Procescalculus bij modelleren van distributiesystemen

Procescalculus bij modelleren van distributiesystemen

In dit artikel gebruiken de auteurs procescalculus om een distributiesysteem te modelleren. Hierbij beperken zij zich tot het dynamische gedrag van het systeem als functie van de invoer en de uitvoer van orders en produkten. In het eerste model wordt het distributiesysteem gebouwd, waarbij er vanuit wordt gegaan dat een overmaat van produkten in het magazijn aanwezig is. In het tweede model wordt een leverancier geïntroduceerd die produkten aan het magazijn levert, en in het derde model wordt een bestelstrategie ingevoerd die zorgt voor de tijdige levering van produkten. In dit artikel wordt geen aandacht besteed aan het orderverzamelen.

In vorige artikelen [Rooda, Arentsen, 1991; Rooda, Arentsen, Smit, 1992] is getoond hoe procescalculus kan worden gebruikt voor het modelleren van flow-produktie en job-produktie (produkt- en proces-georiënteerde) fabrieken. In deze artikelen werd aangenomen dat deze produkten vanuit de fabriek hun eindbestemming vinden. In werkelijkheid bevindt zich meestal een distributiecentrum tussen de fabriek en bijvoorbeeld de groothandel. De functie van een distributiecentrum is enerzijds het overbruggen van de tijd van produktie en de tijd van afname en anderzijds het samenvoegen van produkten die afkomstig zijn van verschillende (gespecialiseerde) fabrieken voor een afnemer. Er zijn zeer veel verschillende typen distributiecentra. Het type wordt mede bepaald door het type produkt dat dient te worden gedistribueerd. Ook de snelheid waarmee het produkt door het distributiecentrum "loopt" geeft aanleiding tot verschillende typen distributiecentra. In dit artikel wordt hier niet op ingegaan. In dit artikel zal worden geïllustreerd hoe met behulp van procescalculus distributiecentra kunnen worden gemodelleerd. Met behulp van de procescalculus-simulator, de procescalculator, zal het gedrag van deze modellen worden onderzocht.

Het eerste model is te beschouwen als een rudimentair distributiecentrum. Via "stepwise refinement" zal dit model worden

Prof.dr.ir. J.E. Rooda, Ir. F.G.J. van Kempen, TU Eindhoven

uitgebreid en aangepast; een eenvoudig distributiecentrum zal worden gepresenteerd. Tenslotte wordt een distributiecentrum gemodelleerd waarin een informatiebeheerder zorgt dat de produkten die op basis van de orders dienen te worden uitgeleverd ook aanwezig zijn. Hiertoe wordt de informatiebeheerder voorzien van een algoritme dat zorgt voor het op tijd bestellen van de produkten. Van dit model wordt het verloop in de tijd van de hoeveelheid opgeslagen produkten gepresenteerd. In dit artikel wordt niet ingegaan op het orderverzamelen zoals dit in het distributiecentrum zelf gebeurt.

Orderregels, produkten, orders en pallets

Voor het modelleren van het distributiesysteem zal gebruik worden gemaakt van een aantal typen objecten: orderregels, produkten, orders, pallets en bestanden. Hierna zullen deze typen met hun methoden worden behandeld.
Iedere orderregel bevat informatie over het type produkt en het aantal van dat produkt dat moet worden uitgeleverd. Ieder produkt heeft een type en een aantal. Procescalculus maakt voor het modelleren van dergelijke instantiaties gebruik van de klasse (class) Associatie (Association) zoals deze in Smalltalk aanwezig is. De klasse Association is hierbij als volgt gedefinieerd:

```
Object subClass Association
instance variable names: key value
```

Een instantiatie kan hierbij worden gemaakt op de volgende manier:

```
a ← Association key: #P value: 10
```

De key van a kan worden opgevraagd met behulp van de opdracht key, de value van a kan worden opgevraagd met de opdracht value. In beide gevallen betreft dit het lezen van een waarde. Het zetten van de key of de value kan respectievelijk middels key: en value:.
Van deze associaties wordt nu gebruik gemaakt voor het definiëren van een regel of van een produkt:
Association subClass RegelOf Produkt

Om de leesbaarheid van de specificaties beter te vergroten worden nog enige methodes toegevoegd:

```
type: eenType aantal: eenAantal
  | rp |
  rp ← super new.
  rp key: eenType.
  rp value: eenAantal.
  ↑rp

type
  ↑key

aantal
  ↑value
```

Een regel en een produkt kunnen nu op de volgende wijzen worden gedefinieerd:

```
RegelOfProdukt subClass Regel

RegelOfProdukt subClass Produkt
```

Een nieuwe orderregel kan worden gemaakt door bijvoorbeeld de opdracht:

```
regel←Regel type: #A aantal: 10
```

Produkten worden gemaakt volgens een specificatie die op een orderregel staat aangegeven:

```
Produkt
maakVolgens: eenRegel
  | p |
  p ← self new.
  p key: eenRegel key.
  p value: eenRegel value
  ↑p
```

Een nieuw produkt kan worden gemaakt door bijvoorbeeld de opdracht:

```
produkt ← Produkt maakVolgens: regel
```

Hierin is regel de orderregel die in het vorige voorbeeld is gecreëerd.

