aus Kapitel 3.4.1 soll nun mit dem in Kapitel 4 vorgestellten B1-Modellformalismus modelliert werden (vgl. Abbildung 5-16).

Dazu werden zunächst die drei Hauptprozesse in den B1-Modellformalismus übertragen. Die Aktivitäten dieser Prozesse werden parametriert und an die Dienste von Funktionseinheiten zugewiesen. Auf die Funktionseinheiten wird später ausführlich eingegangen.

Die erste Prozeßkette beschreibt das Auslagern, den Transport zum Kunden und das Entladen von Lieferungen. Dazu wird in bestimmten Zeitabständen (*Auft_arr*) ein neuer Auftrag generiert. Die Aufträge besitzen genau zwei Parameter *mengeA* und *mengeB*. Diese beiden Parameter geben die Bestellmengen für den jeweiligen Produkttyp an. Bei der Erzeugung von Aufträgen nehmen diese Parameter Zufallszahlen zwischen 50 und 100 an. Sie werden bei der Aktivität *Auslagern* an den Dienst *ausl* der Funktionseinheit *Lager* weitergegeben. Beim Transport ist kein Parameter erforderlich, da für jede Strecke ein Dienst angeboten wird. Schließlich wird die Ladung beim Kunden entladen. Zur Vereinfachung wird angenommen, daß die Ladung pro Auftrag die Kapazität eines LKWs nicht überschreitet. Deswegen reicht eine „1“ (eine Ladung) als Parameter für die Aktivität *Entladen* aus.

Bei den letzten beiden Prozeßketten handelt es sich um die Produktion, den Transport und die Einlagerung von Objekten in das Zwischenlager. Dabei werden die Quellen, die neue Objekte erzeugen, von Diensten des Zwischenlagers gesteuert. Nach jedem Auslagerungsauftrag wird überprüft, ob eine Nachbestellung erforderlich ist. Wenn dies der Fall ist, wird die Quelle des entsprechenden Objekttyps getriggert. Nachdem die Objekte erzeugt sind, werden sie in dem jeweiligen Werk auf LKWs geladen. Dazu stellen die Funktionseinheiten der Werke den Dienst *bel* zur Verfügung. Den Transport übernimmt wieder die Funktionseinheit *Sped*. Schließlich werden die Objekte in das Zwischenlager eingelagert.

Unser Modell besteht also aus drei Hauptprozessen, deren Aktivitäten auf die Dienste von fünf Funktionseinheiten zugreifen. Diese fünf Funktionseinheiten und ihre Dienste werden im folgenden genauer beschrieben.

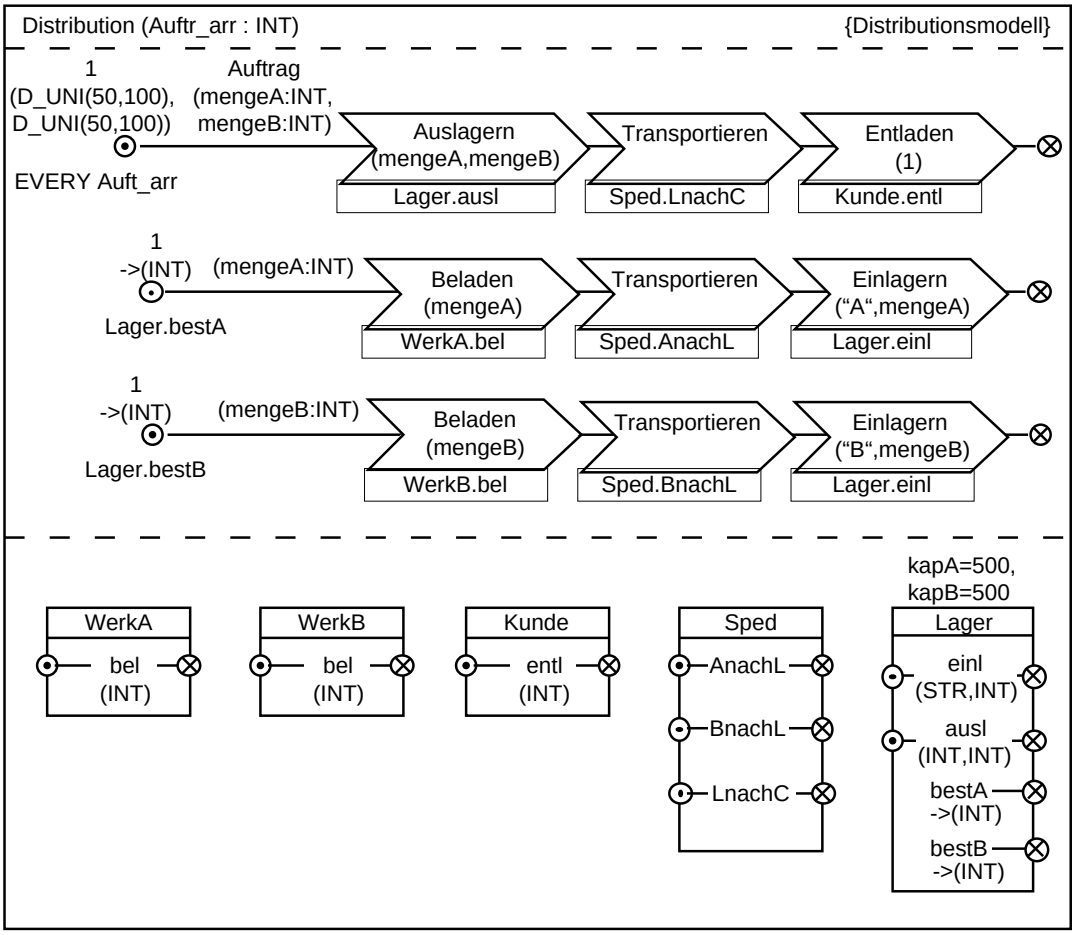


Abbildung 5-16: Distributionsmodell

WerkA und WerkB

Diese beiden Funktionseinheiten stellen nur den Dienst *bel* (beladen) für das jeweilige Werk zur Verfügung. Die Menge der zu beladenden Produkte wird als Parameter übergeben. Die interne Sicht dieser Funktionseinheiten ist in folgender Abbildung zu sehen:

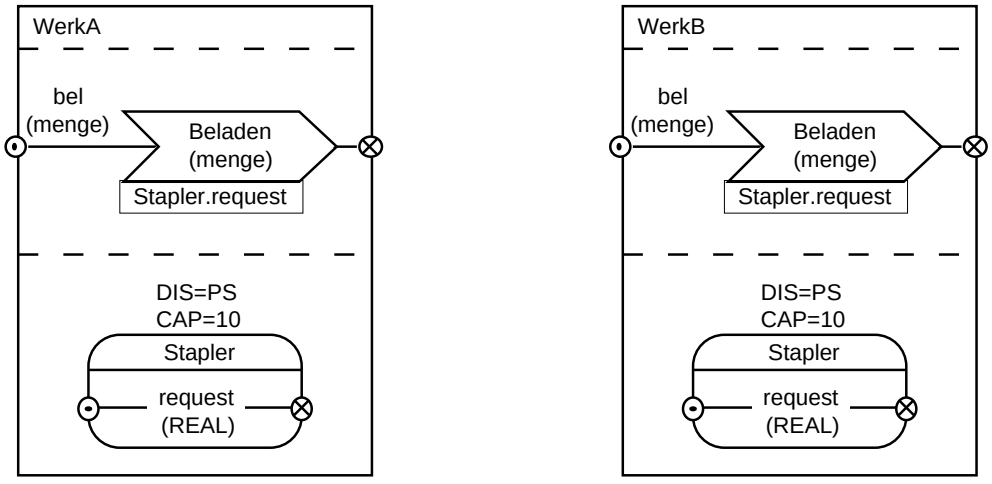


Abbildung 5-17: Funktionseinheiten WerkA und WerkB

Diese Funktionseinheiten besitzen eine Standardfunktionseinheit vom Typ Server. Diese Funktionseinheit stellt die Menge der zur Verfügung stehenden Gabelstapler dar. In unserem Beispiel besitzt

jedes Werk genau zehn Gabelstapler (CAP = 10). Die Beladungsdienste bestehen jeweils nur aus einer Aktivität, die auf die Gabelstapler zugreifen.

Kunde

Die Funktionseinheit Kunde ist analog zu den Werken aufgebaut. Sie bietet jedoch den Dienst *entl* (entladen) an. Hier stehen ebenfalls zehn Gabelstapler zur Verfügung, die für das Entladen von LKWs zuständig sind. Diese werden wieder mit einer Standardfunktionseinheit vom Typ Server modelliert.

