AARHUS UNIVERSITET

Concurrency Spring 2004

Mini-Project:

"Beer Factory"

Group 10	
20034435	Claus Kirkegaard Clausen
20034454	Frank Henningsen
20034441	Jens Peter Troelsen

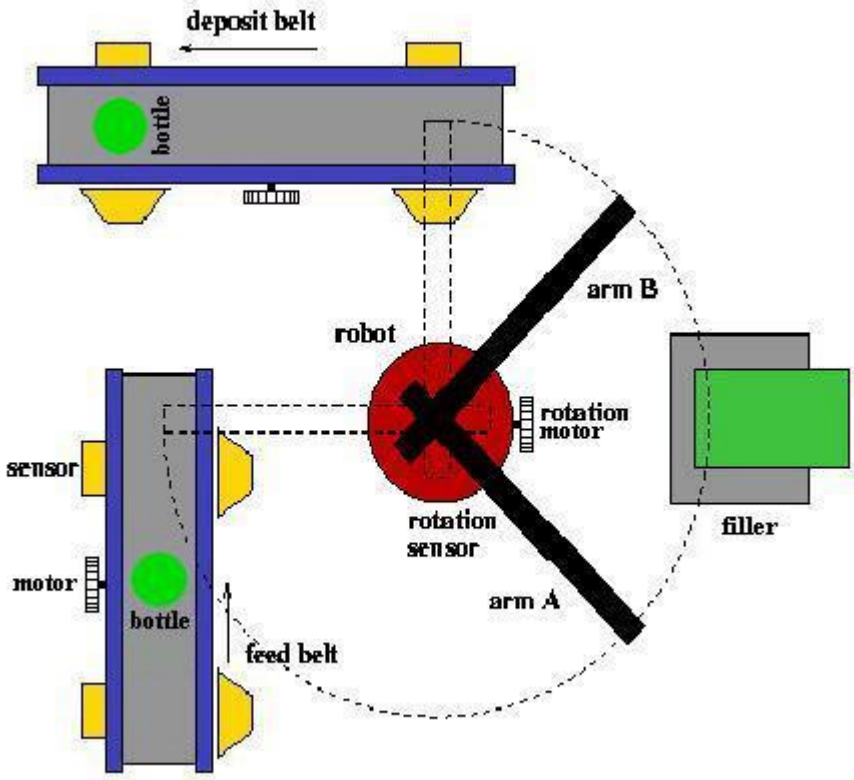
Afleveringsdato

fredag d. 21.05.2004

Indholdsfortegnelse

1 Introduktion

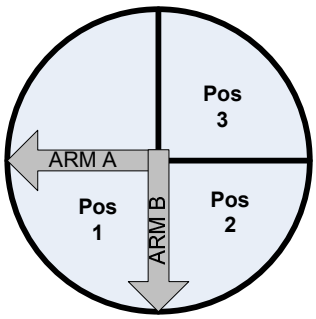
Denne opgave er den afsluttende I kurset Concurrency på Aarhus Universitet. Opgaven skal vise at der er opnået forståelse for modellering, diagrammering og programmering af samtidige systemer.



Figur 1: System oversigt over bryggeriet (afviklings miljøet)

1.1 Status og afgrænsninger

Robotten der flytter flaskerne rundt i systemet, kan bevæge sig i tre positioner, disse positioner er diagrammeret i nedenstående figur.



Figur 1 viser robotens mulige positioner.

Filleren er modeleret som et bord, hvor en flaske stilles, opfyldes og afhentes igen.

1.2 Udvidelser

N/A

2 Model

2.1 Proces Interaktion Pattern

De forskellige processer er afhængige af sensorer for at kunne skabe en sikker interaktion imellem en eller flere processer. Ved at benytte sensorer til interaktion mellem processer, gøres systemet mere modulært, og det vil være lettere at udskifte en komponent.

For at illustrere interaktions mønstret, viser vi fornedet et eksempel med robottens arm A, og feed bæltet.

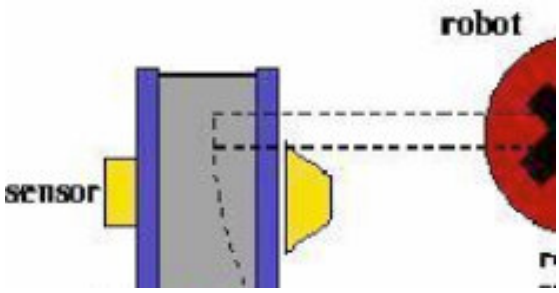


Figure 2.1: Interaktion mellem robottens arm og feed belt

Bæltet har en sensor der indikerer hvornår en flaske findes på sidste position af bæltet. For at armen ved hvornår den kan tage en flaske, skal denne også kunne aflæse den sensor.
Bæltet og armen deler en get action der kommer når en flaske er klar til at blive afhentet. De actions der optræder i de to processer hedder ikke det samme, men bliver relabled når systemet samles.
Når armen kommer hen til bæltet, vil den vente indtil der kommer en flaske den kan samle op. Armen venter indtil sensoren signalerer, at der står en flaske på bæltet. Når controlleren har registeret at der står en flaske på bæltet, bruger den armen til at tage flasken fra bæltet. Når armen tager en flaske, benytter den endnu en delt action, mellem bæltet og armen, til at signalere til bæltet at den har taget flasken, hvorefter bæltet opdaterer sin sensor og roboten roterer.

Bæltet benytter også internt de sensorer når det skal føre flaskerne frem. Så længe den sidste sensor ikke har detekteret en flaske, kan bæltet få lov til at føre flaskerne frem. Når sensoren ser en flaske, er det ikke længere muligt at føre flasker frem, og bæltet venter på at robotten tager den flaske der er klar til afhentning.

Sensorene har mulighed for at være tændte eller slukkede, samt man kan aflæse hvilken tilstand de er i. For ikke at kunne lave dirty-reads, aflæsning af en forkert værdi, er det nødventigt at bestemme hvilke actions der skal have højere prioritet. Det skal være de actions, der henholdsvis tænder og slukker sensoren (on,off).