Een order bestaat uit enige orderregels. Deze orderregels kunnen worden gecombineerd tot een order. De klasse Order is een subklasse van de klasse OrderedCollection:

```
OrderedCollection subClass Order
```

Dit samenvoegen kan op de volgende wijze, bijvoorbeeld:

```
order ← Order
```

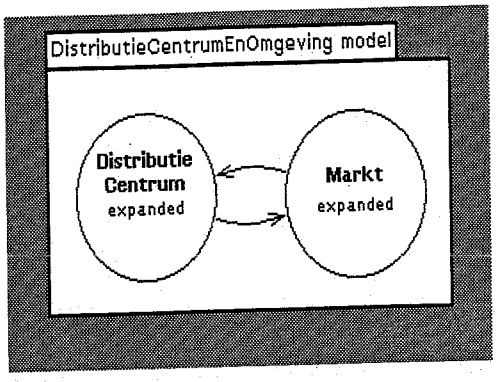


Fig.1.Het model van een Rudimentair distributiecentrum en zijn omgeving

```
  with: (Regel type: #A aantal: 10)
  with: (Regel type: #B aantal: 5)
  with: (Regel type: #C aantal: 1)
```

Deze order bevat 3 orderregels voor de produkten A, B en C.

Zo kunnen ook produkten worden samengevoegd op een pallet. De klasse Pallet is een sub-klasse van OrderedCollection:

```
OrderedCollection subClass Pallet
```

Pallet worden nu samengesteld op basis van een order:

```
Pallet
maakVolgens: eenOrder
  | pallet |
  pallet ← self new.
  eenOrder do:
    [: regel |
      pallet add: (Produkt
      maakVolgens: regel)].
  ↑pallet
```

Zo kan een pallet op de volgende wijze worden gecreeerd:

```
pallet ← Pallet maakVolgens: order
```

Hierin is een order het object dat in het vorige voorbeeld is gemaakt.

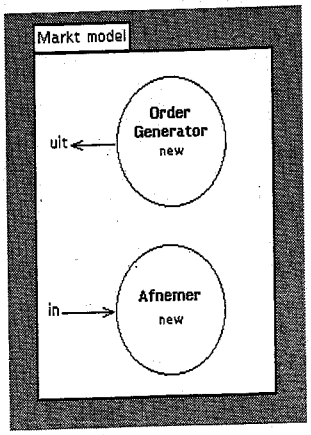


Fig.2.Het model van de Markt

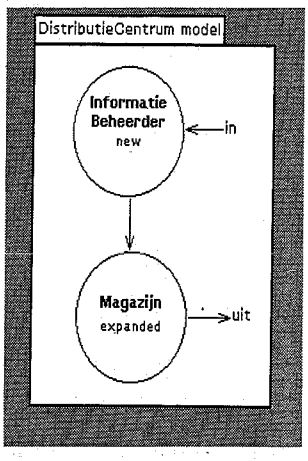


Fig.3. Het model van het distributiecentrum

Bestanden

De verschillende processoren dienen allerlei informatie bij te houden over de voorraad, de bestelgrens en dergelijke. Deze informatie bestaat in dit geval uit een verzameling van typen en aantallen. Deze veelvoorkomende verzameling is reeds beschreven in Smalltalk en wordt Dictionary genoemd. Een object van de klasse Dictionary bestaat uit een verzameling van een (in beginsel onbekend) aantal associaties, waarbij iedere associatie 2 waarden kan bevatten. Een Dictionary is een verzameling van elementen die niet geordend is zoals bij een OrderedCollection, maar waarvan de elementen via een sleutel (key) toch toegankelijk zijn. Een Dictionary kan men vullen met de methode at: put:. Het volgende voorbeeld illustreert dit:

```
d ← Dictionary new.
d at: #lente put: #spring
d at: #zomer put: #summer
d at: #herfst put: #autumn
d at: #winter put: #winter
```

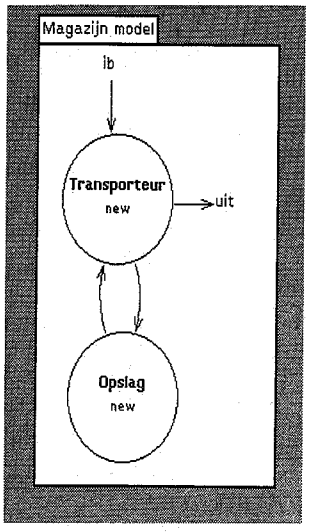


Fig.4. Het model van het magazijn

Een Dictionary kan men lezen met de opdracht at:.

```
seizoen ← d at: #herfst
"seizoen wordt #autumn"
```

Ten behoeve van de verschillende processoren worden zo de volgende klasse en de bijbehorende methoden gedefinieerd:

```
Dictionary subClass Bestand

typen: eenRijTypen hoeveelheden: eenRijHoeveelheden
  | b |
  b ← self new.
  eenRijTypen
    with: eenRijHoeveelheden
    do: [: key : value |
      b at: key put: value].
  ↑b

van: eenType
  ↑self at: eenType

verhoogMet: eenRegelOfProdukt
  self at: eenRegelOfProdukt key
    put: (self at:
    eenRegelOfProdukt key)
      + eenRijOfProdukt value

verlaagMet: eenRegelOfProdukt
  | nieuweWaarde |
  nieuweWaarde ← (self at:
  eenRegelOfProdukt key)
    - eenRegelOfProdukt value.
  nieuweWaarde < 0
    ifTrue: [self error:
    'nieuweWaarde < 0].
  self at: eenRegelOfProdukt key
    put: nieuweWaarde
```