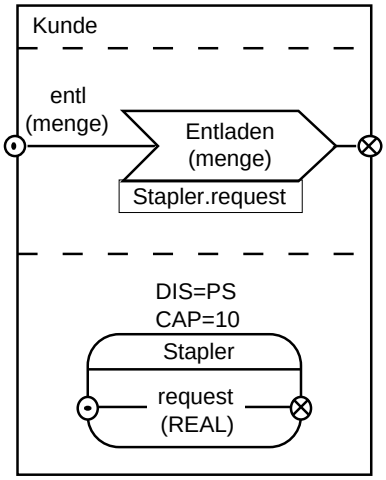


Abbildung 5-18: Funktionseinheit Kunde

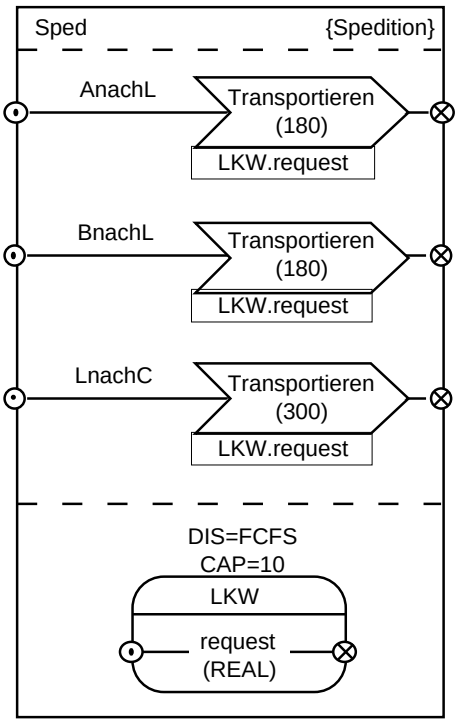


Abbildung 5-19: Funktionseinheit Spedition

Spedition

Diese Funktionseinheit stellt eine Speditionsfirma dar, die sowohl den Transport zwischen den Werken und dem Zwischenlager, als auch den Transport vom Zwischenlager zum Kunden durchführt. Zu diesem Zweck werden drei Dienste angeboten.

Es wird angenommen, daß die LKWs der Spedition bei allen drei Transporten eingesetzt werden können. Deswegen bestehen diese Dienste jeweils nur aus einer Aktivität, die alle auf die selbe Funktionseinheit zugreifen. Es handelt sich hierbei wieder um eine Standardfunktionseinheit vom Typ Server.

Lager

Die Funktionseinheit *Lager* bietet verschiedene Dienste zum Einlagern, Auslagern und Nachbestellen an. Der Dienst *einl* (einlagern) wird mit zwei Parametern *typ* und *menge* aufgerufen. Diese Parameter geben den einzulagernden Objekttyp und die Menge an. Die interne Sicht des Lagers ist in folgender Abbildung zu sehen.

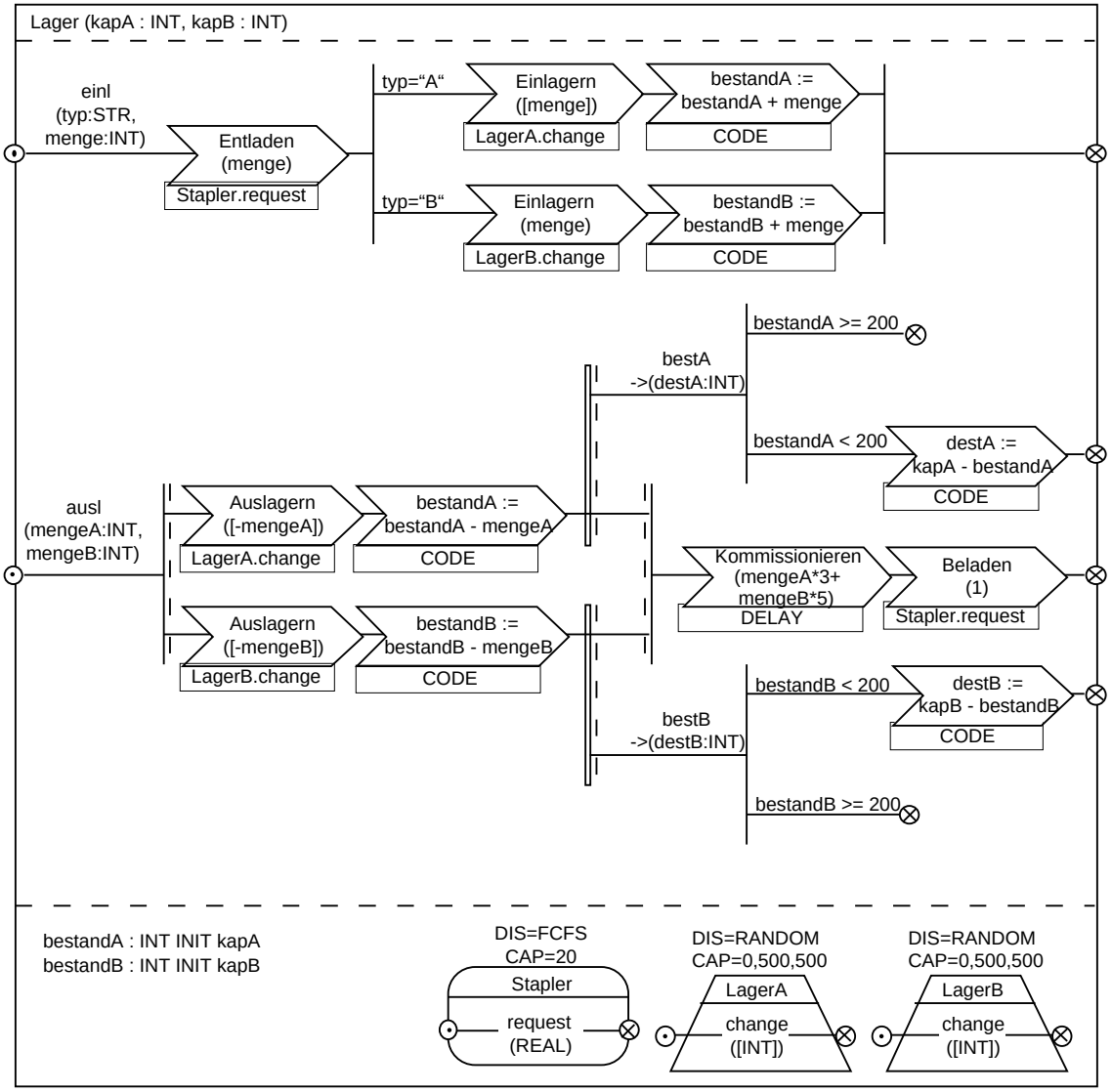


Abbildung 5-20: Funktionseinheit Lager

Zunächst werden die Objekte vom angekommenen LKW entladen. Zum Ent- und Beladen von LKWs stehen 20 Gabelstapler zur Verfügung. Diese werden durch eine Standardfunktionseinheit vom Typ Server modelliert. Nach dem Entladen werden die Objekte je nach Objekttyp in das interne Lager ein-

gelagert. Die Fallunterscheidung wird durch einen ODER-Konnektor erreicht. Die internen Lager werden durch zwei Standardfunktionseinheiten vom Typ Counter beschrieben. Das Auslagern von Objekten aus dem Lager geschieht über den Dienst *ausl* (auslagern). Dieser Dienst wird mit zwei Parametern *mengeA* und *mengeB* aufgerufen. Diese Parameter geben die Mengen der auszulagernden Objekttypen an. Nach dem Auslagern werden die Objekte kommissioniert und zu einer Ladung zusammengefaßt. Deswegen müssen vor dem Auslagern und dem Kommissionieren zeitsynchrone UND-Konnektoren verwendet werden. Nachdem die Ladung fertiggestellt ist, kann sie auf einen LKW geladen werden.

Zusätzlich zu den beiden Diensten *einl* (einlagern) und *ausl* (auslagern) besitzt diese Funktionseinheit noch die beiden Dienste *bestA* (A bestellen) und *bestB* (B bestellen). Diese beiden Dienste überprüfen nach jedem Auslagerungsvorgang die Bestände der internen Lager. Falls die Bestände die untere Grenze von 200 unterschreiten, wird eine Nachbestellung durchgeführt. Das Überprüfen der Bestände wird durch zwei zeitsynchrone UND-Konnektoren erreicht. Diese werden hinter die jeweiligen Auslagerungsaktivitäten geschaltet und stoßen die „Prozesse“ der Nachbestellung an. Falls eine Nachbestellung nötig ist, wird der freie Platz im jeweiligen internen Lager ermittelt und die entsprechende Quelle zur Erzeugung von neuen Objekten angestoßen. Die aktuelle Belegung des Lager wird in zwei globalen Variablen (*bestandA* und *bestandB*) festgehalten.