Samme pattern er benyttet igennem hele systemet, de steder hvor der er behov for en sensor.

2.2 Processer

De processer systemet består af, vil blive beskrevet herunder. De forskellige processers formål beskrives herunder, samt eventuelle fejltilstande. De vitale dele fra FSP koden medtages, en analyse af om processen er aktiv eller passiv og hvordan den enkelte proces passer ind i det samlede system.

2.2.1 Sensor

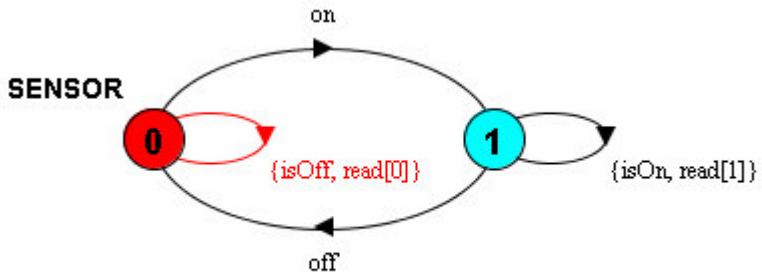
Sensorens rolle er, at fungere som monitor mellem de interagerende processer. Sensoren benytter 2 tilstande: ON og OFF, og har til hver tilstand en blokerende (isOn, isOff) og en ikke-blokerende (read) action. Ved brug af isOn og isOff, sørger sensoren for at blokere den kaldene proces, indtil sensoren skifter tilstand. Read metoden bruges af controllerne, til at aflæse sensorens nuværende tilstand.

Sensoren er modeleret i FSP på følgende måde:

Sensorer

```
SENSOR       = OFF[False],                  // Passive Monitor process
OFF[b:Bool] = (on       -> ON[True]    |   // Enter ON state
               isOff    -> OFF[b]      |   // Read sensor status blocking
               read[b] -> OFF[b]),        // Read sensor status non-blocking
ON[b:Bool]   = (off      -> OFF[False] |   // Enter OFF state
               isOn     -> ON[b]       |   // Read sensor status blocking
               read[b] -> ON[b]).         // Read sensor status non-blocking
```

Det tilsvarende LTS diagram for modellen ser ud som følgende:



Figur 2 viser sensor processens LTS diagram (0 = OFF, 1 = ON).

Da sensor processen er generel, er den samme proces anvendt af samtlige processer systemet.

Sensor processen bruges som monitor mellem:

- Producer og feed bæltet
- Feed bæltet og robotens arm A
- Robotens arm B og deposit bæltet

- deposit bæltet og Consumer

Omdøbning af sensorene bliver foretaget i de processer der anvender sensor.

2.2.2 Producer

Producerens rolle er, hele tiden at sætte flasker på feed bæltet. Feed bæltet har en sensor, som produceren konsulterer, inden han sætter en flaske på bæltet.

Produceren er i FSP modelleret på følgende måde:

Producer

```
PRODUCER = (             // active process (keeps putting bottles on feed belt)
feedbelt.sensor.isOff ->     // make sure the sensor is off before putting.
feedbelt.put ->              // put bottle on belt.
PRODUCER).                   // ready to put next bottle.
```

Produceren er en aktiv proces, og tager selv initiativ til afgøre, om den kan sætte en ny flaske på bæltet eller ej. For at afgøre om produceren kan sætte en flaske på bæltet, synkroniserer produceren med feed bæltet i den shared action `feedbelt.sensor.isOff` . Når denne action eksekveres, vil produceren blokere indtil sensoren kommer i OFF tilstand. I kraft af denne egenskab, kan produceren godt betegnes som en controller, da den er med til at sikre at fejl ikke kan forekomme.

Når produceren kombineres med bæltet, omdøbes bæltets `feedbelt.position1.isOff` action med producerens `feedbelt.sensor.isOff` action.

2.2.3 Consumer

Consumeren er den proces, der tager fyldte flasker af deposit bæltet. Consumeren ligner til forveksling produceren, da de begge interargerer med deres respektive bælte ved hjælp af en sensor. Modsat produceren venter consumeren på at deposit bæltet signalerer, at der står en fyldt øl står på bæltet, hvorefter consumerer tager flasken og drikker alt indholdet.

Processens FSP model er vist herunder:

Consumer

```
CONSUMER = (                 // active process taking bottles off deposit belt
depositbelt.sensor.isOn  ->  // make sure the sensor is on before taking bottle
         depositbelt.get ->  // get the bottle off the belt
         CONSUMER            // ready to get next bottle
).
```

Consumeren er også en aktiv proces, der er med til at afgøre om der kan tages en flaske fra bæltet. For at afgøre om consumeren kan tage en flaske fra bæltet, synkroniserer consumeren med deposit bæltet i den shared action `deposit.sensor.isOn` . Når denne action eksekveres, vil consumeren blokere indtil sensoren kommer i ON tilstand. I kraft af denne egenskab, kan consumeren godt betegnes som en controller, da den er med til at sikre at fejl ikke kan forekomme.

Når consumeren kombineres med bæltet, omdøbes bæltets `depositbelt.position3.isOn` action med consumeren `depositbelt.sensor.isOn` action.

2.2.4 Belt

Bæltets opgave er at transportere flasker langs et transport bånd med 3 positioner. Bæltet anvender 2 sensorer placeret på henholdsvis position 1 og 3, til at signalere til omverdenen at der er kommet en flaske.

En forkortet FSP model er vist herunder:

Belt

```
BELT = BELT[False][False][False],  // No bottles at any position
BELT[p1:Bool][p2:Bool][p3:Bool] = (// BELT process holding 3 positions
  put  -> if(p1)                   // Check if bottle is found at position 1
          then ERROR               // Bottle already there
          else (position1.on ...   // turn on position sensor 1
| move -> if(p3)                   // Check if bottle is found at position 3
          then ERROR               // bottle found at position 3
          . . .                    // Check remaining positions (Not shown)
| get  -> if(p3)                   // Check if bottle is found at position 3
          then (position3.off ...  // Turn of position sensor 3
          else ERROR               // No bottle found
).
```

Bælet er en passiv proces, der vha. sensorer signalerer til omverdenen. I bæltet kan 3 fejl opstå:

- En flaske placeres ovenpå en anden
- En flaske falder ud over kanten
- En flaske tages uden at findes på bæltet

Disse fejl er eksplicit markeret i FSP, ved at gå i ERROR state, og i implementeringen ved at bruge exceptions.