De klasse Bestand erft (inheritance) de eigenschappen van Dictionary, zie ook [Rooda, 1992]. De methode typen: hoeveelheden: wordt gebruikt om een bestand aan te maken en initieel te vullen. De methode van: is geïntroduceerd om de leesbaarheid te vergroten. De twee methoden verhoogMet: en verlaagMet: worden gebruikt voor het bijwerken van bestanden.
Voor de bestellijst wordt daarnaast nog gebruik gemaakt van twee methoden voor het toevoegen en verwijderen van bestellingen. Hiertoe wordt een nieuwe klasse Bestellijst geïntroduceerd:

```
Bestand subClass BestelLijst

voegToe: eenProdukt
  self at: eenProdukt put: true

  verwijder: eenProdukt
  self at: eenProdukt put: false
```

Een rudimentair distributiecentrum

Als eerste model van een distributiesysteem beschouwen we een systeem waarin een onbeperkte hoeveelheid van een drietal produkten is opgeslagen. Het systeem is verbonden met zijn omgeving door middel van een Markt, figuur 1.
De Markt bestaat uit een OrderGenerator en een Afnemer, figuur 2. De OrderGenerator genereert iedere halve dag (= 4 uur) een order voor het distributiesysteem. Iedere order bevat een bestelling voor 10 produkten van type A, 5 produkten voor type B en 1 produkt voor type C. Iedere order bevat dus 3 orderregels. Dit model zal worden gebruikt om (partieel) de materie- en de informatiestromen te modelleren. In het volgende model zal dit model worden uitgebreid met een Leverancier die zorgt voor navulling van het Magazijn.

De beschijving van de OrderGenerator kan nu als volgt luiden:

```
Processor subClass
OrderGenerator
body
  self send:
    (Order
      with: (Regel type: #A
      aantal: 10)
      with: (Regel type: #B
      aantal: 5)
      with: (Regel type: #C
      aantal: 1)
    )
  to: 'uit'.
  self workDuring: 4 hours
```

De Afnemer kan eenvoudig worden beschreven door:

```
Processor subClass Afnemer
body
  | pallet |
  pallet ← self receiveFrom: 'in'
```

Het DistributieCentrum bestaat uit een InformatieBeheerder en een Magazijn, figuur 3. De InformatieBeheerder zorgt ervoor dat een order die wordt ontvangen van de OrderGenerator wordt doorgestuurd naar het Magazijn:

```
Processor subClass Informatie-
Beheerder
body
  self send: (self receiveFrom: 'in')
  to: 'magazijn'
```

In het Magazijn bevinden zich een

Transporteur en een Opslag, figuur 4.
De Transporteur ontvangt een order van de InformatieBeheerder. Deze order dient vervolgens te worden verzameld en te worden verstuurd naar de Afnemer:

```
Processor subClass Transporteur
body
  self slaUit: (self receiveFrom: 'ib')
```

De order wordt regel voor regel verzameld. Door de Transporteur wordt een regel naar de processor Opslag verstuurd. Nadat de orderRegel door de Opslag is afgewerkt, wordt de pallet door de Transporteur ontvangen, waarna een opdracht voor de volgende regel naar de Opslag wordt verstuurd. De produkten worden verzameld in de (tijdelijke) variabele pallet. Hierbij levert de methode slaUit: een pallet af:

```
slaUit: eenOrder
  | pallet produkt |
  pallet ← OrderedCollection
new.
  eenOrder do:
  [: regel |
  self send: regel to: 'info'.
  produkt ← self receiveFrom:
  'opslag'.
  pallet add: produkt].
  self send: pallet to: 'uit'
```

Het orderverzamelen in de Opslag wordt gedefinieerd door een tijd van 10 seconden voor een enkel produkt. Er is verondersteld dat er een onbeperkt aantal van de drie produkten aanwezig is. In de processor wordt een regel direct geconverteerd tot een produkt door de opdracht produkt ← regel. Omdat voldoende produkten worden verondersteld is het niet strikt nodig om de produkten apart te modelleren. De beschrijving van de Opslag kan nu luiden:

```
Processor subClass Opslag
body
  | regel produkt |
  regel ← self receiveFrom: 'info'.
  self workDuring: regel aantal *
  10 seconds.
  produkt ← regel.
  self send: produkt to: 'uit'
```

Met behulp van de procescalculussimulator, de procescalculator, kan nu worden vastgesteld dat er in één week 100 produkten A, 50 produkten B en 10 produkten C door het distributiesysteem zijn geleverd. Dit model wordt in het volgende model uitgebreid met een Leverancier.