5.4.2 Das Gesamtmodell nach Schürholz

Die folgende Darstellung entwickelt schrittweise ein B1-Distributionsmodell, basierend auf der Darstellung /SCHÜ99/, eingeschränkt gemäß DIST_BSP (vgl. 3.4.2). Bezeichnungen der Form „Bezeichnung“ greifen wieder die Terminologie aus /SCHÜ99/ auf.

1. Die Strukturierung der Darstellung erfolgt top-down (also in umgekehrter Reihenfolge wie in /SCHÜ99/ bzw. 3.4.2). Unterschiede zu /SCHÜ99/ ergeben sich insbesondere aufgrund der Tatsache, daß in /SCHÜ99/ nicht auf eine Unterscheidung von „Prozessen“ und „Bausteinen“ Wert gelegt ist, wohingegen das B1-Paradigma die explizite Unterscheidung von Prozeßketten (PKs), den „Prozessen“ entsprechend, und Funktions-/Organisations-Einheiten (FEs), den „Bausteinen“ entsprechend, vorsieht.

   Die Darstellung betrachtet einen einzelnen Distributions–“Standort“, ein Verteillager. Der „Standort“ wird durch eine FE namens *DSO* erfaßt. Seine Schnittstellen nach außen sind im letzten Abschnitt von 3.4.2 beschrieben; ebenfalls die Charakteristika der An- und Auslieferungen.

2. Wie bei /SCHÜ99/ wird auf Aufträge (bzw. ihnen zugeordnete Lieferungen) als Leistungsobjekte von Prozessen fokussiert; Fahrzeuge, Paletten, Lieferobjekte treten nicht als Individuen auf, und damit auch nicht als Leistungsobjekte diesbezüglicher Prozesse.

3. Als Konkretisierung der Ebene der Globalen Steuerungen (vgl. 3.2.2.4) erhalten wir in Abbildung 5-21 die oberste Schicht eines Modells eines einzelnen Distributionsstandorts in seiner Einbettung in eine Umgebung, welche Anlieferungen erfolgen läßt bzw. Auslieferungen anstößt, indem in gewissen zeitlichen Abständen Prozesse der PKs *Anlieferung* bzw. *Auslieferung* gestartet werden. Diese PKs enthalten je eine einzige Aktivität, *einliefern* bzw. *ausliefern*, welche an *DSO* zur Ausführung verwiesen werden.

   Zur Erhöhung der Lesbarkeit und insbesondere als Verständnisbrücke zur ursprünglichen Darstellung (vgl. 3.4.2) sind (zusätzlich zu den Kommentierungsoptionen des B1-Paradigmas) weitere {…}-Beschriftungen eingebracht. Bzgl. der {Kommentare} an Verbindungen zwischen Aktivitäten bzw. zwischen Aktivitäten und Quellen/Senken sei aber explizit darauf hingewiesen, daß die Interpretation von Verbindungen im B1-Paradigma **nicht** daraus besteht, daß über diese Verbindungen Material, Informationen etc. fließen. Vielmehr dienen B1-Verbindungen nur der Notierung der unmittelbaren Aufeinanderfolge von Aktivitäten. Lediglich implizit ist mit einer Verbindung zwischen einer Vor-Aktivität und einer Nach-Aktivität auch ausgedrückt, daß in der Vor-Aktivität geänderte / erzeugte Materialien und Informationen mit Abschluß der Vor-Aktivität zur Verfügung stehen.

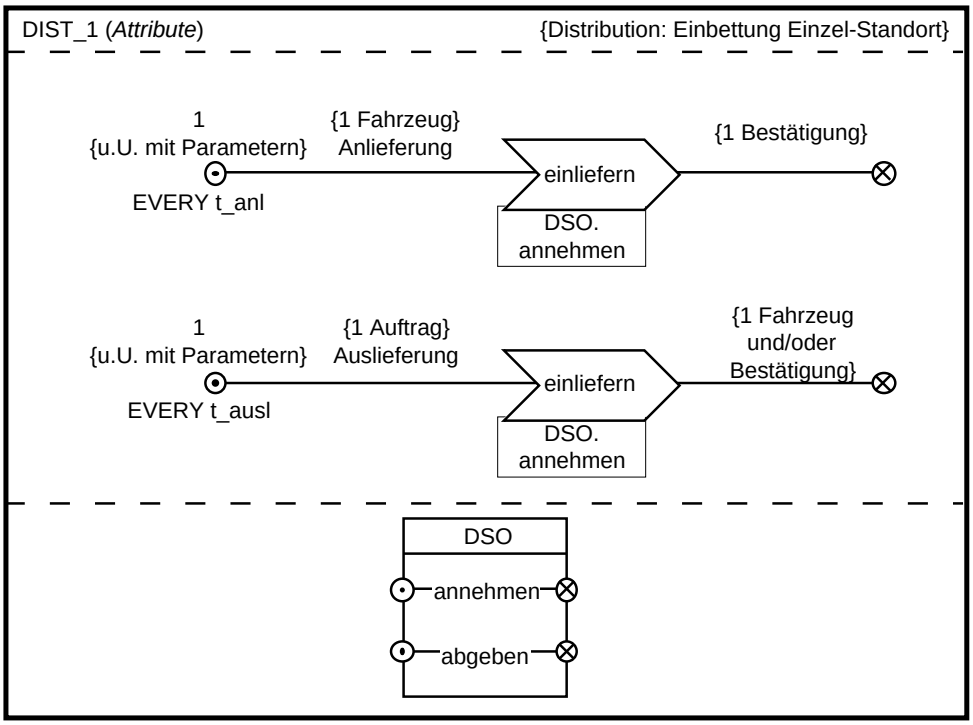


Abbildung 5-21

Die Beschriftung {u. U. mit Parametern} in Abbildung 5-21 soll andeuten, daß Anlieferungs- bzw. Auslieferungsaufträge die zu Ende von 3.4.2 genannten Mengeninformationen über transportierte bzw. zu transportierende Güter einbringen können. Dabei ist es naheliegend, bei (wie hier angenommen) nicht explizit modellierter Umgebung von *DSO* diesbezügliche Parameterwerte in den Quellen „auszuwürfeln“ (formaler: Realisierungen diesbezüglicher kennzeichnender Zufallsvariablen zu ziehen) und an die inkarnierten Prozesse zu übergeben. Alternativ zu diesem Vorgehen besteht regelmäßig die Möglichkeit, die Werte dieser Größen erst anläßlich ihrer ersten Verwendung (irgendwo „innerhalb“ des Modells) zu bestimmen.

4. Der Betrieb innerhalb *DSO* ist durch einen „Standort-Hauptprozeß“ erfaßt (in /SCHÜ99/ nicht explizit so benannt), vgl. 3.4.2.3: Ebene des Standortbausteins. *DSO* (vgl. Abbildung 5-22) enthält die „Physikalischen Modellbausteine“ „Entladebaustein“, „Lagerbaustein“ und „Beladebaustein“, welche hier durch diesbezügliche FEs *ENTL*, *LAG*, und *BEL* erfaßt werden. Zusätzlich sieht /SCHÜ99/ „Lokale Steuerungsbausteine“ für „Lagersteuerung“, „Behältersteuerung“ und „Fahrzeugsteuerung“ vor, welche in /SCHÜ99/ verbal beschrieben sind und hier geeignet integriert werden müssen. Konkret werden die Steuerungsaufgaben beim vorliegenden Vorschlag innerhalb konstruierter Lager-FEs („Bausteine“) wahrgenommen, sofern erforderlich (vgl. die Bemerkungen zu F_POOL des vorliegenden Punktes bzw. der Punkte 6 und 9).

   Wie schon in 3.4.2 diskutiert besteht zwischen Anlieferungen und Auslieferungen kein Leistungsobjekt-Zusammenhang. Der „Standort-Hauptprozeß“ wird daher durch zwei PKs erfaßt, die PKs *annehmen* und *abgeben*.