Når bæltet kombineres med resten af systemet, skal bæltets sensor actions {on,off} gives høj prioritet, så disse actions altid bliver eksekveres først.

2.2.5 Filler

Fillerens opgave er, at modtage en tom flaske og herefter påfylde øl. Når flasken er fuld, signaleres dette, ved at den bool variabel filleren har, sættes til true.

Processens FSP model er herunder:

Filler

```
FILLER = FILLER[False],
FILLER[bottlefilled:Bool]= (
  put  -> if(bottlefilled)
          then ERROR
          else FILLER[True]
| get  -> if(bottlefilled)
          then FILLER[False]
          else ERROR
| filled[bottlefilled] -> FILLER[bottlefilled]
)
```

Filleren er en passiv proces, og ikke udfører nogen opgaver, med mindre det bliver bedt om det af en controller.

Filleren fungerer som et bord, og har ikke nogen fill action til at fylde øl i flasken. I vores model antager vi at når en flaske stilles på bordet, bliver den automatisk fyldt. Det er muligt at spørge filleren om der står en flaske ved at bruge filled action, der fortæller om der står en flaske på bordet eller ej.

Filleren kan fejle på 2 måder:

- En proces forsøger at stille en flaske, men der står allerede en flaske på bordet.
- En proces forsøger at tage en flaske, men der står ikke nogen på bordet

For at undgå disse fejl, skal en controller konsulterere fillerens status før brug.

2.2.6 Arm

Arm processens opgave er at samle en flaske op (`grap`) og sætte en flaske ned igen (`release`). Arm processen benyttes af robot controlleren, til at bevæge flasker rundt i systemet. For at sikre at armene ikke udfører en ulovlig handling, kan controlleren benytte armens `hasBottle` action, til at spørge armen om holder en flaske eller ej.

Processens FSP model er herunder:

Arm

```
ARM = ARM[False],            // Initially not holding any bottles
```

```
ARM[b:Bool] = (              // ARM proces for holding one bottle
  grab ->    if(b)           // Check if arm holds a bottle
             then ERROR      // Arm is allready holding a bottle
             else ARM[True]  // Arm now holds a bottle
| release -> if(b)           // Check if arm holds a bottle
             then ARM[False] // Arm holds a bottle
             else ERROR      // Arm has no bottle to put
| hasBottle[b] -> ARM[b]     // Reader method for controller
).
```

Armen er en passiv proces, der styres af robot controlleren.

Armen kan fejle på 2 måder:

- En proces forsøger at tage en flaske, men armen har allerede en flaske.
- En proces forsøger at stille en flaske, men armen har ikke nogen flaske.

For at undgå disse fejl, skal en controller konsulterere armens status før brug.

Når armen kombineres med resten af systemet, omdøbes

- robotens arm A `armA.grab` action med feed bæltets `feedbelt.get` action.
- robotens arm A `armA.release` action med fillerens `filler.put` action.
- robotens arm B `armB.grab` action med fillerens `filler.get` action.
- robotens arm B `armB.release` action med deposit bæltets `depositbelt.put` action.

2.2.7 Robot

Robottens opgave, er at rotere roboten rundt i systemet, og opdatere dets 3 positions sensorer.

Et udsnit fra robottens FSP model er vist herunder:

Robot

```
ROBOT = INIT,                                // Initialize robot
INIT  = (position3.on -> ROBOT[0][0][1]),    // Start robot in position 3
ROBOT[p1:Bool][p2:Bool][p3:Bool] = (
  moveToPos1 -> if(p1)                       // Move robot to position 1
                then ERROR
                else (position3.off -> position1.on ->
                      ROBOT[True][False][False])
| moveToPos2 -> if(p2)                       // Move robot to position 2
. . . (continued in FSP)
```

Roboten er en passiv proces, der i sin simple form kun kan bevæge sig rundt mellem dets mulige positioner. Robotten har 3 positionssensorer, der fortæller hvor robotten befinder sig. Robotten kan gå i en fejltilstand hvis den forsøger at bevæge sig til en position den allerede befinder sig i.
Når robotten bevæger sig – i dette tilfælde – til position 1, sættes to rotationssensorer. Robotten bevæger sig fra position 3, som sættes til off, til position 1 som sættes til on. Sidst opdateres processens status, og returnerer til start hvor roboten er klar til at bevæge sig på ny.

Når roboten kombineres med resten af systemet, skal dets 3 positionssensor actions {on,off} gives høj prioritet, så disse actions altid bliver eksekveres først.

2.3 Controllers

I FSP'en har vi to direkte controllere samt to indirekte controllere. De direkte controllere er BELTCTRL samt ROBOTCTRL. De indirekte controllere er Produceren samt Consumeren. Controllernes opgave er, at sikre at de fejltilstande de andre processer indeholder, aldrig vil kunne forekomme.

2.3.1 Belt Controllers

Belt controllerens opgave er, at reagere på de sensorer bæltet har tilknyttet og ud fra bæltets tilstand tillade bestemte handlinger.

Processens FSP model vist er herunder:

Beltctrl

```
BELTCTRL = (
  Position1.isOn          -> BELTCTRL |
  position1.isOff -> put  -> BELTCTRL |
  position3.isOn  -> get  -> BELTCTRL |
  position3.isOff -> move -> BELTCTRL
).
```

Bælte controlleren er en aktiv proces, der sørger for at bælterne aldrig kommer i en fejl tilstand. Controlleren sikrer, at bæltet ikke kan køre flasker ud over kanten, og at der ikke kan sættes to flasker oven på hinanden, ved at benytte bæltets positionssensorer. Controlleren sørger også for at tilgang til og fra bæltet kun kan ske, når bæltets sensorer er klar.

