Rasmus Ebdrup Sørensen
Simon Bognolo

Timed PyCSP

maj 2010
Datalogisk Institut, Københavns Universitet

Abstract

Over the last years, multi-core cpu's have become increasingly more common, which has lead to a demand for easy representation of concurrency in application development. This has increased the popularity of CSP, leading to implementations in several common programming languages.

Most practical representations of time in CSP currently breaks with the CSP paradigm, so in this thesis we will explore the possibilities of including a representation of time directly in an implementation of CSP. The goal is to make one or several representations of time that gives a developer the tools needed to handle problems within a timecentric domain. We limit ourselves to uses of time within discrete event simulation, real time planning and interactive planning. For each use we approach the problem using applicable examples to identify key issues and requirements.

We have found that representations of time in applications are not bound to their problem domain, but rather the time model they apply. As such we have developed two representations of time, one using discrete time, the other using real time.

Our solution provides developers with intuitive and flexible representations of time, allowing them to focus more on the actual problem, and less on representation and management of time.

The implementation can be found at the address given below. The code specific for this thesis is located in `/pycsp/simulation` and `/pycsp/deadline`.
`http://code.google.com/p/pycsp/source/browse/#svn/branches/TimedPyCSP`

The examples used in the thesis are located at the address given below, and can also be found in the appendix.
`http://github.com/shamran/TimedPyCSP/tree/master/projects`

Indholdsfortegnelse

Kapitel 1

Introduktion

Vi vil indledningsvis præsentere konteksten for dette speciale. Herefter klarlægger vi, hvilke problemer vi ønsker at berøre samt hvad vores tilgangsvinkel er. Afslutningsvis giver vi et overblik over specialets struktur.

1.1 Kontekst

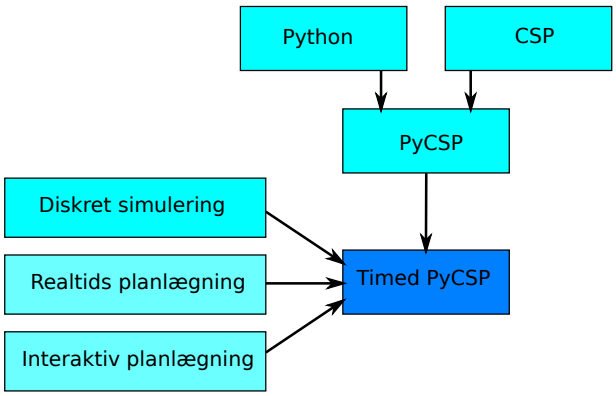
Over de sidste par år er multi-kerne cpu'er blevet hyldevarer, hvilket har afledt et stigende behov for at udvikle programmer, der kan udnytte flere kerner samtidigt. Dette behov har gjort CSP til et populært sprog, da det gør det let at repræsentere samtidighed og desuden kræver eksplicit udveksling af data frem for at benytte delte datastrukturer, som kræver låsemekanismer eller anden form for kontrol over hvem der tilgår og hvordan det delte data tilgås. CSP's stigende popularitet har affødt at det er blevet blevet implementeret i flere andre programmeringssprog, og senest har Google lavet sproget Go, der er baseret på CSP.

Tid har altid været et brugbart værktøj indenfor datalogi, men har ofte været besværligt at repræsentere og håndtere. Det har ført til megen forskning og udvikling indenfor området, og har resulteret i adskillige modeller og frameworks. I den forbindelse er der også lavet en model for tid i CSP, kaldet TimedCSP. Dette er hovedsageligt et teoretisk arbejde som aldrig har vundet indpas i nogle af de gængse implementering af CSP. Der er derfor, så vidt vi ved, på nuværende tidspunkt ikke nogen praktisk anvendt implementering af tid i CSP.

1.2 Specialets problemformulering og struktur

Set i lyset af den nuværende mangel på en praktisk anvendelig implementering af tid i CSP, vil vi undersøge om det er muligt at lave en sådan - dvs. en implementering, som kan bruges af udviklere til at løse problemer, hvori tid indgår.

For at opnå dette vil vi undersøge, hvad der skal til for at introducere følgende tre anvendelsesområder i PyCSP: Diskret simulering, realtids planlægning og interaktiv planlægning. Disse anvendelsesområder repræsenterer områder hvor tid indgår og dækker tilsammen bredt over tid som helhed. Diskret simulering anvendes i stor udstrækning til simulering af komplekse systemer, hvor man ikke på beregne systemets karakteristika. Realtids planlægning benyttes i tidskritiske systemer hvor der er stringente krav om at en given begivenhed er blevet udført inden for en tidsramme. Endeligt bruges interaktiv planlægning i spilindustrien til bl.a. at udregne den visuelle scene i et computerspil. På figur 1.1 på denne side viser vi hvordan vi forventer at kombinere anvendelsområderne med PyCSP for derved at komme frem til vores Timed PyCSP.



Figur 1.1: Samspil mellem CSP, Python og de tre anvendelsområder af tid samlet i Timed PyCSP.

For hver model vil vi definere en række eksempler, der illustrerer disse anvendelsesområderne. Eksemplerne skal sikre den praktiske anvendelighed og senere bruges til at vise, hvordan et tidsspecifikt problem kan løses henholdsvis med og uden vores udvidelse. Eksemplernes formål er altså, at give et klart indblik i de krav, der stilles til en udvidelse af PyCSP, og hvilke fordele en introduktion af de givne anvendelsesområderne i PyCSP vil give. På denne baggrund vil vi komme med løsningsforslag som tager udgangspunkt i den praktiske anvendelighed. Disse løsningsforslag vil såfremt det er muligt, blive implementeret som en udvidelse af PyCSP.

Specialet vil derfor være struktureret som følger. I kapitel 2 vil vi gennemgå CSP og PyCSP med fokus på de dele der er relevante i forhold til at introducere tid. I kapitlerne 3, 4 og 5 vil vi gennemgå de tre anvendelsesområder som beskrevet ovenfor. Afslutningsvis vil vi foretage en samlet i evaluering og konklusion i kapitel 6.

1.3 Termer

I forbindelse med dette speciale vil vi bruge følgende termer:

Skemaplanlægger dækker over det engelske ord Scheduler. En skemaplanlægger er det software der står for at planlægge i hvilken rækkefølge processerne skal udføres på computeren.

Realtid er en tidsmodel, der i litteraturen også er defineret som absolut tid, eller Newtonisk tid. Tiden ses som en fundamental struktur i universet, der fremskrives kontinuerligt og uafhængigt af nogle eksterne kræfter.

Diskret tid er en anden tidsmodel. Her samples værdier fra den kontinuerlige tid, således at tiden fremskrives i ryk. Den enkelte sample er normal taget med et konstant tidsinterval, men kan også være taget med et variabelt tidsinterval. I diskret tid kan tiden enten drives frem af tiden selv, som i realtid, eller fremskrives manuelt. I dette speciale defineres diskret tid til at have variable tidsintervaller, og skal fremskrives manuelt.