   Gleichfalls entsprechend der Anmerkungen in 3.4.2 wird bezüglich der „Lokalen Fahrzeugdisposition“ angenommen, daß

   - keine Leerfahrzeuge auftreten, so daß die PK *annehmen* eintreffende Fahrzeuge unmittelbar zu *ENTL* dirigiert (ein Umfahren von *ENTL* sowie ein Warten vor Weiterfahrt kann entfallen);

   - in *ENTL* entladene Fahrzeuge nach Verlassen von *ENTL* einem Fahrzeugpool *F_POOL* zugeführt werden, den sie allein auf Anforderung aus *BEL* heraus wieder verlassen (ein Ausschleusen von Leerfahrzeugen entfällt).

- Die PK *abgeben* ein in *BEL* benötigtes Fahrzeug vor Eintritt in *BEL* von *F_POOL* bereitstellen läßt.

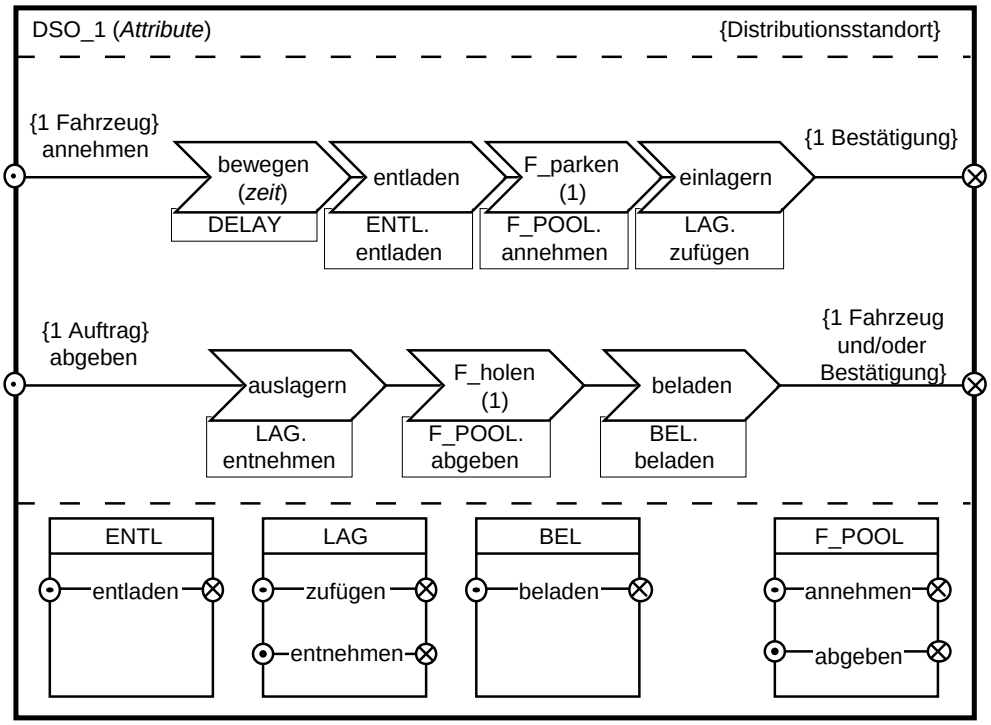


Abbildung 5-22

Zur weiteren Erklärung der Abbildung 5-22:

- *bewegen* (die Fahrzeit des Fahrzeugs zur Entladung) ist als reine Verzögerung mit der entsprechenden Fahrzeit parametrisiert;

- *F_POOL* könnte zur rein logischen Verwaltung verfügbarer bzw. fehlender Fahrzeuge durch eine Standard-FE vom Typ Counter ausgefüllt werden oder aber (bei vorliegender Notwendigkeit, die Fahrzeugdispositionen und Fahrzeugbewegungen in größerem Detail darzustellen) durch eine konstruierte FE ausgefüllt werden. Die Verwendung eines Standard-Counters ist, bei ähnlicher Sachlage, beispielhaft unter Punkt 6 beschrieben; die Konstruktion einer speziellen *F_POOL*-Funktionseinheit wäre analog zu den hier genutzten FEs *ENTL*, *LAG*, *BEL* (und ihren Konstruktionen unter Punkten 5, 6, 7) vorzunehmen.

Im Fluß der *annehmen*- und *abgeben*-Prozesse ist im Vergleich zu /SCHÜ99/ eine Sequentialisierung von Fahrzeug- und (Liefer-)Palettenmenge-Behandlung eingeführt. Diese ist nicht durch fehlende Möglichkeiten des B1-Paradigmas bedingt (Parallelisierungen der Behandlungen ließen sich unschwer ausdrücken), sondern erfolgt im Hinblick auf die Ermöglichung auch nichtsimulativer Modellanalysen. Die komplexeren FEs *ENTL*, *LAG* und *BEL* sind noch zu konstruieren (siehe Punkte 5, 6, 7). Sie werden die Dynamik der „Hauptprozesse“ von /SCHÜ99/ aufzunehmen haben (vgl. 3.4.2.2: Ebene der Bausteine).

5. In Weiterführung von 3.4.2.2 („Hauptprozeß“ „Entladung“ identisch mit „Modellprozeß“ „Entladung“, vgl. 3.4.2.1) erhält ENTL die Form entsprechend Abbildung 5-23.

   Hierin ist *entladen* mit der „Vorgabezeit“

   Zahl zu entladender Paletten (p_an) * Vorgabezeit Entladen je Palette (tE)

   parametrisiert und an einen Stapler-Pool *STAPLER* zur Ausführung verwiesen.

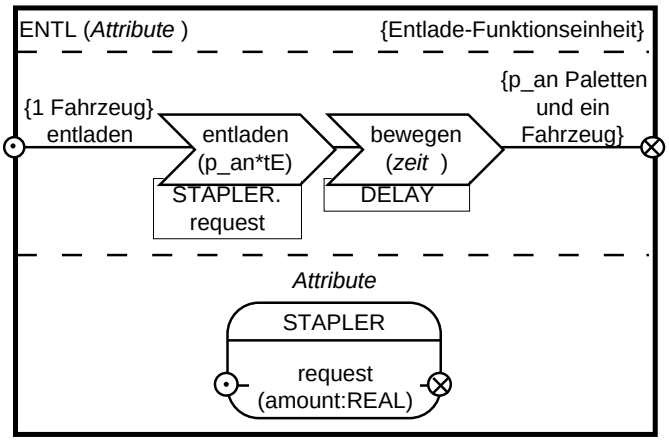


Abbildung 5-23

STAPLER ist hier als Standard-FE vom Typ Server erfaßt; *STAPLER* wird, je nach Anzahl vorhandener Stapler und je nach momentaner Belastung, diese Anforderung entsprechend schneller oder langsamer erfüllen. Die Anzahl (für eine bestimmte Untersuchung) vorgesehener Stapler kann bei Initialisierung des Modells in den *Attributen* von *ENTL* geschehen und an die *Attribute* von *STAPLER* weitergereicht werden (vgl. 4.2.6.1).

Bei Bedarf an detaillierteren Darstellungen der Verwaltungsmaßnahmen und Dispositionen innerhalb des Stapler-Pools könnte leicht eine entsprechende spezielle Funktionseinheit konstruiert und hier genutzt werden.

6. *LAG* ist entsprechend /SCHÜ99/ bzw. 3.4.2.2 weiter hierarchisch strukturiert und stützt sich auf die FEs *DEP_FKT* (zur Depalettierung), *LAG_FKT* (zur physischen Ein- und Auslagerung der Lieferobjekte), *KOM_FKT* (zur Kommissionierung) und *PAL_FKT* (zur Palettierung) ab (vgl. Abbildung 5-24). Zusätzlich vorgesehen ist eine FE *P_POOL*, in welcher leere Paletten zwischengelagert werden, und aus der benötigte Leerpaletten abgerufen werden (Standard-FE vom Typ Counter, „geeignet“ zu attributieren).