Hvis `pos1.isOn` er aktiv, vil der stå en flaske på bæltets første plads. Det er ikke muligt at sætte en ny flaske før `pos1.isOff`. Så længe `pos3.isOff` kan bæltet flytte flaskerne frem, og når `pos3.isOn`, kan flasken fjernes af en anden proces.

Når bælte controlleren kombineres med resten af systemet, omdøbes

- Producerens `feedbelt.sensor.isOff` action med feed bæltets `feedbelt.position1.isOff` action.
- Consumerens `depositbelt.sensor.isOn` action med deposit bæltets `depositbelt.position3.isOn` action.

2.3.2 Robot Controller

Robot controllerens opgave er at transportere flasker rundt i systemet, uden at der kan opstå fejl. Roboten controllerens spiller en stor rolle i systemet, og udfører flølgende opgaver:

- Rotere robot
- Kontrollere armene
- Kontrollere tilgang til bælter
- Kontrollere tilgang til filler

For at undgå fejl tilstande, aflæser robot controlleren systemets sensorer og sørger for kun at udføre lovlige handlinger.

Et udsnit af FSP modellen er vist herunder:

Roboctrl

```
ROBOTCTRL = (
  position1.isOn -> moveToPos2 -> MOVETOPOS2
| position2.isOn -> moveToPos3 -> MOVETOPOS3
| position3.isOn -> moveToPos1 -> MOVETOPOS1
),
MOVETOPOS1 = (                                  // Move robot to position 1
  armA.hasBottle[bottleInHand:Bool] ->          // Read Arm A status
  feedbelt.position3.read[bottleReady:Bool] -> // Read feedbelt status
    if(!bottleInHand && bottleReady)            // check arm & feed belt
    then (armA.grab -> ROBOTCTRL)               // Grab the bottle
    else ROBOTCTRL                              // otherwise return control
),
MOVETOPOS2 = (                                  // Move robot to position 2
. . . (continued in FSP)
```

Robot controlleren er en aktiv proces, der konstant roterer mellem robotens tre mulige positioner og venter på, at de respektive sensorer bliver aktiveret, så den kan få armene til at flytte flaskerne.

Robotens rotation sikres, ved at konsultere robotens rotations sensor med `positionX.isOn` action, så roboten ikke kan bevæge sig til en forkert position.

Armene sikres ved altid at konsultere armenes tilstand med `armB.hasBottle` action, før man benytter armene.
Tilgang til feed bæltet sikres, ved at spørge til bæltets tilstand med feedbelt.position3.read action, mens deposit bæltet sikres via BeltCTRL processen, der sørger for at en flaske kun kan placeres på bæltet hvis depositbelt.position1.isOff action er sand. Hvis ikke der er plads på deposit bæltet, vil roboten blokere, indtil bæltet er klar.
Tilgang til filler sikres, ved at konsultere fillerens tilstand med dens filler.filled action.

Når robotten står på den sidste position, position 3, befinder arm B sig ved deposit bæltet, mens arm A befinder sig ved filleren. Controlleren sørger for, at hvis armA har en flaske, så bliver den sat på filleren, bagefter sætter armB, hvis den har en flaske, sin flaske på deposit bæltet, hvorefter den kører tilbage til start.

2.4 Struktur

Vores model af bryggeriet er bygget op af 9 processer: 4 aktive og 5 passive.

Proces	Beskrivelse
SENSOR	Passiv Monitor proces.
PRODUCER	Aktiv producer, der putter flasker på feed bæltet
CONSUMER	Aktiv proces, der tager flasker fra deposit bæltet
BELT	Passiv bælte process
BELTCTRL	Aktiv controller proces, der kontrollerer et bælte
FILLER	Passiv filler proces, modelleret som et simpelt bord.
ARM	Passiv arm proces, til at flytte flasker.
ROBOT	Passiv robot proces.
ROBOTCTRL	Aktive controller process, der kontrollerer roboten.

Til at oprette instanser af processerne, har vi lavet en række set´s:

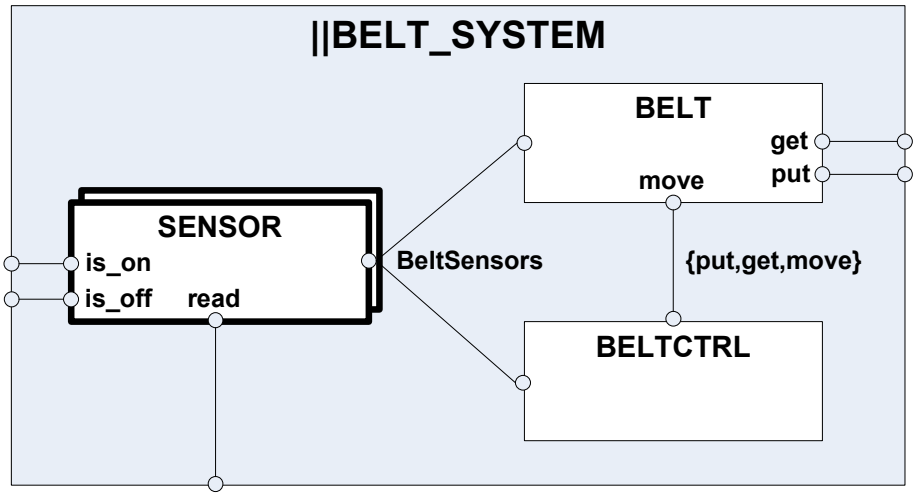
Set	Indhold
Belts	feedbelt,depositbelt
Arms	armA, armB
BeltSensors	position1,position3
RobotSensors	position1, position2, position3

De 9 processer samt set´s er fordelt på 3 sammensatte processer: BELT_SYSTEM, ROBOT_SYSTEM og BREWERY.