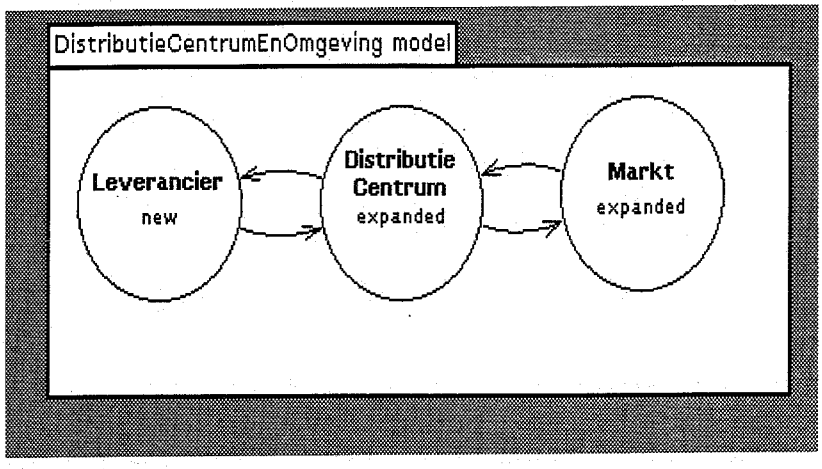


Fig.5. Het model van een distributiecentrum en zijn omgeving

Een eenvoudig distributiecentrum

Figuur 5 toont het nieuwe model van het distributiecentrum en zijn omgeving. Het model van de Markt is identiek aan figuur 2. Het model van het distributiecentrum is weergegeven in figuur 6.
Voorlopig wordt aangenomen dat de Leverancier dadelijk de gevraagde produkten kan leveren. In deze processor wordt de order als het ware direct getransformeerd naar materiaal:

```
Processor subClass Leverancier
body
  self send:
    (Pallet maakVolgens:
      (self receiveFrom: 'in'))
  to: 'uit'
```

De taak van de InformatieBeheerder, figuur 6, is nu wat complexer geworden. De InformatieBeheerder ontvangt van de Markt een order. Een kopie van deze order wordt doorgestuurd naar de Leverancier en de order zelf wordt doorgestuurd naar het Magazijn (de kopie van de order en de order zijn vanzelfsprekend verwisselbaar):

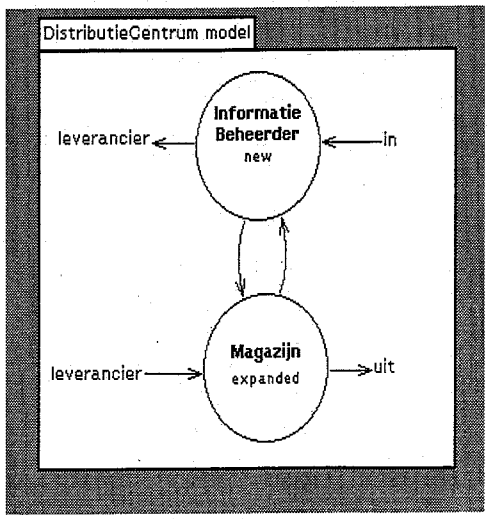
```
Processor subClass Informatie-
Beheerder
body
  | order |
  order ← self receiveFrom: 'in'.
  self send: order copy to:
  'leverancier'.
  self send: order to: 'magazijn'
```

Het Magazijn is weergegeven in figuur 7. De Transporteur is nu ingewikkelder dan in het vorige model. De Transporteur zorgt namelijk zowel voor de uitslag van de in opslag zijnde produkten als voor de inslag van nieuwe produkten.
Een order die binnenkomt van de InformatieBeheerder wordt doorgeleid naar de Opslag, waarna de Opslag de produkten verzamelt (uitslag) en de Transporteur de produkten verstuurt naar de Markt.
Produkten die zijn verstuurd door de Leverancier dienen door de Transporteur in het magazijn te worden ingeslagen. De beschrijving van de Transporteur luidt nu als volgt:

```
Processor subClass Transporteur
body
  self receiveFrom: 'ib'
  then: [: order |
    self slaUit: order]
  or: 'leverancier'
  then: [: pallet |
    self slaIn: pallet]
```

waarin slaUit: reeds eerder is gedefinieerd.
De methode slaIn: regelt de ontvangst en de verwerking van

Fig.6. Het model van het distributiecentrum



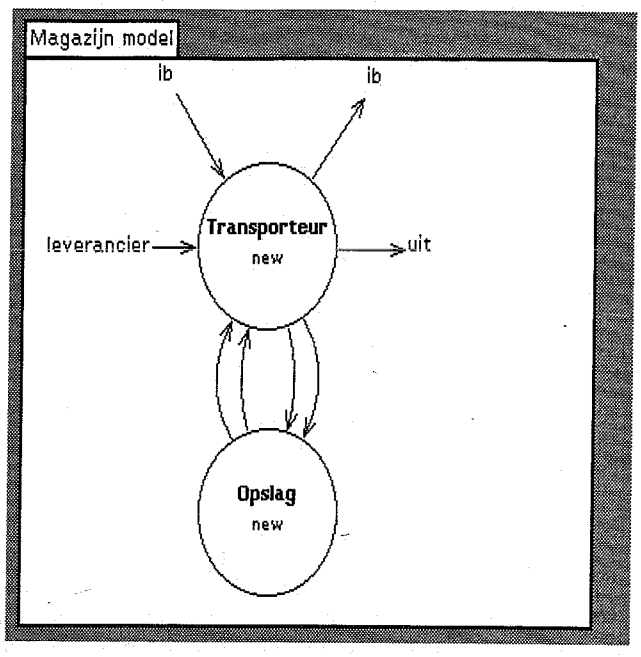


Fig.7.Het model van het magazijn

```
pallets:
slaIn: eenPallet
  eenPallet do:
    [: produkt |
    self send: produkt to: 'opslag'.
    self receiveFrom: 'info'].
  self send: eenPallet copy to: 'ib'
  orReceiveFrom: 'ib'
  then:
    [ :order |
    self slaUit: order.
    self send: eenPallet copy to:
'ib']
```