Anvendelsesområde er en konkret implementering af en tidsmodel.

`Skrivemaskine-font` markerer i dette speciale variabelnavne, funktioner, klasser og moduler som findes i koden.

Kapitel 2

CSP og PyCSP

Vi vil i dette kapitel give en kort introduktion til CSP, mere specifikt implementeringen af CSP i Python kaldet PyCSP. Vi vil ikke gennemgå hverken CSP eller PyCSP i deres helhed, men kun gennemgå de dele af PyCSP, der er relevante i fht. introduktionen af tid.

CSP er et sprog til at beskrive uafhængige processer, der udelukkende udveksler information ved at sende og modtage beskeder over eksplicitte kanaler. Det blev introduceret af Hoare i „Communicating sequential processes“[13] og ligger til grund for adskillige praktiske implementeringer i et udvalg af programmeringssprog heriblandt Occam, Java, C++ og Python[3, 4, 17, 21]. En implementeringen i Python hedder PyCSP og er udviklet i et samarbejde mellem Tromsø Universitet og Københavns Universitet med henblik på at kombinere Pythons muligheder for effektivt og hurtigt at udvikle programmer med CSPs evne til at udtrykke samtidighed[3].

2.1 Kommunikation mellem processer

I CSP sker al kommunikation over kanaler. En kanal har to eller flere kanalender, som processer kan kobles til. En proces vælger på forhånd, om den vil læse eller skrive til en kanal, ved at koble sig på den respektive kanalende. De nævnte implementeringer har forskellige typer kanaler, der adskiller sig ved, hvordan de forbindes til andre processer. Generelt findes der fire typer kanaler: one-to-one, any-to-one, one-to-any og any-to-any. Forskellen ligger i hvem og hvor mange, der kan læse og skrive til en kanal. Det er klart, at any-to-any kanalen er den mest generelle og har funktionalitet som de andre. Dette er udnyttet i bl.a. PyCSP hvor det er den eneste kanaltype der er til rådighed. Fælles for de fire kanaltyper er, at processen ikke kender til kanalen og hvilken proces, der er i den anden ende af kanalen. I stedet kender processen kun til en kanalende, som den enten kan

skrive til eller læse fra. Hvis to processer både skal kunne læse fra og skrive til hinanden, skal der bruges to kanaler med tilhørende kanalender.

Kommunikation mellem to processer i CSP kan kun ske, når begge processer er klar til at kommunikere. Hvis den ene proces er klar før den anden, er den nødt til at vente. Når de begge er klar kan de kommunikere og så fortsætte.

En `alternation` er en struktur til at foretage et prioriteret valg omkring kommunikation. En `alternation` kan indeholde et vilkårligt antal kanalender, men der kommunikeres kun på en af dem. Ligesom ved almindelig kommunikation i CSP skal begge kanalender i en kanal være klar, før der kommunikeres. Det er en `guard` der styrer, hvilke kanalender, der indgår i kommunikation i en `alternation`. Der kan tilknyttes to særlige guards, `SKIP-` og `timeout-guard`. En `SKIP-guard` er altid klar og giver mulighed for konstruktioner som f.eks. "kommuniker, hvis der er processer, der ønsker at kommunikere med os, ellers fortsæt". En `timeout-guard` er i modsætning til en `SKIP-guard` ikke funderet i CSP, men er lavet udfra et pragmatisk synspunkt. Den er klar efter et angivent tidspunkt og giver mulighed for konstruktioner som f.eks "kommuniker med en af de ønskede processer indenfor en tidsgrænse, ellers fortsæt".

2.2 Tre implementeringer

I artiklen „Three Unique Implementations of Processes for PyCSP“[11] præsenteres tre forskellige implementeringer af processer i PyCSP. Hver implementering har fordele og ulemper, og formålet med at have tre til rådighed er at give udvikleren mulighed for at vælge den implementering der passer bedst til en given applikation. Implementeringerne deler samme API, så det er let at skifte rundt mellem dem og teste samspillet med den udviklede applikation.

De tre versioner adskiller sig kun ved den måde CSP-processerne internt er implementeret, og derfor også ved hvilke egenskaber, der er gældende for dem. De benytter henholdsvis operativsystemets tråde (threads), multiprocesseringsmodulet (processes) og brugertråde (greenlet) i Python. En applikation i threadsversionen vil typisk være begrænset til ca. 1000 samtidige tråde[11, s. 3]. Man ville umidelbart forvente at denne version er i stand til at udnytte flere kerner i processoren. Dette er dog ikke tilfældet, fordi Pythons Global Intepreter Lock (GIL) kun tillader, at en operation udføres ad gangen. Dette kan til dels omgås ved at flytte beregningstunge dele af programmet ud i eksterne moduler udviklet i et mere effektivt sprog som f.eks. C. Disse vil ikke være begrænset af GIL'en og kan derfor køres på andre kerner.

Ønsker man udnyttelse af flere kerner direkte i PyCSP, kan man i stedet benyt-

te processes-versionen. Denne version udnytter Pythons multiprocessing, der blev introduceret i Python version 2.6, og fungerer ved, at der oprettes en GIL for hver proces, hvorved problemet fra thread-versionen omgås. Ulempen er, at det bliver mere ressourcekrævende og dyrt at foretage kontekstskift. Den sidste mulighed er Greenlets-versionen, der går den anden vej og implementerer CSP-processer som user-level-tråde i form af greenlet-modulet til Python[12]. Herved kan man have utroligt mange samtidige tråde, der er hurtige at skifte mellem, og som er meget lidt ressourcekrævende. Dette gør denne version velegnet til applikationer, som består af mange CSP-processer. Fordi hele programmet i greenlets-versionen kun består af en tråd, som de forskellige user-level tråde dele, vil der ikke kunne benyttes flere kerner. Til håndtering af kørslen af de enkelte user-level-tråde er der i denne version introduceret en skemaplanlægger, der også har sin egen user-level-tråd, og som bla. står for at udvælge hvilken CSP-proces der skal aktivere. Fordi skemaplanlæggeren også er en user-level-tråd, vil den ikke kunne foretage preemptive kontekstskift mellem CSP-processerne, men de skal selv sørge for at afgive kontrollen, når de ikke længere kan foretage et arbejde, så skemaplanlæggeren kan aktivere en ny CSP-proces.

2.3 Introduktion af tid i de tre implementeringer

Vi vil foretrække en implementering af tid, der kan dækker over alle tre versioner. Herved kan vi følge ideen i PyCSP med at benytte samme API på tværs af implementeringerne.