2.4.1 BELT_SYSTEM struktur diagram

Sammensat proces	Tilknyttede processer

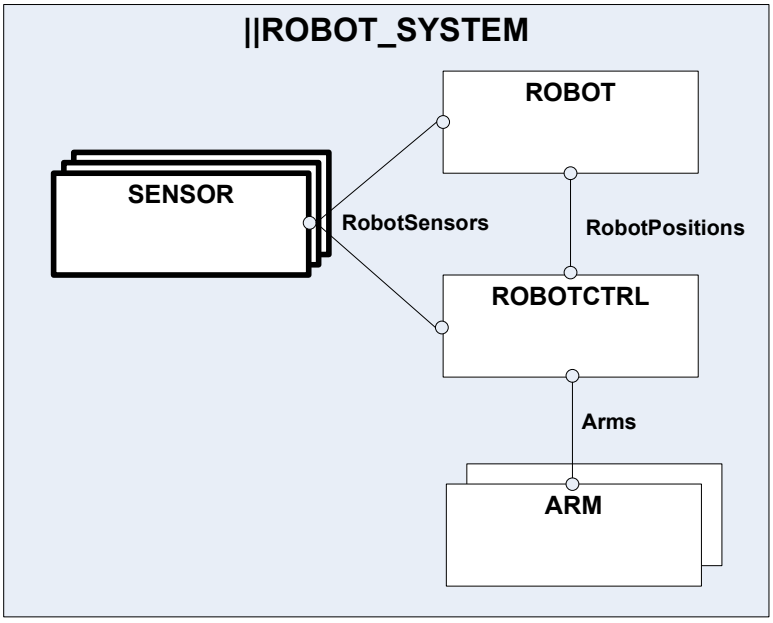
BELT_SYSTEM	BeltSensors:SENSOR || BELT || BELTCTRL || SAFE_BELT



Figur 3 viser et struktur diagram over den sammensatte BELT_SYSTEM proces.

2.4.2 ROBOT_SYSTEM struktur diagram

Sammensat proces	Tilknyttede processer
ROBOT_SYSTEM	RobotSensors:SENSOR || Arms:ARM || ROBOT || ROBOTCTRL || filler:FILLER || Arms:SAFE_ARMS

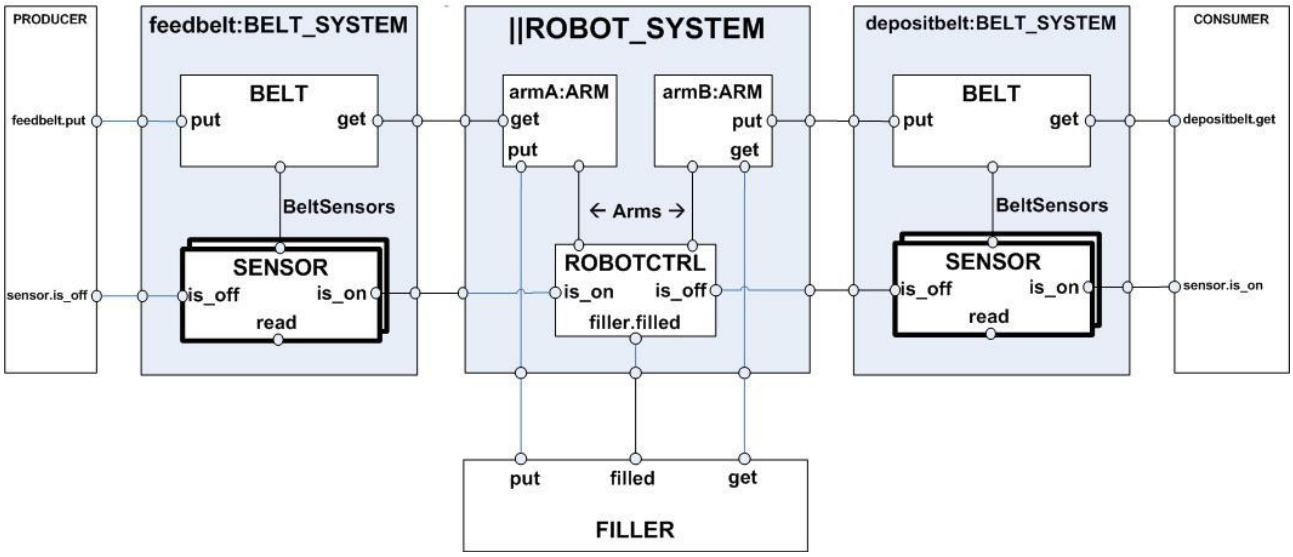


Figur 4 viser et struktur diagram over den sammensatte ROBOT_SYSTEM proces

2.4.3 BREWERY struktur diagram

Sammensat proces	Tilknyttede processer
BREWERY	PRODUCER||CONSUMER||Belts:BELT_SYSTEM||ROBOT_SYSTEM

Systemet er styret af 3 direkte controllere: feedbelt, depositbelt samt robot controller, og 2 indirekte controllere i produceren og consumeren.



Figur 5 viser et struktur diagram over hele systemet, samlet i den sammensatte proces BREWERY.

På struktur diagrammet, kan vi se de hvordan processerne interagerer med hinanden. Vi har dog valgt ikke at vise bælternes BELTCTRL på diagrammet, da det er en intern proces, der ikke interagerer med andre end bæltet selv. På samme måde er robotens ROBOTCTRL og sensorer udeladt, da de bliver behandlet internt i roboten.

2.5 Safety og Liveness Properties

Vi har valgt at lave to safety properties, og en liveness property til systemet.
Der er lavet en safety property – SAFE_BELT – til at kontrollere at flasker sat på bælterne, også bliver taget af igen. I FSP er processen modelleret som:

SAFE_BELT

```
property SAFE_BELT = (put -> SAFE_BELT[1]),
SAFE_BELT[c:1..MAX_BOTTLES] = (
  when(c<MAX_BOTTLES) put -> SAFE_BELT[c+1] |
  when(c==1)          get -> SAFE_BELT |
  when(c>1)           get -> SAFE_BELT[c-1]
).
```

En anden safety property kaldet SAFE_ARMS, er lavet for at sikre at en flaske samlet op af armen, også vil blive sat tilbage igen. I FSP er processen modelleret som:

SAFE_ARMS

```
property SAFE_ARMS = (grab -> release -> SAFE_ARMS).
```

Vores liveness, ALL_THE_WAY, garanterer at systemets producer og consumer proces vil kunne foretage deres handlinger, således at vi ved at der kan sættes flasker i systemet, og at der før eller siden bliver taget flasker fra systemet. I FSP er processen modelleret som:

ALL_THE_WAY
`progress ALL_THE_WAY = {feedbelt.put, depositbelt.get}`

2.6 Test og verificering

For at teste modellen, har vi gjort brug af de indbyggede funktioner i LTSA tool'et. Vi har fundet ud af, at vi ikke kan komme i en deadlock situation, samt at der ikke kan ske fejl. Ligeledes ses det, at hvis produceren sætter en flaske ind i systemet, så vil consumeren på et eller andet tidspunkt tage den flaske ud af systemet.