Deze methode is ingewikkeld, omdat gedurende de verwerking van leveringen ook orders binnen kunnen komen (zie ook de opgaven).
De processor Opslag kan nu worden beschreven door:

```
Processor subClass Opslag
instance variable names: opslag

initializeTasks
  opslag ← Bestand
    typen: #(A B C)
    hoeveelheden: #(0 0 0).
```

Initieel is de opslag leeg.

```
body
  | produkt |
  self receiveFrom: 'info'
  then:
    [: regel |
    self workDuring: regel aantal
    * 10 seconds.
    produkt ← regel.
    opslag verlaagMet: produkt.
    self send: produkt to: 'uit' ]
  or: 'in'
  then:
    [: produkt |
    self workDuring: produkt
    aantal * 10 seconds.
    opslag verhoogMet: produkt.
    self send: 'klaar' to: 'info' ]
```

Met behulp van de processcalculator kan opnieuw worden vastgesteld dat er in 1 week 100 produkten, 50 produkten B en 10 produkten C kunnen worden gedistribueerd.
In de praktijk worden bestelde produkten niet direct geleverd: er bevindt zich altijd enige tijd tussen bestelmoment en levermoment, de levertijd is altijd groter dan nul, omdat de leverancier enige tijd nodig heeft om de produkten te leveren. In het volgende model wordt de processor Leverancier aangepast. Daarnaast zijn de orders die bij een Distributiecentrum worden geplaatst meestal niet even groot. Dit betekent dat de InformatieBeheerder dient te worden voorzien van een bestelalgoritme dat er voor zorgt dat er altijd voldoende voorraad is. Daarnaast zullen orders van verschillende grootte aan het Distributiecentrum worden toegevoerd. Hiertoe wordt de processor OrderGenerator aangepast.

Een distributiesysteem

Het model van het distributiecentrum en zijn omgeving is weergegeven in figuur 5. Er wordt aangenomen dat de Leverancier een levertijd van 5 dagen heeft. In het nieuwe model is de processor Leverancier geexpandeerd in een processor Producent en een processor Expediteur, figuur 8. De Producent ontvangt onmiddelijk de orders van de InformatieBeheerder. De Producent heeft 3 uren nodig om het bij de order behorende pallet naar de processor Expeditie te verzenden:

```
Processor subClass Producent
body
  | order |
  order ← self receiveFrom: 'in'.
  self workDuring: 3 hours.
  self send:
    (Pallet maakVolgens: order)
  to: 'uit'
```

De processor Expediteur verzamelt de pallets gedurende een periode van 40 uren, 5 dagen. Nadat de periode is verstreken wordt (eventueel) een nieuwe pallet met alle geproduceerde produkten verzonden. De Expediteur kan nu worden gedefinieerd door:

```
Processor subClass Expediteur
instance variable names: periode

initializeTasks
  periode ← 40 hours

body
  | vorigeVerzendTijd
  nieuwePallet |
  vorigeVerzendTijd ← self time.
  nieuwePallet ← Pallet new.
  [ self time - vorigeVerzendTijd <
  periode ]
  whileTrue:
    [self
      receiveFrom: 'in'
      before: vorigeVerzendTijd
      + periode
      then:
        [ :p |
        p do:
          [: produkt |
          nieuwePallet addLast:
          produkt ]
        ]
        ].
  nieuwePallet isEmpty not
    ifTrue: [ self send:
    nieuwePallet to : 'uit' ]
```

Hierin is de variabele nieuwePallet de pallet die opnieuw wordt aangemaakt. Indien tijdens de periode niets door de Producent wordt aangeleverd, dan wordt er geen lege nieuwe pallet verzonden. De door de Producent aangeleverde pallet is aangegeven door de variabele p.

De processor OrderGenerator,

figuur 2, wordt nu op de volgende wijze aangepast. Drie kansverdelingen worden gebruikt om vast te stellen hoe groot de order zal zijn. De waarden van de hoeveelheden te bestellen produkten A, B en C liggen tussen respectievelijk 0 - 20, 0 - 10 en 0 - 2. De gemiddelde waarden van deze discrete uniforme verdeling [Rooda, 1992] bedragen respectievelijk 10, 5 en 1. De beschrijving van de processor OrderGenerator luidt nu:

```
Processor subClass OrderGenerator
instance .variable names: randomA, randomB, randomC

initializeTasks
  randomA ← SampleSpace data: (0 to: 20).
  randomB ← SampleSpace data: (0 to: 10).
  randomC ← SampleSpace data: (0 to: 2)

body
  | order aantal |
  order ← Order new.
  aantal ← randomA next.
  aantal > 0
    ifTrue: [ order add: (Regel key: #A value: aantal)
  aantal ← randomB next.
  aantal > 0
    ifTrue: [ order add: (Regel key: #B value: aantal)
  aantal ← randomC next.
  aantal > 0
    ifTrue: [ order add: (Regel key: #C value: aantal)
  order isEmpty not
    ifTrue: [ self send: order to: 'uit'].
  self workDuring: 4 hours
```

Deze specificatie is zodanig dat alleen orderregels voor produkten worden gecreëerd, indien het aantal > 0. Indien een order geen orderregels heeft, dan wordt de order niet verstuurd (Dit zal in ongeveer 0,1 % van de orders het geval zijn).