Når vi ønsker at introducere tid i PyCSP, heriblandt tidsmæssige begrænsninger, skal vi kunne styre afviklingen af CSP-processer. I `threads`- og processes-versionerne kan vi ikke styre, hvilke af de tråde, der eksekverer CSP, der er aktive, men kun styre hvad en tråd gør, når den bliver aktiveret. Dette skyldes, at kontrollen ligger i operativsystemets skemaplanlægger, som vi som udgangspunkt ikke kan ændre i. At introducere tid i disse versioner kræver en væsentlig mængde kommunikation mellem trådene til styring af, hvad der må afvikles hvornår. Greenlets-versionen har som sagt sin egen skemaplanlægger, da den er baseret på user-level tråde, og vi kan derfor styre afviklingen af CSP-processerne direkte. Vi har derfor valgt at begrænse os til kun at lave en implementering af tid i greenlets-versionen. Som følge heraf vil alle fremtidige referencer til PyCSP være med henblik på greenlets-versionen med mindre andet er nævnt. Ligeledes vil CSP-processer blot blive kaldt processer.

2.4 Afvikling af processer i `greenlet`

På baggrund af valget af greenlets-versionen, vil vi kort beskrive denne versions afvikling af processer da den har sin egen skemaplanlægger, og afvikling af processer derfor er anderledes end i de to andre versioner, og dette vil være relevant at kende metoderne til at afvikle processerne i forbindelse med udvidelsen af greenlets-versionen i de kommende kapitler. I PyCSP findes der tre metoder til at afvikle processer, `Parallel`, `Spawn` og `Sequence`. `Parallel`-kaldet benyttes til samtidig afvikling af processer ved at lægge de angivne processer på skemaplanlæggeren og igangsætte eksekvering af dem. `Spawn` er meget lig `Parallel` med den forskel, at den ikke igangsætter eksekveringen af processer. Dette skal håndteres af et foregående eller efterfølgende kald til `Parallel`. `Sequence`-kaldet benyttes, når man ønsker at afvikle processer sekventielt. Det bruger ikke skemaplanlæggeren, men gennemløber blot processerne sekventielt.

Kapitel 3

Simulering i diskret tid

Det første anvendelsesområde vi vil kigge på, er simulering i diskret tid. Vi vil i dette kapitel gennemgå, hvad simuleringer i diskret tid kan bruges til, samt redegøre for hvilke problemstillinger, vi skal tage højde for, for at kunne implementere diskret simulering i PyCSP.

Overordnet set har simuleringer længe været et værdifuldt værktøj til at klarlægge, hvordan et system fungerer og er specielt brugbart til at repræsentere systemer, hvis tilstand ændres over tid, eller hvis der er interaktion mellem flere systemer. Typiske områder, hvor simulering ofte bruges, fremgår af tabel 3.1.

Inden for simulering ønsker vi at fokusere på diskret simulering, der i litteraturen ofte kaldes discrete event simulation (DES), for at vise at fokus ligger på simuleringen af begivenheder. DES adskiller sig fra andre simuleringsmodeller i den måde tid opfattes på. I DES anskues tid som diskrete tidsskridt uden kobling til realtid. I disse tidsskridt udføres en eller flere begivenheder, som hver især ændrer på systemets tilstand. Når alle begivenheder for et tidsskridt er udført, kan tiden tælles op, og begivenheder for det næste tidsskridt kan udføres. Der kan være fastlagt et vilkårligt antal begivenheder til et tidsskridt, og hver begivenhed kan variere vilkårligt ifht., hvor langt tid den tager at udføre i realtid. Dermed er der ikke nogen kobling mellem den diskrete tid og realtid, og et diskret tidsskridt kan variere vilkårligt i realtid. Begivenhederne, der skal udføres af systemet, kan enten være givet på forhånd eller blive skemaplanlagt dynamisk under afviklingen af andre begivenheder. Afhængig af hvad der simuleres, kan man udtrække relevant information om systemet f.eks. gennemsnitlig behandlingstid for elementer, længden af køer i systemet eller den samlede aktivitetstid for hvert delelement i systemet. For at kunne konstruere en DES skal vi have følgende til rådighed:

1. En repræsentation af tid til at styre, hvornår vi skifter tidsskridt.
2. En liste over begivenheder, der skal udføres i hvert tidsskridt.

General Situation	Examples
Real system does not yet exist and building a prototype is cost prohibitive, time-consuming or hazardous.	Aircraft, Production System, Nuclear Reactor
System is impossible to build.	National Economy, Biological System
Real system exists but experimentation is too expensive, hazardous or disruptive to conduct.	Proposed Changes to a Materials Handling System, Military Unit, Transportation System, Airport Baggage Handling System
Forecasting is required to analyze long time periods in a compressed format.	Population Growth, Forest Fire Spread, Urbanization Studies, Pandemic Flu Spread
Mathematical modeling has no practical analytical or numeric solution.	Stochastic Problems, Nonlinear Differential Equations

Tabel 3.1: Anvendelsesområder for simuleringer. Tabellen er kopieret fra [18, s. 10]

3. Mulighed for at opsamle statistisk data fra simuleringen.

En repræsentation af diskret tid skal kunne udtrykke den aktuelle tid, endelig fremtidig tid og gerne kunne foretage en variabel fremskrivning af tiden. Eksempelvis kan man forestille sig, at tiden er defineret som er række diskrete tidsskridt $t_0, t_1 \ldots t_n$. I tiden t_0 planlægges en begivenhed til tiden t_3. Såfremt der ikke er planlagt andre begivenheder, er det her en fordel at kunne springe over t_1 og t_2 og fremskrive tiden direkte til t_3.

Listen over begivenhederne, der skal udføres, skal repræsenteres, så vi dels kan bestemme den næste begivenhed, der skal udføres, dels har mulighed for indsættelse af nye begivenheder i listen.

Hvilke informationer, der er interessante at lave statistik over, vil være meget

afhængig af den enkelte applikation. Opsamlingen bør derfor foretages i den enkelte udviklede simulering og ikke i vores simuleringssprog. Vi skal derfor muliggøre opsamlingen med nogle værktøjer, som udviklere kan benytte. Disse værktøjer skal være generiske, således de kan bruges på tværs af simuleringsdomæner.

PDES

Når vi arbejder med CSP, hvis styrke bl.a. er samtidighed, er det oplagt at afsøge hvilket arbejde, der er lavet med henblik på samtidighed og paralellitet i DES. Dette forskningsområde kaldes parallel discrete event simulation(PDES). Formålet med PDES er en parallelisering af simuleringen for at kunne udnytte flere cpu'er, eller flere maskiner. Dette introducerer dog flere problemstillinger relateret til håndtering af tid, da tiden er nødt til at være lokal i stedet for global, og dermed kan varierer på tværs af simuleringen. Dette giver problemer som f.eks. modtagelse af beskeder fra fortiden, der kan tvinge en enkelt maskine til at rulle sin del af simuleringen tilbage til et tidligere tidspunkt.