Da systemet er bygget modulært, har vi i første omgang testet de 3 sammensatte processer hver for sig, og først når disse processer fungerede, samlet dem med resten af systemet.

Desuden har vi lavet et par processer - udelukkende med det formål at teste interaktionen mellem enkelte del-moduler - for at se hvordan processerne ville opføre sig i en større kontekst. Disse integrationstest er:

Proces	**Tilknyttede processer**
BELT_TEST	PRODUCER||CONSUMER||BELT_SYSTEM
ROBOT_TEST	PRODUCER||CONSUMER||ROBOT_SYSTEM

Grunden til at producer og consumer processerne er brugt til at testen er, at testen har haft fokus på interaktionen mellem de enkelte processer. Vi var således interesserede i at finde ud af, hvordan overgangen fra en proces til en anden proces virkede.

2.6.1 Succesfuld Trace

Vi har lavet en tabel der viser den korteste vej gennem systemet. Grunden til vi mener at vores vej er den korteste, er fordi vi til hver proces har undersøgt den korteste vej igennem processen hver for sig, og senere foretaget integrationstest over flere processer. Vi har altså vha. grundig analyse over de moduler der benyttes, fundet frem til de actions der er nødventige. I tabellen er en række actions markeret med rødt, for at illustrere de shared actions der forekommer på vej gennem systemet.

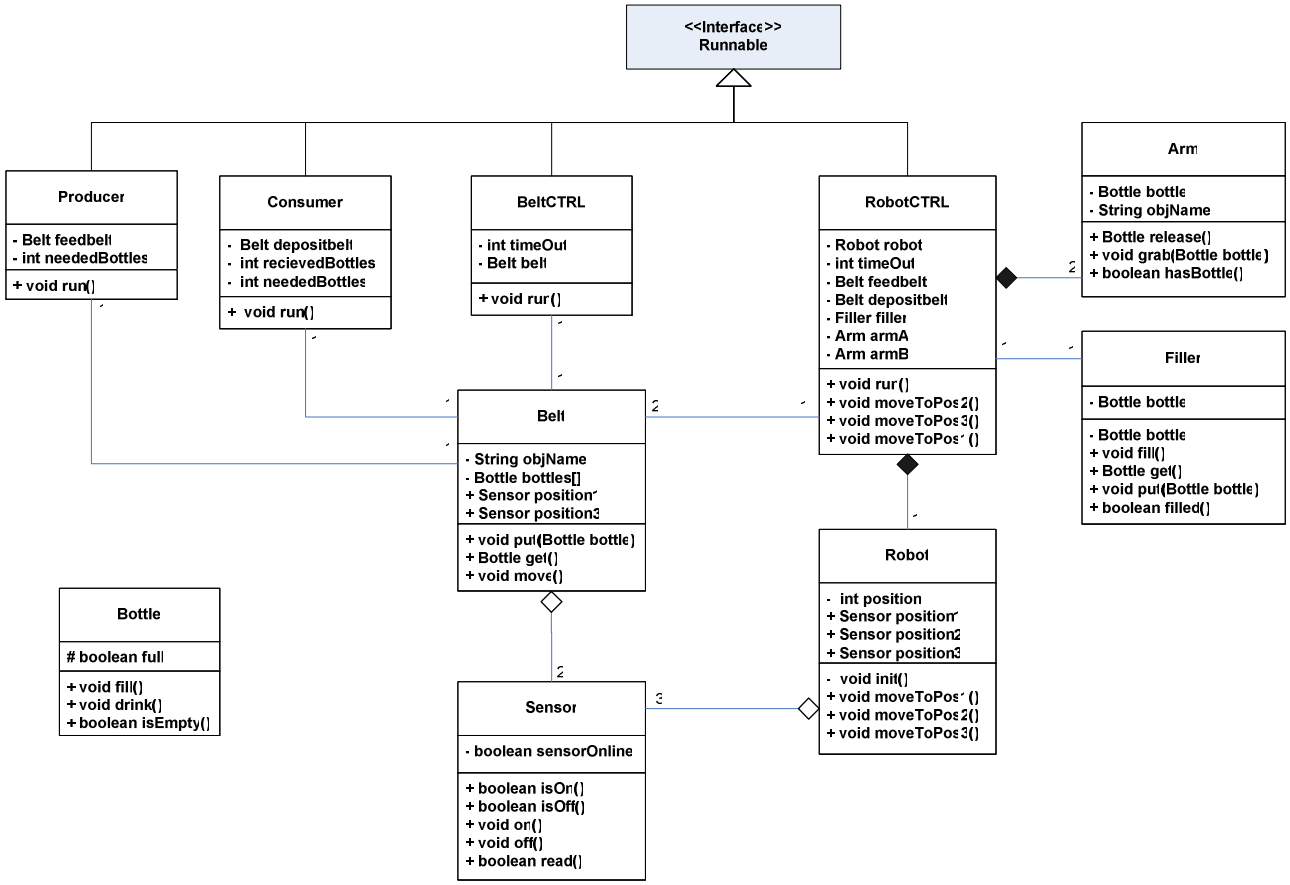
Proces	Producer	Feed belt	Robot	Filler	Deposit belt	Consumer
Actions	feedbelt.sensor.isOff feedbelt.put	feedbelt.position1.isOff feedbelt.put feedbelt.position1.on				