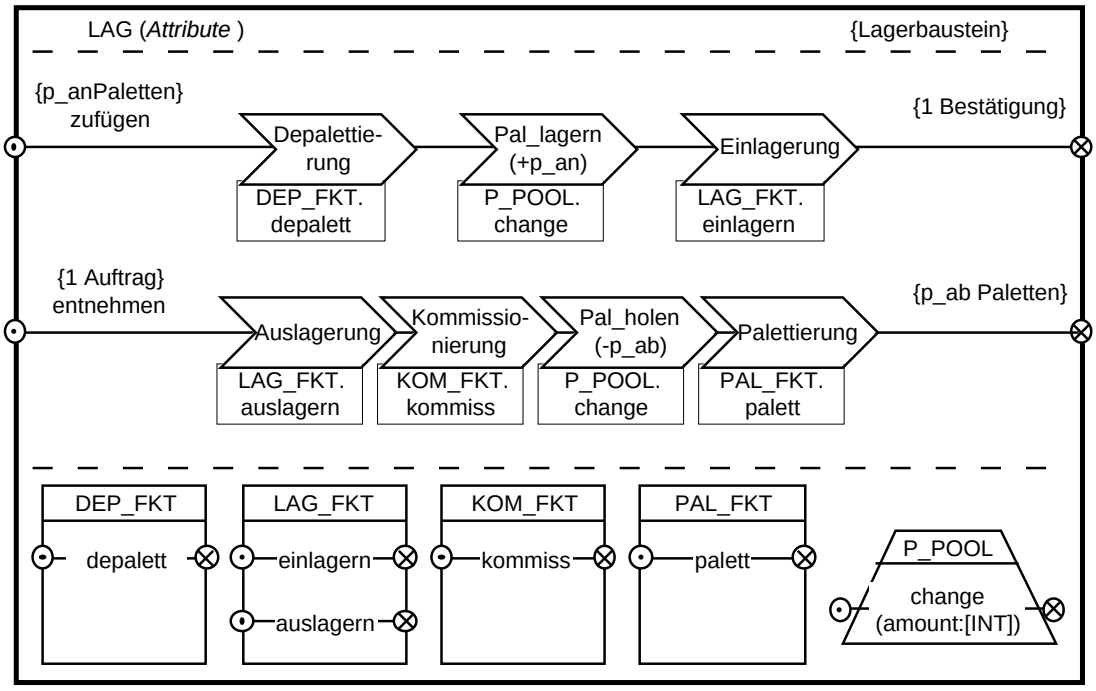


Abbildung 5-24

Im Fluß der *zufügen*- und *entnehmen*-Prozesse ist im Vergleich zu /SCHÜ99/ eine Sequentialisierung von Paletten- und Lieferobjekt-Behandlung eingeführt. Die Motivation dafür ist analog der entsprechenden Änderung in 4. zu sehen.

7. *BEL*, der dritte der „Physikalischen Modellbausteine“ von *DSO*, erhält in der B1-Fassung von DIST_BSP die Form der Abbildung 5-25 (vgl. auch 3.4.2.2 bzw. 3.4.2.1).

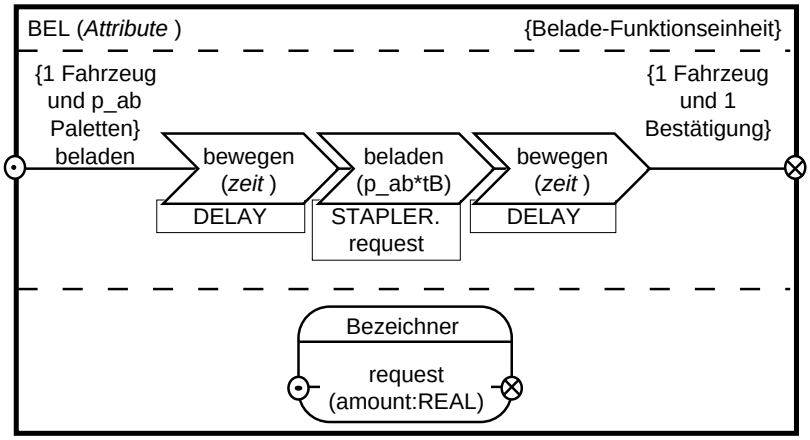


Abbildung 5-25

Die Entscheidungen (bzw. Änderungen gegenüber dem „Modellprozeß“ „Beladung“) betreffen:

- Eine Zuführung und Pufferung von Fahrzeugen entfällt aufgrund der Tatsache, daß *DSO* (siehe dort) das Fahrzeug schon zuvor bereitgestellt hatte.

- *beladen* wird mit der Vorgabezeit

  Zahl zu entladender Paletten (p_ab) * Vorgabezeit Beladen je Palette (tB)

  parametrisiert und an einen Stapler-Pool *STAPLER* zur Ausführung übergeben. *STAPLER* ist als Standard-FE vom Typ Server erfaßt Und noch geeignet zu parametrisieren). Es wird angenommen, daß *BEL* und *ENTL* je über eigene Stapler-Pools verfügen. Eine Zusammenlegung der Stapler-Pools wäre möglich.

- Eine (u. U. auch raumbegrenzte) Pufferung zu ladender Paletten kann direkt von FEs des Typs Server abgedeckt werden.

8. In der hierarchischen Verfeinerung von *LAG* fortfahrend (wir befinden uns auf der Ebene der Modellprozesse, vgl. 3.4.2.1) erhält *DEP_FKT* die Fassung entsprechend Abbildung 5-26.

   Die Entscheidungen (bzw. Änderungen gegenüber dem „Modellprozeß“ „Depalettierung“) betreffen:

   - Eine Zuführung und Pufferung von Fahrzeugen entfällt aufgrund der Tatsache, daß DSO (siehe dort) das Fahrzeug schon zuvor bereitgestellt hatte.

   - *depalettieren* wird mit der Vorgabezeit

     Zahl zu entladender Paletten (p_an) * Vorgabezeit Depalettieren / Palette (tD)

     parametrisiert und an einen Arbeiter-Pool *WORKER* zur Ausführung übergeben. *WORKER* ist als Standard-FE vom Typ Server erfaßt. Es wird angenommen, daß getrennte Arbeiter-Pools je Baustein existieren. Eine Zusammenlegung der Stapler-Pools wäre möglich.

Eine (auch raumbegrenzte) Pufferung zugeführter Paletten kann direkt von FEs des Typs Server abgedeckt werden.

- Die Abführung der Leerpaletten erfolgt außerhalb *DEP_FKT* (vgl. *LAG*).

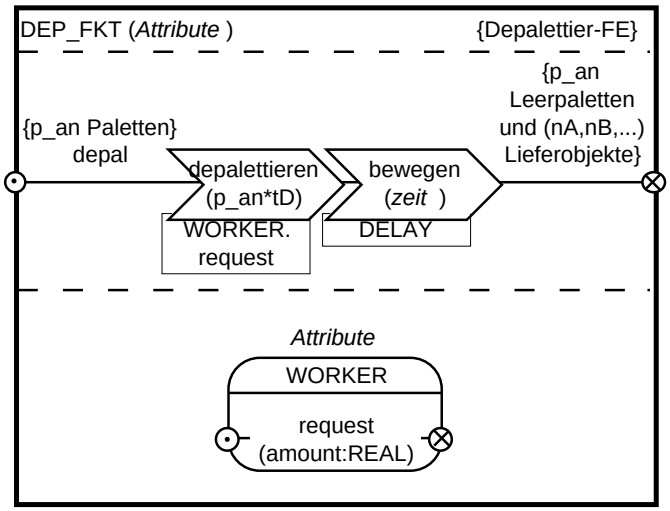


Abbildung 5-26

9. *LAG_FKT* verfügt, wie besprochen, über 2 Dienste, *einlagern* und *auslagern* welche das physische Lager (FE *PHYS_L* vom Standardtyp Counter) manipulieren. *einlagern* bemüht sich zunächst um hinreichenden Lagerplatz und belegt diesen, *auslagern* prüft zunächst die Verfügbarkeit angeforderter Mengen von Lieferobjekten. Die Arbeit des Ein- und Auslagerns wird durch ein Arbeiter-Team (FE *WORKER* vom Standard-Typ Server) vollzogen, wobei dieses Team sich, je nach anstehenden Aufgaben, die Arbeit des Ein-und Auslagerns teilt. Die Vorgabezeiten werden als proportional zur Anzahl der betroffenen Lieferobjekte angenommen, mit einer Vorgabezeit je Objekt von tL.