Periode	DES	PDES
1970 til 1980	296	2
1980 til 1990	1.460	95
1990 til 2000	6.190	1.260
2000 til 2010	13.100	1.210

Tabel 3.2: Publisering af artikler if. google scholar ved søgning på hhv. "discrete event simulation" og "parallel discrete event simulation"

PDES har ikke vundet stort indpas i den videnskabelige verden, som man kan se af tabel 3.2 på denne side. En grund til dette er, at når tiden kan køre parallelt, øges omkostningerne ved at administrere den mere komplekse tidsrepræsentation, hvilket resulterer i lavere hastighed, end når tiden kan holdes synkront på tværs af processerne. Grundet den forholdsvis dårlige ydelse og deraf følgende manglende interesse, har vi også valgt at lade PDES ligge, og fokusere udelukkende på almindelig DES indenfor simulering i diskret tid.

Barrierer

I DES er tiden den samme for alle processer, og de skal derfor have en fælles tid, der fremskrives samtidigt for alle processerne. En global viden som tid kræver derfor synkronisering af alle processerne, og til denne koordinering og synkronisering af flere processer er den mest brugte metode at introducere en barriere. Barrierer blev først introduceret i MPI [10], hvor den bruges til at sikre, at alle processer venter i barrieren før de fortsætter.

kodeuddrag 3.1: En barriere i PyCSP

```
@proces
def Barrier(nprocesses, signalIN, signalOUT):
  while True:
    for i in range (nprocesses):
      signalIN()
    for i in range (nprocesses):
      signalOUT(0)
```

I CSP kan man lave sin egen barriere ved at udnytte, at begge kanalender skal være klar, før der kan kommunikeres, og at en proces, der indgår i en kommunikation, derfor vil vente indtil den anden ende er klar. Ved hjælp af kanaler kan man derfor lave en simpel barriere ved brug af kommunikation over kanaler. En implementering af den simple barriere som en selvstændig proces kan ses i kodeuddrag 3.1 på denne side.

Barrierer er en meget effektiv metode til at synkronisere processer, der kører parallelt, og er brugt flittigt i MPI. I CSP findes der dog en konflikt i brugen af barrierer, selvom det er nemt at implementere. Problemet er at hver proces bør fungere i isolation, og den eneste interaktion mellem processerne er via kanalerne. Derfor virker introduktionen af barrierer og kald til disse kunstig i CSP.

Barnes, Welch og Sampson beskriver begrundelsen for brugen af barrierer:

> *"[...] where the barriers may be used to maintain global and/or localised models of time and to synchronise safe access to shared data[...]"* [2, p. 1]

Barrierens berettigelse i CSP er derfor at introducere tid og at kunne bruge delt data. I CSP bør der ikke være delt data mellem processerne, men derimod kun lokalt data. Hvis data er delt pga. den arkitektur CSP er implementeret på, bør dette abstraheres væk og udnyttes internt i kanalerne. At introducere hjælpemidler for at styre delt data, er derfor at tilskynde til en forkert brug af CSP. Tiden er den anden begrundelse for at benytte barrierer. Barrierer giver dog kun en primitiv model for tid, og vi vil vise, at med brugen af DES får man et stærkere værktøj, der blandt meget andet også kan erstatte brugen af barrierer.

3.1 Eksempler

Som eksempler på programmer der kan benytte DES har vi valgt to eksempler. Vi har brugt "Hajer og fisk på Wa-Tor" der er af typen continous simulations, og

"kunder i en bank", som er et klassisk eksempel inden for DES.

3.1.1 Hajer og fisk på Wa-Tor

Vores første eksempel tager udgangspunkt i et Lotka-Volterra predator-prey scenarie, som Dewdney har beskrevet i artiklen „Sharks and fish wage an ecological war on the toroidal planet Wa-Tor"[8]. Vi har valgt dette eksempel, da det indeholder en global verden, hvor en implementering i CSP, umiddelbart vil kræve en synkronisering af flere processer. Til at foretage denne synkronisering virker det oplagt at introducere barrierer, og vi vil med eksemplet vise om vores løsning kan erstatte brugen af disse barrierer. Desuden vil eksemplet vise om vores løsning kan bruges til simuleringer af typen continous simulations.

Artiklen beskriver den fiktive planet Wa-Tor, der har form som en torus og er fuldstændig dækket af vand. Verdenen er inddelt i felter, som kan være tomme, indeholde en fisk eller en haj[8, s. 20]. Følgende karakteristika beskriver fisk og hajers opførsel:

Fisk Lever af plankton, en ressource som er uendelig. Hvis der er ét ledigt tilstødende felt, bevæger den sig til dette felt. Hvis der er flere ledige felter vælges et tilfældigt. Såfremt en fisk overlever tre tidsskridt, formerer den sig.

Hajer Såfremt der er fisk i et eller flere tilstødende felter, vil hajen bevæge sig til et af disse felter og spise fisken. Hvis der er ikke er nogen fisk i et af disse felter flytter hajen sig til et tilfældigt valgt ledigt felt. Hvis en haj ikke spiser i tre tidsskridt dør den. Overlever den i ti tidsskridt formerer den sig.

For hvert tidsskridt udfører alle fisk og hajer en handling ud fra ovenstående opførsel. Til at initiere systemet skal der defineres en størrelse af verdenen samt hvor mange fisk og hajer, der er til stede fra start. Disse fisk og hajer placeres tilfældigt i verdenen. Såfremt de initielle parametre for antal fisk og hajer understøtter en bæredygtig bestand, forventer vi at se bestanden af henholdsvis fisk og hajer oscillere afhængigt af hinanden.

Vi har valgt dette eksempel, da det er et klassisk simuleringsproblem, der er enkelt og let forståeligt, men samtidig introducerer problemstillinger omkring synkronisering, når det paralleliseres. Disse problemstillinger optræder, fordi en opdatering af hvert felt er afhængigt af de omkringliggende felter, og derfor også er afhængig af felter fra andre processer i grænsetilfælde. Ud over at være afhængig af information fra andre processer, kan en opdatering også påvirke data hos andre processer. Eksemplet er repræsentativt for de problemer, der ligger i kategorien af continuous simulation, og viser, at hvis vi kan bruge vores implementering af

DES til simulering af dette eksempel vil man helt generelt kunne bruge det til continuous simulations.

Før introduktion af et tidsbegreb i PyCSP

For at simulere Wa-Tor verdenen i PyCSP, hvori tidsbegrebet ikke er introduceret, er vi nødt til at udføre en synkronisering af de enkelte processers arbejde. Denne synkronisering kan ske ved brug af en barriere, hvor alle processer udfører en handling og mødes i barrieren, før de fortsætter.