Zoals eerder vermeld dient de InformatieBeheerder te worden voorzien van een bestelalgoritme dat ervoor zorgt dat de opslag niet "uit voorraad loopt". In de literatuur worden verschillende bestelstrategieën beschreven [Monhemius, 1987]. Een eenvoudige bestelregel is de zogenaamde (B, S) - beslissingsregel. Bij deze regel wordt indien de voorraad onder het bestelniveau B is gekomen een hoeveelheid besteld die gelijk is aan een bestelgrens S minus de aanwezige voorraad.
De bestelgrenzen S worden bepaald op 400, 300 en 200. De bestelniveaus B worden arbitrair gelegd op 0,2 * de bestelgrenzen en bedragen 80, 60 en 40. De InformatieBeheerder kan nu worden beschreven door:

```
Processor subClass InformatieBeheerder
instance variable names: voorraad bestelgrens bestelniveau
  bestellijst naleveringen

initializeTasks
  | deTypen |
  deTypen ← #(A B C).
  voorraad ←
    Bestand
      typen: deTypen
      hoeveelheden: #( 200 150 100 ).
  "liggen reeds in de processor Opslag"
  bestelgrens ←
    Bestand
      typen: deTypen
      hoeveelheden: #( 400 300 200 )
  bestelniveau ←
    Bestand
      typen: deTypen
      hoeveelheden: #( 80 60 40 ).
  bestellijst ←
    Bestellijst
      typen: deTypen
      hoeveelheden: #( false false false ).
  naleveringen ← OrderedCollection new
```

In de bovenbeschreven methode worden de bestanden van de InformatieBeheerder aangemaakt en initieel gevuld. De beschrijving van de InformatieBeheerder is redelijk voor de hand liggend: indien een order binnenkomt, verwerk de order en indien een levering binnenkomt, verwerk de levering. De volgende beschrijving geeft dit weer:

```
body
  self receiveFrom: 'in'
  then: [: order |
    self verwerkOrder: order)
  or: 'magazijn'
  then: [: levering |
    self verwerkLevering: levering]
```

De methode verwerkOrder: onderzoekt of alle produkten op voorraad zijn. Indien dit het geval is, stuur dan de order naar het magazijn, verlaag vervolgens voor iedere regel van de order de voorraad en controleer de voorraad, waarbij eventueel een bestelling kan worden gegenereerd. Indien de produkten niet meer voldoende aanwezig zijn, bewaar dan de order. Deze order zal dan nageleverd dienen te worden. De methode kan als volgt worden beschreven:

```
verwerkOrder: eenOrder
  (self isAllesOpVoorraad: eenOrder)
    ifTrue:
      [self send: eenOrder to: 'magazijn'.
      eenOrder do:
        [: regel | voorraad verlaagMet: regel].
      self controleer Voorraad En Bestel: eenOrder
      ]
    ifFalse:
      [self bewaar: eenOrder]
```

De methode isAllesOpVoorraad levert true indien alle produkten op voorraad zijn, anders false. Alle regels van de order worden vergeleken met de voorraad. Indien van de produkten niet voldoende aanwezig is, dan wordt de gehele order voorlopig niet uitgeleverd (Dit is uiteraard een arbitraire keus):

```
isAllesOpVoorraad: eenOrder
  eenOrder do:
    [: regel |
    (voorraad van: regel type) < regel aantal
      ifTrue: [↑false]
    ].
  ↑true
```

De methode controleerVoorraadEnBestel: wordt gedefinieerd door:

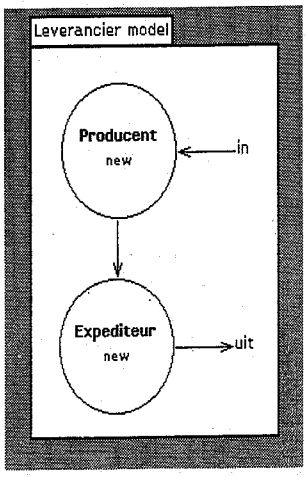


Fig.8. Het model van de leverancier

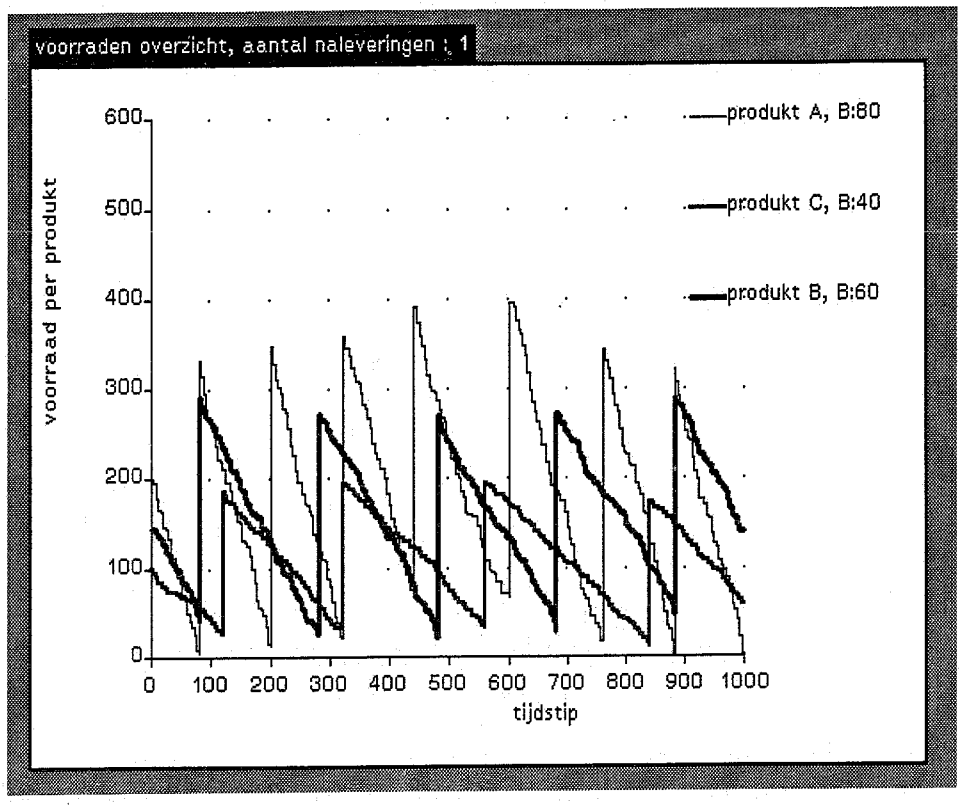


Fig.9. Het verloop van de voorraden in de tijd B=0,2 s

```
controleerVoorraadEnBestel:
eenOrder
  | bestelling produkt |
  bestelling ← Order new.
  eenOrder do:
    [: regel | produkt ← regel
    type.
    (self produktMoetWorden
    Besteld: produkt)
      ifTrue: [bestelling ←
        self vul: bestelling
        aanMet: produkt]
    ].
  bestelling isEmpty not
    ifTrue: [self send: bestelling
    to: 'leverancier']
```

Een produkt hoeft alleen te worden besteld indien het produkt nog niet is besteld en de voorraad kleiner is dan het bestelniveau. De methode produktMoetWordenBesteld: is gedefinieerd door:

```
produktMoetWordenBesteld:
eenProdukt
  ↑(bestellijst van: produkt) not
  and:
    (voorraad van: eenProdukt) <
    (bestelniveau van: Produkt)
```

Er wordt een orderregel besteld met een hoeveelheid die overeenkomt met het verschil van de bestelgrens en de nog aanwezige voorraad, de (B, S)-regel. De methode vul: aanMet: is gedefinieerd door:

```
vul: eenBestelling aanMet: een-
Produkt
  eenBestelling add:
    (Regel
      type: eenProdukt
      aantal: (bestelgrens van:
      eenProdukt)
        - (voorraad van:
        eenProdukt)
    ).
  bestellijst voegToe: eenProdukt.
  ↑eenBestelling
```

De methode bewaar: wordt gedefinieerd door:

```
bewaar: eenOrder
  naleveringen addLast:
  eenOrder
```

De methode verwerkLevering: wordt gedefinieerd door:

```
verwerkLevering: eenLevering
  eenLevering do:
    [: produkt |
    voorraad verhoogMet:
    produkt.
    bestellijst verwijder: produkt
    type].
  self controleerNaleveringen
```

De methode controleerNaleveringen wordt gedefinieerd door:

```
controleerNaleveringen
  | oudeLijst |
  oudeLijst ← naleveringen copy.
  naleveringen ← naleveringen
  removeAll.
  oudeLijst do:
    [: order | self verwerkOrder:
    order]
```

De processor Transporteur blijft ongewijzigd.

Initieel worden 200, 150 en 100 produkten A,B en C op voorraad gelegd. De processor Opslag wordt als volgt aangepast:

```
Opslag
initializeTasks
  opslag ← Bestand new.
  opslag typen: # (A B C) hoe-
  veelheden: # (200 150 100).
```

De body van de processor Opslag blijft ongewijzigd.

Met dit model van het complete distributiecentrum is het mogelijk om allerlei experimenten uit te voeren. Zo is het mogelijk om het verloop van de voorraden in de tijd weer te geven, figuur 9. Duidelijk is te zien dat de instelling van het bestelniveau groot genoeg is. Verlaging van het bestelniveau, naar bijvoorbeeld 0,05 * bestelgrens levert een geheel ander beeld, figuur 10.