		feedbelt.position3.isOff feedbelt.move feedbelt.position1.off feedbelt.position3.isOff feedbelt.move feedbelt.position3.on				
			position3.on position3.isOn moveToPos1 position3.off position1.on armA.hasBottle.0 feedbelt.position3.read.1			
		feedbelt.position3.isOn feedbelt.get	armA.grab feedbelt.position3.off position1.isOn moveToPos2 position1.off position2.on armB.hasBottle.0 filler.filled.0 position2.isOn moveToPos3 position2.off position3.on armA.hasBottle.1 armB.hasBottle.0			
			armA.release position3.isOn moveToPos1 position3.off position1.on armA.hasBottle.0 feedbelt.position3.read.0 position1.isOn moveToPos2 position1.off position2.on armB.hasBottle.0 filler.filled.1	filler.put		
			armB.grab position2.isOn moveToPos3 position2.off position3.on armA.hasBottle.0 armB.hasBottle.1 depositbelt.position1.isOff	filler.get		

			armB.release		depositbelt.put depositbelt.position1.on depositbelt.position3.isOff depositbelt.move depositbelt.position1.off depositbelt.position3.isOff depositbelt.move depositbelt.position3.on depositbelt.position3.isOn depositbelt.get	 depositbelt.sensor.isOn depositbelt.get

3 Implementering

3.1 Struktur



Figur 6 viser et UML klasse diagram over hele systemet.

Vi har valgt at opbevare sensor objekterne inde i henholdsvis Belt og Robot klassen, og markeret dem som public. De klasser der ønsker at tale med en sensor, må gå igennem dets respektive klasse. F.eks. vil produceren tilgå feed bæltets sensor med et feedbelt.position1.isOff() kald.

3.2 Java implementering

3.2.1 Sensor

Sensor klassen er passiv, og fungerer som monitor mellem interagerende klasser. Sensorerne indeholder to blokerende funktioner(isOff, isOn), der er implementeret vha. '*condition synchronization*'. Nedenstående viser funktionerne on og isOn. Off og isOff er implementeret på samme måde. Desuden indeholder sensoren en ikke blokerende metode (Read), til at spørge på sensorens nuværende status, som enten kan være falsk eller sand.

```
class Sensor
boolean sensorOnline = false;
public synchronized boolean isOn() throws InterruptedException {
    while(!read()) wait();
    return read();
  }
public synchronized void on() {
    sensorOnline = true;
    notifyAll();
  }
```

3.2.2 Producer

Produceren er en aktiv proces, og implementerer derfor Runnable interfacet. Produceren er oprettet med en reference til et bælte, og en integer der fortæller hvor mange flasker produceren skal producere. På bæltet sidder to sensorer, som er public, og som produceren derfor kan tjekke om er off/on, ved hjælp af sensorernes funktioner isOn() og isOff(). Begge funktioner er blokerende.

Produceren fungerer ved at den opretter en ny instans af Bottle, venter på at feed bæltets sensor1 bliver slukket, og derefter placerer flasken på bæltet. Det gentages indtil den har oprettet så mange flasker som den har fået besked på.

Det eneste der er interessant ved produceren er run metoden.

```
class Producer implements Runnable
Belt feedbelt;
int neededBottles;
public void run() {
  int producedBottles = 0;    // Produced bottles counter
  try {
      while (producedBottles < neededBottles) {
        Bottle bottle = new Bottle();    // create a new bottle.
        feedbelt.position1.isOff();      // wait (blocking) for sensor to be off
                                         // (in LTS: feedbelt.sensor.is_off )
        feedbelt.put(bottle);            // put bottle on the conveyor.
        System.out.println("Put #" + producedBottles++);
      }
  }
  catch(InterruptedException _) {}
  catch(BottleAllreadyThereException _b) {}
}
```

3.2.3 Consumer

Consumer er en aktiv proces. Consumeren er oprettet med reference til Belt, samt en int værdi, som fortæller hvor mange flasker consumeren skal drikke i alt. Consumeren tager flasker af bæltet og drikker dem.

Consumeren fungerer på den måde, at han læser deposit bæltets position3, med sensorens blokerende læse metode, Når der står noget på bæltets position, tager han flasken og drikker den. Dette bliver han ved med, indtil han har drukket alle de flasker han er bedt om.

Run metoden er det mest interessante ved Consumeren.

class Consumer **implements Runnable**

```
Belt feedbelt;
public void run() {
  try {
    while (recievedBottles < neededBottles) {
      // wait (blocking) for sensor to be off.(in LTS: feedbelt.sensor.is_off )
      depositbelt.position3.isOn();
      Bottle bottle = depositbelt.get();  // get bottle from belt
      do {                                // and drink
        bottle.drink();
      } while(!bottle.isEmpty());
    }
  }
  catch(InterruptedException _) {}
  catch(NoBottleFoundException _) {}
  System.exit(0);  //End program. All bottles received.
}
```

3.2.4 Belt

Bæltet er en passiv proces, der vha. BeltCTRL bevæger transport båndet fremad. Bæltet er implementeret ligesom modellen, men i stedet for at gå i error tilstanden, bliver der i java kastet en exception. Bæltet er altså ikke sikker, og der kastes en exeption, hvis der udføres en ulovlig handling. Herunder er move metoden vist, som illustrerer hvorledes bæltet bliver bevæget.

class Belt

```
public Sensor position1;
public Sensor position3;
```

```
public synchronized void move() throws BottleDroppedException {
    if(bottles[2] != null) {
      throw new BottleDroppedException();
    }
    position1.off();
    if(bottles[1]!= null){
      position3.on();
    }
    bottles[2] = bottles[1];
    bottles[1] = bottles[0];
    bottles[0] = null;
}
```

3.2.5 BeltCTRL

BeltCTRL er en aktiv proces, hvis eneste opgave er at bevæge et bælte. Det er BeltCTRL's ansvar at der ikke ryger nogen flakser på gulvet.
BeltCTRL bliver oprettet med en reference til en Belt klasse, samt en int parameter, som fortæller hvor lang tid der skal være imellem at den flytter bæltet. Det mest interessante ved BeltCTRL er dens run metode, som er vist herunder.

```
                          class BeltCTRL implements Runnable
Belt belt
Public void run() {
    while(true){ // Run forever
      try {
        belt.position3.isOff();     // Check belt position sensor 3(Blocking)
        belt.move();                // No bottle found at position 3, so move belt
        Thread.currentThread().sleep(timeOut); // Wait some time until next move

      } catch (InterruptedException _) {}
        catch (BottleDroppedException _) {}
    }
}
```

3.2.6 Arm

Arm er en passiv klasse, der kan holde og sætte en flaske. Armen bliver oprettet med et navn, som bruges til at identificere armen, om det er arm A eller B. Armen har ligesom modellen mulighed for at sende en exception, hvis armen forsøger at tage en flaske men allerede holder en flaske.
For at undgå fejl, kontrolleres armen af RobotCTRL´eren, der kan aflæse armens status med hasBottle() funktionen.