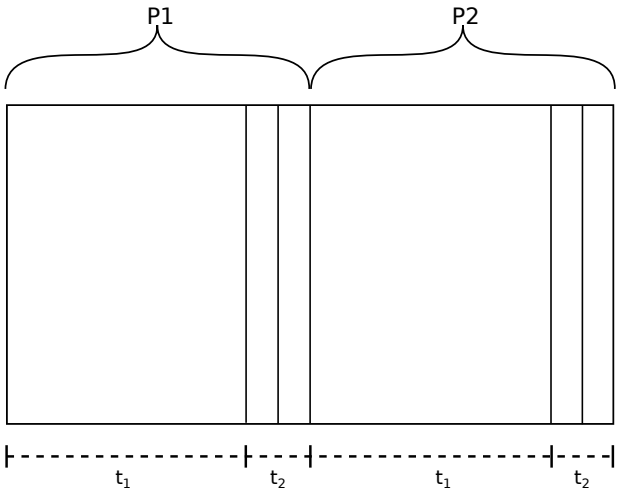
Vi har valgt at basere vores model på „Barrier Synchronisation for occam-pi“[2], hvor verdenen repræsenteres som en delt datastruktur og adgangen til denne styres med barrierer. I vores model er hver proces derved ansvarlig for en del af verdenen, og tilgangen til den delte datastruktur sker ud fra CREW-princippet (Concurrent Read, Exclusive Write)[2, s. 5], der styres vha. barrierer.

I en mere ren CSP-model ville man foretage en direkte udveksling af data mellem processerne og undgå den delte datastruktur, vi har i vores model. Vi har valgt denne model frem for den mere rene CSP model, da den klarlægger brugen af barrierer bedre. Det bliver meget eksplicit i koden, hvornår hvilke dele opdateres, og hvornår processerne venter i en barriere.

Vi deler verdenen lodret, og hver proces styrer en verdensdel. Hver proces gennemgår sin verdensdel og udfører en mulig handling for hver fisk og haj, dog vil fisk og hajer i de sidste to kolonner i hver verdensdel ikke bliver opdateret samtidigt med de resterende kolonner. Dette skyldes, at der kan opstå en race condition, hvis disse to kolonner opdateres samtidig med de andre. Årsagen til dette er, at både processen, der er ansvarlig for kolonnerne, og processen umiddelbart til højre for den skal have mulighed for at flytte fisk og hajer fra og til dette område. Når denne opdatering er fuldført, mødes processerne i en barriere, hvorefter hver proces opdaterer de to sidste kolonner. Herefter mødes processerne igen i barrieren, og når alle kolonner er opdateret, kan der foretages en visualisering af de udførte opdateringer. Til slut mødes alle i en barriere igen, før processerne kan begynde en ny iteration.

En repræsentation af opdelingen ses på figur 3.1 på næste side. Her er verdenen opdelt mellem to processer, og de to kolonner mellem hver del opdateres i et separat tidsskridt.

Afviklingen af programmet sker ved, at et antal worldpart-, en visualize- og en barrier-proces køres parallelt. Af kodeuddrag 3.2 og kodeuddrag 3.3 på side 16 ses det overordnede design af henholdsvis worldpart- og vizualize-processen.



Figur 3.1: Opdeling af verdenen mellem to processer. Der er for hver verdensdel to kolonner, som opdateres i et separat tidsskridt.

Konklusion

Det var ikke svært at implementere et predator-prey simulering i PyCSP, men da store dele af koden bruges på at simulere de forskellige fisk og ikke på synkronisering, forventer vi ikke, at introduktionen af tid i PyCSP vil medføre de store ændringer.

Værd at bemærke i kodeuddrag 3.3 på næste side er det store antal barrierekald i visualize-processen. Dette er til for, at man kan benytte den samme barriere som worldpart-processerne i kodeuddrag 3.2 på den følgende side. Alternativt kunne man have to barriere-processer; en til synkronisering af worldpart med vizualize, samt en som worldpart-processerne bruger til synkroniseringen af de to sidste kolonner.

3.1.2 Kunder i en bank

Vores andet eksempel indenfor diskret tid, er simulering af en bank. Det repræsenterer et klassisk eksempel inden for DES, hvor en række kunder, der alle ankommer til en butik, skal serviceres. Dette simple problem kan udvides til at modellere mange forskellige problemstillinger, der berører, hvordan flowet ændrer sig hvis man varierer en eller flere parametre i systemet. Programmeringssproget SimPy [14] har som et af deres eksempler en simulering af kunder i en bank. SimPy bruger dette som et gennemgående eksempel, hvor de løbende udvider modellen for at vise forskellige egenskaber ved SimPy. For at kunne sammenligne SimPy med vores version, vil vi implementere to af eksemplerne med kunder i en bank i PyCSP.

kodeuddrag 3.2: Uddrag af worldpart-processen i WaTor

```
@proces
def worldpart (part_id, barR, barW):
  ...
  while True:
    #Calc your world part:
    main_iteration()
    barW(1)
    barR()
    #Update the two shadowcolumns
    for i in range(world_height):
      for j in range(2):
        element_iteration(Point(right_shadow_col+j,i))
    barW(1)
    barR()
    #visualize have single access
    barW(1)
    barR()
```

kodeuddrag 3.3: Uddrag af visualize-processen i WaTor

```
@proces
def visualize(barR,barW):
  for i in xrange(iterations):
    barW(1)
    barR()
    barW(1)
    barR()
    pygame.display.flip()
    barW(1)
    barR()
  poison(barW,barR)
```

Figur 3.2: `barW(1)` og `barR()` er henholdsvis skrivning til og læsning fra en barriere, som defineret i afsnittet "Barrierer" i kapitel 3 på side 11.

I det simple tilfælde af eksemplet ankommer kunderne til banken på tilfældige tidspunkter. De opholder sig i banken i et tilfældigt tidsrum, hvorefter de igen forlader banken. I dette eksempel kan der ikke uddrages meget information, men det viser, hvordan en simpel model er opbygget i hhv. SimPy og PyCSP for at håndtere tid.

I det andet eksempel er modellen udvidet med en servicedisk, hvor alle kunderne skal betjenes af en servicemedarbejder, som kun kan ekspedere en kunde ad gangen. Alle kunder ankommer til banken på tilfældige tidspunkter og stiller sig i kø for at blive serviceret. Det er igen et tilfældigt tidsrum, som kunden bruger på at blive serviceret. Dette er stadigt en meget simpel model, men med introduktionen af en begrænset ressource kan man uddrage information om den tilhørende kø, f.eks. kan der måles, hvor lang det tager for hver kunde at blive betjent af servicemedarbejdere, samt hvordan køen opfører sig over tid.

Før introduktion af et tidsbegreb i PyCSP

I SimPy opretter generatorfunktionen dynamisk en kundeprocess og kører den parallelt med sig selv, I PyCSP opretter vi en generatorproces som i SimPy, men vi introducerer en bankproces i stedet for en kundeproces og lader kunderne være arbejdet, der flyttes mellem dem, over kanalerne. Dermed skal vi selv modellere vores ressource, som i dette tilfælde er banken, i modsætning til SimPy, hvor de blot bruger den interne ressource-klasse.