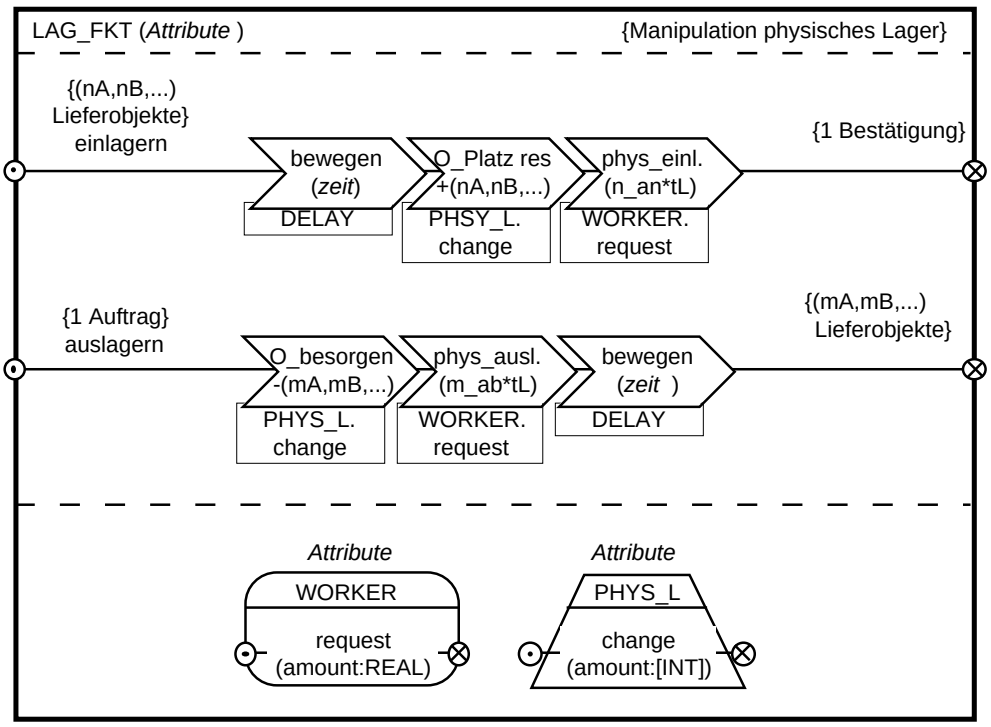


Abbildung 5-27

10. Kurz, und ohne detaillierte Kommentare, die FE *KOM_FKT* (Abbildung 5-28), in der eine Sortimentier-Vorgabezeit je Lieferobjekt von tS angenommen ist:

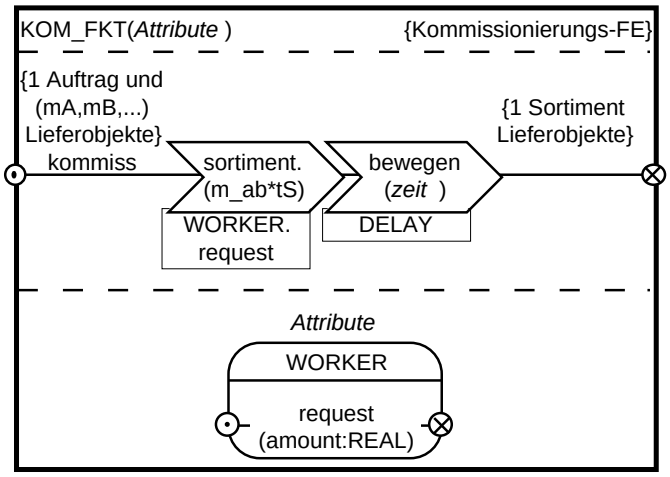


Abbildung 5-28

11. Und gleichfalls kurz und kommentarlos die FE *PAL_FKT* (Abbildung 5-29) mit einer Palettier-Vorgabezeit je Palette von tP:

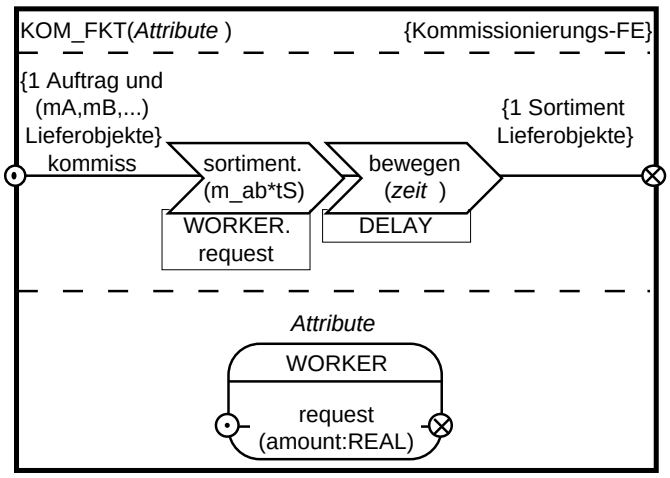


Abbildung 5-29

Als Fazit der Bearbeitung des Schürholz'schen Gesamtmodells läßt sich feststellen, daß sich mit dem B1-Formalismus auch umfassendere einschlägige Modellvorstellungen relativ mühelos darstellen lassen. Mannigfaltige Attributierungsmöglichkeiten (welche in obigen Skizzen weitgehend unausgefüllt blieben) erlauben dabei eine Modellierung in der Art von Referenzmodellen, die unter Setzung konkreter Attributsätze in entsprechende konkrete Modelle übergehen. Es soll nochmals hervorgehoben werden, daß nach Setzung der Attribute alle Informationen vorliegen, welche eine automatische Modellanalyse hinsichtlich leistungsorientierter Bewertungskriterien erlauben. Es sei zusätzlich darauf hingewiesen, daß die Modelle bzw. Referenzmodelle „nach unten offen“ sind in dem Sinne, daß verfeinernde Spezifikationen „unterer“ Funktionseinheiten, die hinsichtlich präziserer Erfassung von Details der erfaßten realen Systeme etwa wünschenswert erscheinen, sich mühelos einbringen lassen, wobei diese Verfeinerungen weitestgehend ohne Änderungen der jeweils bereits vorliegenden „höheren“ Modellkonstrukte spezifizierbar sind.

6 Einige Analyseergebnisse

Nachdem im vorangegangenen Kapitel eine mögliche Modellierung der Probleme im B1-Paradigma dargestellt worden ist, sollen in diesem Kapitel einige Ergebnisse zu möglichen Fragestellungen vorgestellt werden. Dazu wurden die im B1-Paradigma formulierten Modelle manuell in Eingaben für das Leistungsbewertungswerkzeug HIT (vgl. /HIT94/) umgesetzt und mit dem Werkzeug analysiert. In den folgenden Unterkapiteln werden die Analyseergebnisse für die Beispiele angegeben. Die Ergebnisgrößen (Population, Durchsatz, Auslastung, Verweilzeit) stammen aus dem Bereich der Leistungsanalyse. Sie können u. U. zur Berechnung von Kosten genutzt werden.

6.1 Güterverteilzentrum

In diesem Abschnitt werden einige Meßwerte präsentiert, die sich aus einem Simulationsmodell des GVZ ermitteln ließen (vgl. Kapitel 5.1). Das Güterverteilzentrum wurde in der einfachen Variante ohne Huckepackverkehr simuliert.

Mögliche Fragestellungen an das Modell des Güterverteilzentrums

Das Modell des Güterverteilzentrums kann unter einer Reihe von Fragestellungen untersucht werden. Fragen an das Modell könnten sein:

- Wie groß sind die Verweilzeiten von LKWs und Zügen?
- Wie groß ist der Durchsatz an Gütern?
- Wie groß ist die Auslastung der Gabelstapler?

Modellierte Lastsituation am GVZ

Um die Leistungsfähigkeit des GVZ beurteilen zu können, muß die auf das GVZ einwirkende Last (im Modell durch Quellen dargestellt) quantifiziert werden. Die hier präsentierten Ergebnisse wurden unter folgender Lastannahme gewonnen:

Alle sechs Minuten kommt ein LKW mit weiterzuleitenden Gütern am GVZ an. Jeder LKW hat im Mittel 2 Einheiten geladen. Die Güter sind für den Weitertransport mit dem Zug oder mit anderen LKWs bestimmt. Weiterhin treffen Züge am GVZ ein. Diese haben eine mittlere Zwischenankunftzeit von einer Stunde. Die Züge haben im Mittel 62,5 Einheiten ((50 Einheiten +75 Einheiten)/2)geladen und wollen im Mittel 62,5 Einheiten weiter transportieren. Sämtliche Umschlagvorgänge werden in einem Lager gepuffert. Von Überschreitungen der Lagerkapazität und Unterschreitungen des minimalen Lagerbestandes wird hier abstrahiert. Für das Umschlagen der Güter stehen 15 Gabelstapler zur Verfügung. Diese arbeiten mit einer mittleren Geschwindigkeit von einer Einheit pro Minute. Die folgende Tabelle faßt die Parametrierung des GVZ noch einmal zusammen.