Nabeschouwing

Er is een streven in de industrie

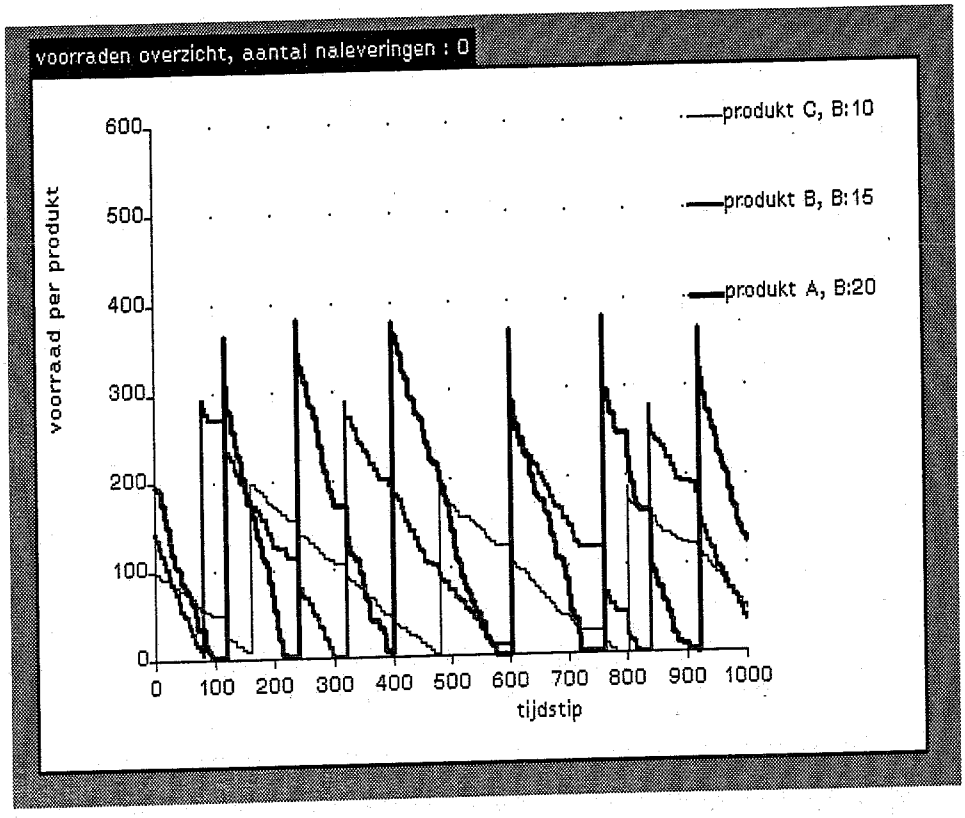


Fig.10. Het verloop van van de voorraden in de tijd B=0,05 s

om zoveel mogelijk alles precies op tijd te produceren ("just-in-time"). Distributie blijft er echter altijd nodig. Met name wanneer men enige specialistische fabrieken heeft, waar in iedere fabriek slechts één produkt wordt vervaardigd is een distributiecentrum noodzakelijk. Men probeert echter ook zoveel mogelijk distributiecentra "just-in-time" te laten functioneren. Het blijkt dat ondanks de aanwezigheid van vele distributiecentra in West-Europa er in de publieke literatuur weinig bekend is van het gedrag van distributiecentra, met name met betrekking tot het dynamisch gedrag van het distributiecentrum zelf en het dynamische gedrag van orderverzamelsystemen. Het blijkt dat beschouwingen op basis van gemiddelden zoals de gemiddelde hoeveelheid regels per order en het gemiddelde aantal orders, voor onplezierige effecten in de werkelijkheid kunnen zorgen. Een eenmaal geïnstalleerd en in gebruik genomen systeem laat zich meestal niet eenvoudig en goedkoop vervangen. Modelbouw met simulatie is dan de oplossing om toch op voorhand inzicht in het gedrag van het systeem te verkrijgen.
Binnen de sectie Automatisering van de Produktie is enige tijd geleden onderzoek gestart om de inslag, opslag en uitslag in een distributiecentrum te systematiseren. Deze systematiek wordt gebruikt om snel model-varianten op te kunnen stellen van functioneel verschillende systemen. Het blijkt op voorhand heel moeilijk te voorspellen welk distributiesysteem met bijbehorend orderverzamelsysteem optimaal is. Het is niet alleen een afweging van investeringskosten en operationele kosten. De beschikbaarheid van personeel en assortimentswijzigingen spelen hierbij een grote rol.

Opgaven

Opgave 1
Onderzoek de werking van de methode slaIn: zoals beschreven in de processor Transporteur.

Opgave 2
Onderzoek de werking van de methode controleerVoorraadEn-Bestel: in de processor InformatieBeheerder.

Opgave 3
Onderzoek de werking van de methode verwerkLevering: in de processor InformatieBeheerder.

Opgave 4
Geef aan hoe het model van het distributiecentrum kan worden gewijzigd om meer dan 3 produkten te verwerken.

Opgave 5
Bouw een model van een distributiecentrum, waarbij de InformatieBeheerder in plaats van een (B, S) -strategie zijn produkten bestelt volgens een (B, Q)-strategie [Monhemius, 1987].

Literatuur

Monhemius W., 1987,
Logistiek management,
Kluwer, Deventer.

Rooda J.E., Arentsen J.H.A., 1991,
Procescalculus 4: Modelleren van flow-produkte fabrieken,
Mechanische Technologie 1(1), 10 -20.

Rooda J.E., Arentsen J.H.A., Smit G.H., 1992,
Procescalculus 5: Modelleren van job-produktie fabrieken,
Mechanische Technologie 2(2), 36 - 45.

Rooda J.E., 1992,
Procescalculus 6: Analyse van gegevens en statistische technieken,
Mechanische Technologie 4(4), 38 - 47.