Mens SimPy kalder kundeprocessen fra generatoren og lader denne stå for håndteringen af kunden og den tid, hun befinder sig i banken, kender bankprocessen i PyCSP ikke tid som sådan. Bankprocessen skal derfor selv vedligeholde en liste med de kunder, der findes i banken og til hvert tidskridt vide hvilke kunder, der skal forlade den.

Tiden er igen modelleret ved brug af barrierer. I stedet for at have de to kald til barrieren, som den kræver, lige efter hinanden, lader vi bankprocessen gå ind i barrieren i starten af tidsskridtet, og så modtage kunder, indtil den modtager det andet kald fra barrieren (se kodeuddrag 3.4 på næste side). Dette er nødvendigt for at lade banken have mulighed for at modtage et vilkårligt antal kunder i samme tidsskridt, samt vide hvornår der ikke vil komme flere kunder. Vi kan, ved simpel indsigt, ræsonnere os frem til, at barrieren stadig virker efter hensigten. For at generatorprocessen kan komme foran med et tidsskridt og sende en kunde i et forkert tidsskridt, skal den have fuldført begge begge kald til barrieren, samtidig med at bankprocessen ikke har modtaget et kald fra barrieren. Barrieren vil dog ikke modtage kald fra nogen, før den har kaldt bankprocessen. Derfor må generatorprocessen vente i sit første kald til barrieren, indtil banken har modtaget sit

kald fra barrieren, før den kan sende en ny kunde. Når bankprocessen har modtaget et kald fra barrieren er den ikke længere villig til at modtage kunder før i det efterfølgende tidsskridt. Vi kan bruge samme ræsonnement for at bankprocessen heller ikke kan komme et tidsskridt foran generatorfunktionen, og derfor virker barrieren stadigt som forventet.

kodeuddrag 3.4: Modtage en kunde eller et barrierekald i bankprocessen

```
1 while True:
      (g,msg) = Alternation([{
3     barrierREADER:None,
      customerREADER:None
5     }]).select()
      if g == barrierREADER:
7       break
      elif g == customerREADER:
9       heappush(customers,(time+msg.waittime,msg))
```

I det mere avancerede eksempel, hvor kunderne skal tilgå den samme begrænsede ressource, dannes en kø. Denne kan i PyCSP modeleres på flere måder afhængigt af hvilken proces, der skal have ansvaret for at vedligeholde køen. En metode er internt i en proces at have en liste af kunder, der venter, og lade det være processens ansvar at håndtere denne liste som en kø. Processen med ansvaret for køen kan så enten være den begrænsede ressource eller en separat proces, hvis eneste formål er at vedligeholde køen. For nyligt[1] er der i PyCSP blevet introduceret bufferkanaler[19], og disse kan også bruges som en kø. Dermed kan man modellere sin proces uden hensyntagen til håndtering af køen, og blot lade processen læse fra kanalen, når den er klar. Vi har i dette eksempel valgt at lade køen være repræsenteret ved en bufferkanal, da denne kræver færrest linjer kode, men den kunne lige så godt være repræsenteret som en liste i servicedisk-processen.

Konklusion

Ved at sammenligne implementeringen af de to eksempler af kunder i en bank med implementeringen i SimPy, kan man se, at DES egner sig godt til simulering af disse problemer, og fra eksemplet kan man se, at der findes meget kode til vedligeholdelse af de interne tidsvariabler, som SimPy ikke har behov for. Det drejer sig om kode, der sørger for, at hver proces kender tiden, samt at den er synkroniseret på tværs af processerne. Vi forventer derfor at koden kan simplificeres i PyCSP med tid, så det bliver lige så simpelt at implementere som i SimPy.

[1] d. 22. december 2009.

I SimPy findes begrebet ressource direkte i sproget som en type, og servicedisken er blot et objekt af denne type. En ressource i SimPy bruges til modellering af en begrænset ressource. Ved oprettelsen af ressourcen angives hvor mange, der samtidigt kan tilgå den begrænsede ressource, og ressourcen står selv for adgang til den samt den tilhørende kø, der eventuelt måtte opstå. I PyCSP modelleres servicedisken som en separat proces, og bankprocessen er derfor reduceret til blot at sende kunden videre til servicedisken og lade kanalen håndtere køen.

En ulempe ved brugen af bufferkanaler som kø er, at kanalen har en fast størrelse på sin buffer, som angives, når kanalen oprettes. Man kan dermed risikere en deadlock ved brug af bufferkanaler sammen med barrierer. Dette opstår, hvis ikke generatoreprocessen i samme tidsskridt kan foretage kommunikationen og kalde barrieren. Det kan løses med en `alternation` og `skip guard`, men så skal bankprocessen håndtere fejlede forsendelser og må nødvendigvis introducere en sekundær kø, hvilket gør brugen af en bufferkanal overflødig. For undgå dette har vi i vores tilfælde valgt blot at angive en maksimal størrelse på bufferen som er større end det totale antal kunder banken modtager.

3.2 Design og implementering

For at designe en implementering af simulering i diskret tid i PyCSP, skal vi foretage en række ændringer i forhold til den nuværende implementering. Specifikt skal vi ændre på planlægningen og eksekveringen af processer, hvortil vi har brug for at kunne repræsentere en diskret tidsmodel. Vi vil i dette afsnit gennemgå de relevante problemstillinger og løsningsmodeller samt give et overblik over, hvordan vi har valgt at implementere ændringerne rent praktisk i koden.

3.2.1 Kodestruktur

Efter i kapitel 2 at have valgt at udvide greenlets-versionen, skal vi vælge hvordan vi ønsker at videreudvikle koden. Vi forventer at genbruge store dele af koden fra greenlets-versionen, og kun foretage udvidelser på enkelte afgrænsede områder. Desuden ønsker vi at isolere vores ændringer fra den originale greenlets-version. Med denne isolation forventer vi, at hvis/når der sker tilrettelser af greenlets-versionen af PyCSP, vil man ikke skulle foretage de samme tilrettelser i vores version. Isolationen mellem de to versioner skal opnås via nedarvning, således at det fra en brugers synsvinkel ser ud til, at vores version er fuldstændigt sepereret fra greenlets-versionen.

Hver af de tre versioner har sin egen mappe i PyCSP og i hver af disse findes en tilhørende `__init__.py` fil, der fungerer som et manifest for den givne version.