Objekt	LKWs	Zug	Gabelstapler
Ankunftrate/(1/h)	**10**	**1**	
Mittl. Ladung/(Stck)	**2**	**62,5**	**1**
Geschwindigkeit/(Stck/Min)			**1**

Weitere Versuche könnten die Lastsituation variieren, um die Leistungsfähigkeit des Lagers z. B. unter Hochlast zu evaluieren. Andere denkbare Szenarien könnten die Ressourcen wie Lagerfläche oder die verfügbaren Gabelstapler variieren. Diese Variationen sind hier jedoch nicht betrachtet, da nur einige prinzipiell zu ermittelnde Ergebnisse dargestellt werden sollen.

Ergebnisse aus dem Modell

Eine Analyse des beschriebenen Modells liefert für die betrachteten Prozesse folgende Resultate:

Meßwert	Fahrzeug	GVZ	Terminal	Stacker (beladen)	Stacker (entladen)
Verweilzeit/ (Min)	**LKW**	**16,7**	**12,7**	**0,35**	**0,35**
Verweilzeit/ (Min)	**Zug**	**38,9**	**28,9**	**9,5**	**9,5**
Durchsatz/ (Stck/Min)	**LKW**	**0,16**	**0,16**	**0,16**	**0,16**
Durchsatz/ (Stck/Min)	**Zug**	**0,016**	**0,016**	**0,016**	**0,016**
Population/ (Stck)	**LKW**	**2,8**	**2,1**	**0,06**	**0,06**
Population/ (Stck)	**Zug**	**0,65**	**0,48**	**0,16**	**0,16**
Auslastung/ (%)	**LKW**			**4**	**4**
Auslastung/ (%)	**Zug**			**14**	**15**

Die Meßwerte sind nach Verursachern aufgeteilt. Aus der Tabelle lassen sich die durch die verschiedenen Nutzungen verursachten Wartezeiten ablesen. So hält sich z. B. ein LKW (erste Zeile in der Tabelle) im Mittel 16,7 Minuten im GVZ auf, davon verbringt er 12,7 Minuten im Terminal und beschäftigt den Reach-Stacker Pool 0,35 Minuten mit Entladen und 0,35 Minuten mit Beladen. Die kurze Belegung der Reach-Stacker ist durch die große Anzahl parallel arbeitender Einheiten (15) bedingt. Die Zug-Population (sechste Zeile in der Tabelle) beträgt im Mittel im GVZ 0,65.

Die Aufteilung der Meßwerte nach den Verursachern kann u. U. dazu dienen, Kosten gemäß der verursachten Belastung zu schlüsseln.

Die Auslastung der Gabelstapler beträgt nur 37% (Summe der Auslastungen durch die einzelnen Prozesse). Diese Auslastung scheint gering. In einem weiteren Versuch könnte die Anzahl der Gabelstapler reduziert werden.

6.2 Redistribution

Der Behälterkreislauf ist in einer einfachen Variante, ohne die Planungs- und Entscheidungsprozesse für die Behälterbedarfe an den verschiedenen Orten genau zu ermitteln, modelliert worden (vgl. Kapitel 5.2, analytisch lösbares Modell). Schwankungen der Behälterbedarfe an den Orten 1 und 2 werden durch die Wahrscheinlichkeit h_{31} ausgedrückt, die für einen einzelnen Behälter die Wahrscheinlichkeit, von Ort 3 nach Ort 1 geleitet zu werden, beschreibt. Wird der Behälter nicht nach Ort 1 geleitet, so wird er (mit Wahrscheinlichkeit 1- h_{31}) von Ort 3 nach Ort 2 gebracht. In ähnlicher Weise beschreibt h_{15} die Wegewahrscheinlichkeiten von Ort 1 nach Ort 5 und von Ort 1 nach Ort 4.

Fragestellungen an das Modell

Die im folgenden aufgeführten Ergebnisgrößen lassen sich aus dem Modell mit analytischen Verfahren bestimmen. Weitere interessierende Größen sind nur durch eine explizite Abbildung der Steuerung berechenbar. Diese Abbildung ist hier nicht gemacht worden, so daß weiterführende Ergebnisse nicht vorliegen.

Folgende Fragestellungen lassen sich aus dem analytischen Modell beantworten:

- Wie sind die Behälterbestände an den verschiedenen Orten?
- Wie groß ist die Anzahl benötigter Behälter im System?

Modellierte Lastsituation

Alle im folgenden ermittelten Ergebnisse beziehen sich auf die Zeiteinheit von einem Tag. Das Modell wurde analytisch gelöst. Bei den Ergebnissen handelt es sich um Mittelwerte.

Im Modell wird von einer gleichmäßigen Produktion ausgegangen, die jede Woche im Mittel 300 Behälter von Ort 3 verschickt. Der Anteil der Behältermenge, der aus der Produktion (Ort 3) zu den Kunden (Ort 1 und Ort 2) geht, wird durch den Parameter h_{31}=50% (Anteil nach Ort2: 1- h_{31}) festgelegt. Der Anteil der Behältermenge, der von Ort 1 zu Ort 5 geht, wird durch den Parameter h_{15}=50% (Anteil nach Ort4: 1- h_{15}) bestimmt. Die Behälter verbleiben an den Arbeitstagen an den Orten 1-5 und werden am Wochenende transportiert. Der Gesamtbestand ergibt sich aus den Behältermengen an den Orten 1-5. Dieses Vorgehen muß bei der Bewertung der vom Simulator gelieferten Zahlen berücksichtigt werden.

Ergebnisse aus dem Modell

Es wurden mit dem Werkzeug HIT Ergebnisse für die Orte 1, 3 und 5 ermittelt (Der Zeitbedarf für die Analyse beträgt nur wenige Sekunden). Weiterhin wurde der gesamte Behälterbestand im System ermittelt.

Verteilung der Behälter auf die Orte:

Orte	gesamt	Ort1	Ort2	Ort3	Ort4	Ort5
Mittlerer Behälterbestand/(Stck)= Mittlerer Durchsatz/Woche/(Stck)	**900**	**150**	**150**	**300**	**75**	**225**

6.3 Beschaffung

Das in Kapitel 5.3 vorgestellte Modell der Beschaffung ist in ein einfaches Simulationsmodell umgesetzt worden. Dabei sind folgende Modellparameter gesetzt worden:

Auslader/(Stck)	1
Maschinen/(Stck)	4-8
Prüfer/(Stck)	1
LKW/(Stck) (CAP-Wert der FE „Lkw“)	1-3

Fragestellungen an das Modell

Im Modell interessieren die Durchlaufzeiten der einzelnen Aufträge und ihre Beeinflussung durch die Beschaffung und die Anzahl der für die Fertigung zur Verfügung stehenden Maschinen. Hierzu wurde eine Eigenschaft des Lieferanten (die Anzahl der verfügbaren LKWs) sowie der Fertigung (Anzahl Maschinen) variiert.

Angenommene Lastsituation

Der Hersteller möchte pro Tag zwischen 750 und 850 Teile fertigen. Die Anzahl der zu fertigenden Teile ist zwischen 750 und 850 gleichverteilt.

Ergebnisse aus dem Modell

Im Simulationslauf mit nur einem LKW erwies sich diese Ressource als Engpaßressource. Daher wurde die Zahl der verfügbaren LKWs sowie die Anzahl der verfügbaren Maschinen in den folgenden Versuchen erhöht.

Maschinen/ (Stck)	5	6	5	6	7	7
LKW/ (Stck)	1	1	2	2	2	3
Verweilzeit/ (Min)	264256 (≈ 184 Tagen)	246244 (≈ 171 Tagen)	71103 (≈ 49 Tagen)	19113 (≈ 13 Tagen)	19027 (≈ 13 Tagen)	18968 (≈ 13 Tagen)
Bestand/ (Aufträge)	246	246	48	13	13	13
Durchsatz/ (Auftr/Min)	0,000231 (≈ 0,334 Tagen)	0,000231 (≈ 0,334 Tagen)	0,000615 (≈ 0,886 Tagen)	0,000683 (≈ 1 Tag)	0,000683 (≈ 1 Tag)	0.000688 (≈ 1 Tag)

Die Erhöhung der Maschinenanzahl von 5 auf 6 bringt bei nur einem LKW keine Verbesserung der Verweilzeiten, so daß vermutet werden kann, daß die Anzahl der LKWs zu gering ist. Daher wurde im nächsten Versuch die Anzahl der *Lkw* auf 2 gesetzt und der Attributparameter *Maschinen* auf 5 zurückgesetzt. Jetzt ist eine deutliche Verbesserung erzielt worden; der Durchsatz von einer Tagesproduktion/Tag wird aber immer noch nicht erreicht. Daher wird *Maschinen* wieder auf 6 gesetzt. Nun wird ein Durchsatz von einer Tagesproduktion erreicht. Eine weitere Erhöhung der Werte für *Lkw* und *Maschinen* bringt keine nennenswerte Steigerung der Leistung mehr. Der hohe Auftragsbestand läßt sich aus der Auftragsreihenfolgedisziplin bei den Maschinen der Fertigung erklären. Diese Maschinen arbeiten nach der FCFS-Strategie. So ist immer genau eine Maschine für einen Auftrag zuständig. Ein Auftrag hat eine mittlere Bearbeitungszeit von 8000 Minuten (5,5 Tage). Dazu kommen die sieben Tage Vorlauf für die Teilebeschaffung. Würde die Reihenfolgedisziplin der Maschinen auf PS umgestellt (die Last eines jeden Auftrags wird gleichmäßig auf die Maschinen verteilt), so sollte die Durchlaufzeit deutlich kleiner werden.