Herunder er der vist et eksempel på en af Robottens move metode, hvor der flyttes fra en position til en anden, og de respektive sensorer bliver opdateret.

```
class Arm
private String objName;     // Arm identifier
  public void grab(Bottle bottle) throws BottleAllreadyThereException {
    if(hasBottle()) { // Check that the arm does not hold a bottle
      throw new BottleAllreadyThereException();
    }
    this.bottle = bottle;
    if(objName.compareTo("A") == 0)  // Arm A
    else                             // Arm B
  }
```

3.2.7 Robot

Robotten er en passiv proces. Robotten bruger tre sensorer, til at fortælle hvilken position den befinder sig i. Det er så op til controlleren at flytte roboten, ud fra disse sensorer. Når robotten flyttes medfører det at dens sensorer tændes og slukkes. Robottens sensorer er public, så de kan aflæses af RobotCTRL. Robotten kan kun være placeret på en af de tre lovlige positioner, hvor dens arme kan tage og sætte flasker. Det vil sige, at den ikke kan placere sig mellem to sensorer, og f.eks. tabe en flaske.

Herunder er der vist et eksempel på en af Robottens move metode, hvor der flyttes fra en position til en anden, og de respektive sensorer bliver opdateret.

```
class Robot
private int position;     // current robot position
public Sensor position1;
public Sensor position2;
public Sensor position3;
public void moveToPos1() throws RobotAllreadyThereException {
    if( position == POSITION1) { // Check if robot is allready at position 1
      throw new RobotAllreadyThereException();
    }
    position3.off();      // Turn off sensor 3
    position1.on();       // Turn on sensor 1
    position = POSITION1; // Update robot process
}
```

3.2.8 RobotCTRL

RobotCTRL er en aktiv proces. Dens ansvar, er at sørge for at robotten ikke kommer i en fejl tilstand, samt sørge for at flaskerne bliver flyttet sikkert fra bælte til bælte. RobotCTRL bliver oprettet med referencer til to bælter, en filler, samt en robot. RobotCTRL opretter selv to objekter af Arm klassen.

Det mest interessante ved RobotCTRL er run metoden.

class RobotCTRL **implements Runnable**

```
Robot robot;
Belt feedbelt, depositbelt;
Filler filler;
Arm armA, armB;
```

```
public void run() {
    while(true){ // Run forever
      try {
        if(robot.position1.isOn()) { // Check that position is at position 1
          moveToPos2();              // Move robot to position 2
        }
        if(robot.position2.isOn()) { // Check that position is at position 2
          moveToPos3();              // Move robot to position 3
       }
       if(robot.position3.isOn()) {  // Check that position is at position 3
          moveToPos1();              // Move robot to position 1
       }
       Thread.currentThread().sleep(timeOut); // Wait some time until robot rotates
     }
     catch ( RobotAllreadyThereException _) {}
     catch ( BottleAllreadyThereException _) {}
     catch ( NoBottleFoundException _) {}
     catch ( InterruptedException _) {}
   }
}
```

RobotCTRL sørger for altid at gå samme vej gennem systemet. Dvs. den vil altid starte med at gå til position 1 og udføre sine opgaver, hvorefter den går til position 2 og udfører sine opgaver, og til sidst position 3 og udfører sine opgaver, hvorefter den starter forfra.

3.2.9 Filler

På grund af tidspres er filleren implementeret som et bord. Det vil sige at der kan sættes en flaske på den, hvorefter den vil blive fyldt op. Der er dog stadig kun plads til en flaske af gangen på filleren.

4 Konklusion

4.1 Forbedringer

En forbedring der ville kunne gøre systemet mere overskuelig og brugervenligt, var at lave en grafisk bruger grænseflade vha. swing.

Filleren burder være mere intelligent. Der kunne være et par sensorer til at kontrollere om der fandtes en flaske i filleren, samt en anden til at informere om filileren har fyldt flasken eller ej, dette vil også betyde at der skulle tilføjes en controller til at styre fillerens handlinger.

Robot controlleren kunne forbedres på mange måder, f.eks. med flere arme og flere positioner at bevæge sig i. Man kunne ligeledes forestille sig, at robotten kunne flytte mere end en flaske ad gangen – så skulle filleren naturligvis også kunne påfylde flere flasker ad gangen.

Et generelt problem med robot controlleren, er at den har for mange opgaver, vi kunne have lavet en transportør proces, til at styre og kontrollere en flaskes forløb gennem systemet, og derved aflaste robot controlleren for nogle af dens opgaver.

4.2 Afsluttende bemærkninger

Vi har i dette projekt, for alvor arbejdet med de principer og værktøjer vi har lært gennem kurset. Vi har lært hvordan man kan modellere samtidige processerer, og afprøve modellerne før de implementeres. Ved at lave modellen før koden, undgår man at skulle fejlfinde i produktionskoden, hvor fejlen kan være svær at finde. Ved at opbygge en model, kan man afprøve, analysere, forbedre og teste den egentlige system funktionalitet, uden indblanding af andre – i samtidighed henseende – ligegyldige ting.

Når modellen så er lavet, og opfylder de krav man har til systemet, kan koden nærmest implementeres med hoved på hylden, da man allerede har gjort sig de nødventige tanker for, hvordan opbygningen skal være.

At man kan benytte et tool til at teste sin model og finde deadlocks inden man står med den komplette implementation, vil i fremtiden være en yderst brugbar hjælp.