Vi opretter vores egen version kaldet *simulation* og opretter også en tilhørende mappe på samme niveau som de andre versioner og med sin egen manifestfil. Manifestfilen kan nu bruges til at udvælge de funktioner, der skal tages direkte fra greenlets-versionen, og hvilke funktioner, der skal udvides og som derfor vil ligge i den nye mappe.

kodeuddrag 3.5: Uddrag af `__init__.py` for simulationsversionen.

```
1 from guard import Timeout, Skip
  from pycsp.greenlets.alternation import choice
3 from alternation import Alternation
  from pycsp.greenlets.channel import ChannelPoisonException, ChannelRetireException
```

I kodeuddrag 3.5 på denne side kan man se, at funktionerne `choice`, `ChannelPoisonException` og `ChannelRetireException` alle bliver hentet fra greenlets-versionen, mens funktionerne `Timeout, Skip` og `Alternation` bliver importeret fra samme mappe og derfor er modificerede versioner. For udvikleren vil dette dog ikke være synligt, og han vil blot se simulation-versionen som en selvstændig version på lige fod med de andre tre versioner.

3.2.2 `Scheduler`-klassen

Med valget af greenlets-versionen som grundversion, og med henblik på at hovedparten af vores ændringer vil være i skemaplanlæggeren, vil vi kort gennemgå dele af klassen `Scheduler`.

I kodeuddrag 3.6 på næste side ses et uddrag af initialiseringskoden, der er interessant, fordi det er her alle de interne datastrukturer oprettes. Man kan her se de tre lister af processer, som skemaplanlæggeren har til rådighed til at varetage processkiftene.

`new`: Initieres på linje 140 og består af processer, som lige er blevet planlagt for første gang. Nye processer kan ankomme til listen `new` via funktionerne `Parallel` og `Spawn`.

`next`: Initieres på linje 141 og indeholder de processer, der er klar til at blive kørt, og som har været kørt på et tidligere tidspunkt. Et eksempel kunne være en proces, der har stoppet sin kørsel for at vente på kommunikation. Processen vil blive placeret i denne liste, når kommunikationen er overstået.

`timers`: Initieres på linje 147 og indeholder de processer, der har tilknyttet en timeout. De skal først planlægges på et senere tidspunkt og venter dermed

kodeuddrag 3.6: Uddrag af Scheduler.py i greenlets-versionen.

```
        def getInstance(cls, *args, **kargs):
            '''Static method to have a reference to **THE UNIQUE** instance'''
            if cls.__instance is None:
135             # (Some exception may be thrown...)
                # Initialize **the unique** instance
                cls.__instance = object.__new__(cls)

                # Initialize members for scheduler
140             cls.__instance.new = []
                cls.__instance.next = []
                cls.__instance.current = None
                cls.__instance.greenlet = greenlet.getcurrent()

145             # Timer specific  value = (activation time, process)
                # On update we do a sort based on the activation time
                cls.__instance.timers = []

                # Io specific
150             cls.__instance.cond = threading.Condition()
                cls.__instance.blocking = 0
```

blot. Hvert element i listen består både af processen samt et tidsstempel for hvornår processen skal genaktiveres.

`blocking`: Initieres på linje 151 og er en variabel. Processer, der venter på IO-operationer, er ikke klar til at blive planlagt, men heller ikke afsluttet. Skemaplanlæggeren kan derfor ikke planlægge dem, men holder styr på antallet af ventende processer vha. denne variabel. Dette bruges bla. for at kunne afgøre om skemaplanlæggeren har planlagt alle processer.

Når skemaplanlæggeren er startet, gennemløber den alle tre lister gentagne gange, indtil de alle er tomme, og der ikke er nogle processer, der er blokeret. Dette betyder at der ikke længere kan komme nye processer til der ønsker at blive planlagt, og dermed kan skemaplanlæggeren afslutte.

For at markere at vi ikke kun skal planlægge rækkefølgen af processerne, men også styre tiden, har vi lavet en `Simulation`-klasse, der arver fra `Scheduler`-klassen. Alle ændringer vi skal foretage for at kunne planlægge processer under hensyntagen til tid, vil således være indkapslet i denne klasse. Dette har yderligere den fordel, at man tydeligt kan se at alle klasserne i simulation-versionen arver en skemaplanlægger fra simulerings-versionen og ikke en skemaplanlægger fra greenlets-versionen.

3.2.3 Tid

For at kunne planlægge begivenheder i DES kræves det at alle processer og skemaplanlæggeren har en global forståelse af tid. Det er derfor en af hjørnestenene i implementeringen af DES, hvordan tid introduceres til PyCSP. Begrebet tid er ikke en del af CSP, men er alligevel blevet introduceret i PyCSP, for at kunne tilknytte en timeout til en `alternation`. Til at introducere tid er `time`-modulet brugt der benytter sig af en realtidsmodel. Vi ønsker at videreudvikle PyCSP, så det ikke kun er i forbindelse med en `alternation` at PyCSP kender til tiden, men at PyCSP generelt kan håndtere tid for alle processer.

Kodeuddrag 3.7 på denne side viser et eksempel på brugen af tid i det eksisterende PyCSP, hvor en `alternation` er villig til at læse fra kanalenden `Cin` i 0,5 sekunder. Hvis ikke der er modtaget en besked indenfor 0,5 sekunder, accepteres dens `timeout-guard`, og processen fortsætter sin kørsel uden at have læst fra kanalen.

Kodeuddrag 3.7: Timeout i Alternation (fra dokumentationen til PyCSP)

```
Alternation([{Timeout(seconds=0.5):None},
             {Cin:None}]).select()
```

I `time`-modulet er den underliggende tidsmodel realtidsmodellen, hvor tiden frem af tiden selv. Vi skal derfor ændre PyCSP så den bruger en diskret tidsmodel. Denne ændring vil medfører at fremskrivningen af tiden ændres så den drives af begivenheder, og ikke af tiden selv.

Da man i `time`-modulet har et fast tidsskridt, og realtiden også er inddelt i faste størrelser som eks. sekunder, kan man med `time`-modulet måle tidsintervaller, der korresponderer med realtiden. I DES findes der ikke en sammenhæng til realtiden, da tiden blot er et tal, der starter som 0, og stiger i arbitrære tidskridt. Når tiden i DES på denne måde er afkoblet en relation til realtid, giver det ikke mening at have elementer i simuleringen, der er afhængige af realtid. I PyCSP kan man planlægge en timeout til at ske om f.eks. 5 sekunder. I DES findes sekunder som begreb ikke, men man angiver i stedet, at når tiden er talt op med 5 tidsskridt, skal begivenheden ske. Der findes dog ikke en total afkobling mellem tiden i DES og realtiden, for givet et konkret problem, der skal modelleres i DES, vil der altid være en sammenhæng mellem tiderne. Men da denne sammenhæng ikke er fast, skal den defineres eksplicit af udvikleren, som f.eks at fem sekunder i problemet defineres som en stigning i tiden med f.eks 5, 0,5 eller 0,05 i simuleringsmodellen.