Ergebnisse mit Reihenfolgedisziplin PS:

Maschinen/ (Stck)	5
LKWs/ (Stck)	2
Verweilzeit/ (Minuten)	13319 (≈ 9 Tagen)
Bestand/ (Aufträge)	9
Durchsatz (Auftr/Min)	0,000687 (≈ 1 Tag)

6.4 Distribution

Das Modell der Distribution aus Kapitel 5.4.1 ist in ein Petri-Netz übersetzt und numerisch analysiert worden. Hierbei wurden einige vereinfachende Annahmen getroffen, die den Zustandsraum des Petri-Netzes begrenzen sollen. Auf diese Annahmen wird hier nicht weiter eingegangen.

Fragestellungen an das Modell

Das Modell wurde vor allem hinsichtlich der Kundenzufriedenheit analysiert. Hierbei spielt die Reaktionszeit auf eine Kundenbestellung die entscheidende Rolle.

Angenommene Lastsituation

Da keine konkreten Werte für die einzelnen Zeitverbräuche vorlagen, wurden alle Transitionsraten im Petri-Netz auf 1.0 gesetzt. Im Modell wurden die Anzahlen der verfügbaren LKWs und der verfügbaren Gabelstapler variiert. Aufgrund der vagen Annahmen über die Geschwindigkeit einzelner Ressourcen sind die numerischen Ergebnisse nicht praxisrelevant, zeigen aber die prinzipiellen Möglichkeiten der Analyse.

Ergebnisse aus dem Modell

Die folgende Tabelle gibt Auskunft über die Zeiten von der Auftragserteilung durch den Kunden bis zur Lieferung der Ware (Antwortzeit).

Anz. LKWs/ (Stck)	Anz. Gabelstapler/ (Stck)	Antwortzeit/ (Tage)
1	2	25,3
2	2	17,8
3	2	15,7
4	2	14,7
5	2	14,2
1	3	25,0
5	3	13,8

In der Tabelle läßt sich der Einfluß der verfügbaren Ressourcen auf die Antwortzeiten gut ablesen. Eine deutliche Erhöhung der verfügbaren Ressourcen läßt die Antwortzeit auf einen Wert von ca. 14. laufen. Diese Grenze entspricht einem Ablauf, bei dem der Auftrag an keiner Stelle auf Ressourcen warten muß.

7 Fazit

Mit der Entwicklung der B-Formalismen wird das Ziel verfolgt, ausgewählte Aspekte des grundlegenden (A-)Prozeßkettenparadigmas zu konkretisieren und zu präzisieren. Diese Konkretisierung stellt sich insbesondere der Aufgabe,

- Darstellungen von Prozeßkettenplänen derart zu vervollständigen und zu präzisieren, daß sie einer automatisierten Abbildung auf analysierende Techniken (insbesondere auf die Analysetechnik der dynamischen ereignisorientierten Simulation) zugänglich sind,

- bei den entwickelten Darstellungsformen insbesondere die konkreten Modellierungsprobleme der Anwendungs-Teilprojekte des SFB zu berücksichtigen,

- ohne sich von der Vorstellungswelt des A-Paradigmas (allzu weit) zu entfernen.

Der vorgestellte B1-Formalismus skizziert einen ersten Schritt in diese Richtung. Die Auswahl der berücksichtigten Aspekte des A-Paradigmas ist von einer Konzentration auf leistungsorientierte Beurteilungen von Prozeßketten-Modellen (als Abbild logistischer Systeme) geprägt – die zusätzliche Berücksichtigung von Kosten- und Umwelt-Aspekten (aber auch von Funktionsfähigkeits- und Zuverlässigkeits-Aspekten) verbleibt als offensichtliche Aufgabe nächster Schritte.

Der B1-Formalismus erreicht die gesteckten Ziel insoweit als

- die vorgestellten B1-Konstrukte und die daraus erstellbaren Prozeßkettenmodelle eine hinreichend präzise Bedeutung (Semantik) besitzen, um eine automatische Abbildung auf analysierende Techniken in Angriff nehmen zu können (Kapitel 6, in dem solche Abbildungen händisch vorgenommen wurden, belegt diese Aussage in der Form, daß sich die händischen Abbildungen "straightforward", insbesondere ohne jede Zufügung weiterer Informationen, vornehmen ließen),

- die uns zur Verfügung gestellten Beispiele der Anwendungsprojekte und die darin enthaltenen Modellierungsprobleme mittels des B1-Formalismus sämtlich (und wie wir meinen verständlich) erfaßt werden konnten (Kapitel 5 belegt diese Aussage),

- wobei allerdings der vorgeschlagene Grad der Abweichung vom A-Paradigma, und die damit implizierte Akzeptanzfrage, offensichtlich zukünftiger (Anwendungs-)Erfahrung bedarf.

Es sei zusätzlich darauf hingewiesen, daß der B1-Formalismus,

- obwohl er sich im Grunde (jenseits aller Details) auf wenige, relativ einfache Grundprinzipien und Grundkonstrukte abstützt,

- durch seine Optionen zur starken Strukturierung beschriebener Modelle die Hoffnung auf tatsächliche Modellierbarkeit auch (sehr) großer und komplexer Systeme begründet.

Es ist beabsichtigt, den B1-Formalismus in der Folgezeit, und in Kontakt mit anderen SFB-Teilprojekten, weiterhin intensiv hinsichtlich seiner Tauglichkeit und Akzeptanzfähigkeit zu überprüfen. Zusätzlich sollen Erweiterungen der B-Formalismen entwickelt werden mit dem Ziel der Berücksichtigungsfähigkeit von Kosten- und Funktionsfähigkeits-Aspekten. Erfahrungen und Ergebnisse dieser Arbeitsschritte sollen in einem (zukünftigen) B2-Formalismus zusammenfließen.

8 Literatur

/BECK96/ H. Beckmann: Theorie einer evolutionären Logistik-Planung – Basiskonzepte der Unternehmensentwicklung in Zeiten zunehmender Turbulenz unter Berücksichtigung des Prototypingansatzes, Dissertation Universität Dortmund 1996.

/HIT94/ H. Beilner, J. Mäter, C. Wysocki: The Hierarchical Evaluation Tool HIT, Short Papers and Tool Descriptions of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Vienna (Austria), 1994.

/KLÖP91/ H.-J. Klöpper: Logistikorientiertes strategisches Management, DGfL Logistik Leitfaden, R. Jünemann (Hrsg.), 1991.

/KUHN95/ A. Kuhn: Prozeßketten in der Logistik, Entwicklungstrends und Umsetzungsstrategien, Verlag Praxiswissen, A. Kuhn (Hrsg.), 1995.

/LOGI/ LogiChain Handbuch, Fraunhofer-Gesellschaft, Institut für Materialfluß und Logistik.

/PORT86/ M. E. Porter: Wettbewerbsvorteile, Campus Verlag, 1986.

/SCHÜ99/ A. Schürholz; Synthese eines Modells zur simulationsgestützten Potentialanalyse der Distribution; Dissertation Universität Paderborn 1999.

/ScKr95/ A. Schürholz, O. Krill: Referenzmodelle zur Lösung strategischer und taktischer Aufgabenstellungen der Distribution und Beschaffung, Fraunhofer-Gesellschaft, Institut für Materialfluß und Logistik, Projekt-Nr. 133612, Arbeitspaket-Nr. 5220, Lastenheft Teil I, Juni 1995.

/WiQu97/ G. Winz, M. Quint: Prozeßkettenmanagement: Leitfaden für die Praxis, Verlag Praxiswissen, A. Kuhn (Hrsg.), 1997.