Når tiden i DES er uafhængig af realtiden, er der ingen grund til at bruge

en kompleks model af tiden, og vi har derfor valgt at repræsentere tiden som et positivt tal, der findes internt i skemaplanlæggeren. Dermed findes der kun en version af tiden, da skemaplanlæggeren er en singelton. For processer, der ønsker at kende tiden, har vi introduceret funktionen `Now`, der returnerer tiden fra skemaplanlæggeren, når funktionen kaldes. En fordel ved brugen af funktionen `Now` som en wrapperfunktion til at bede om tiden i forhold til den eksisterende kode, der direkte kalder `time`-modulet, er, at vi frigøres fra en konkret implementering af tid. For fremtiden er det kun funktionen `Now`, der skal omskrives, for at hele systemet bruger en anden implementering tid.

I den eksisterende kode har det ikke været tiltænkt, at man ønskede at udskifte implementeringen af tiden, vi skal derfor ændre de steder, hvor `time` -modulet er refereret. Heldigvis bruges `time`-modulet kun ved udvælgelse af processer fra `timers`-listen (kodeuddrag 3.8 på denne side). Her sammenlignes på linje 204

kodeuddrag 3.8: Udvælgelse af proces fra listen timers (fra scheduling.py)

```
204 if self.timers and self.timers[0][0] < time.time():
      _,self.current = self.timers.pop(0)
206   self.current.greenlet.switch()
```

kodeuddrag 3.9: Udvælgelse af proces fra listen timers (fra simulation.py)

```
124 if self.timers and self.timers[0][0] <= Now():
      assert self.timers[0][0] == Now()
126   _,self.current = heapq.heappop(self.timers)
      self.current.greenlet.switch()
```

tidsværdi for den første proces i `timers` med det nuværende tidspunkt givet af `time`-modulet. Hvis det nuværende tidspunkt er større end værdien i `timers`, udvælges denne proces til at køre næste gang og fjernes fra listen. For at planlægge begivenheder præcist, skal processerne kunne eksekveres på et specifikt tidspunkt. Dermed har skemaplanlæggeren i simulations-versionen behov for at kunne styre aktiveringen af processerne på et finere niveau end, hvad der er muligt med greenlets-versionen. I simulerings-versionen har vi fuld kontrol over tiden, da den står stille, mens processerne eksekveres, hvorfor tiden i dette tilfælde ikke er et problem. Vi har i simulerings-versioner tilføjer den begrænsning, at tiden skal være præcist det, der er angivet i `timers`-listen, før processen skal aktiveres, og ikke kun større, som angivet i greenlets-versionen. Kodeuddrag 3.9 på denne side viser udvælgelsen af en proces fra `timers` i simulerings-versionen ved brug af `Now`-funktionen, hvor tidspunktet skal være præcist det som processen har

angivet.

Fremskrivning af tid

I pseudo realtid drives tiden frem af et eksternt modul for at efterligne realtid, der kontinuerligt stiger. I pseudo realtid fremskrives tiden derfor uafhængigt af processernes tilstand og derfor vil et program der med gentagende gange beder om tiden, få et stigende tidspunkt. I DES skal tiden i modsætning til realtid stå stille, når processerne er aktive, og kun i forbindelse med en planlagt begivenhed skal tiden drives frem til tidspunktet for denne begivenhed.

Vi kan demonstrere, hvordan den kontinuerlige tid har indflydelse på PyCSP med et eksempel. Proces 1 har startet en ny tråd via `io-decoratoren` og er derfor blokeret. Proces 2 står i en `alternation` med en `timeout-guard`. Uafhængigt af den tid, det tager proces 1 at komme ud fra sit blokerede kald, skal proces 2 vide hvornår dens timeout er indtrådt. Dette er implementeret i greenlets-versionen i kodeuddrag 3.10 på næste side på linjerne 242 til 251. Her startes en separat tråd, der signalerer skemaplanlæggeren, når tiden for næste begivenhed i `timers` listen indtræffer. Skemaplanlæggeren kan nu nøjes med at vente på et signal, som vil komme fra enten en blokeret proces eller den nyoprettede tråd.

Når tiden i DES ikke er drevet af en eksternt modul, er nødvendigheden af en ekstra tråd til håndtering af tid irrelevant. Først når alle begivenheder til et tidsskridt er eksekveret, skal tiden i DES tælles op. Dette betyder i vores konkrete eksempel, at så længe proces 1 er blokeret, står tiden stille, og skemaplanlæggeren venter på dem. Vi kan ikke tælle tiden op, blot fordi nogle processer er blokeret af `io-decoratoren`, ligesom vi ikke kan tælle tiden op, så længe der er processer der er aktive.

Først når alle processer venter i enten `timers` listen eller på kommunikation, kan der ikke ske flere begivenheder og tiden fremskrives. I dette tilfælde kan skemaplanlæggeren finde tidspunktet for den begivenhed, der ligger tættest på det nuværende tidspunkt, og springe frem til denne begivenhed. Dette er implementeret i kodeuddrag 3.11 på modstående side.

Timeout

I PyCSP findes der, som nævnt i kapitel 2 en `alternation`, hvor udvikleren har mulighed for at tilknytte to specielle guards. Den ene er en SKIP-guard, der giver mulighed for at kommunikere, hvis kanalen er klar, og ellers fortsætte uden at kommunikere. Den anden er timeout-guarden, der udvider SKIP-guarden, så man venter på kommunikation i en given periode, hvorefter man tager SKIP-guarden. Når DES indføres, ændres tiden, så timeout-guarden opererer på tidsskridt fremfor

kodeuddrag 3.10: Uddrag af skemaplanlæggeren i `Scheduler`

```
    self.cond.acquire()
240 if not (self.next or self.new):
        # Waiting on blocking processes or all processes have finished!
242     if self.timers:
            # Set timer to lowest activation time
244         seconds = self.timers[0][0] - time.time()
            if seconds > 0:
246             t = threading.Timer(seconds, self.timer_notify)
                # We don't worry about cancelling, since it makes no
248             # difference if timer_notify is called one more time.
                t.start()
250             # Now go to sleep
                self.cond.wait()
252     elif self.blocking > 0:
            # Now go to sleep
254         self.cond.wait()
        else:
256         # Execution finished!
            self.cond.release()
258         return
    self.cond.release()
```

kodeuddrag 3.11: Uddrag af skemaplanlæggeren i Simulation-versionen

```
158 self.cond.acquire()
    if not (self.next or self.new):
160   # Waiting on blocking processes
      if self.blocking > 0:
162     # Now go to sleep
        self.cond.wait()
164   #If there exist only processes in timers we can increment
      elif  not (self.next or self.new or self.blocking):
166       if self.timers:
              # inc timer to lowest activation time
168           self._t = self.timers[0][0]
          else:
170           # Execution finished!
              self.cond.release()
172           return
    self.cond.release()
```

en tidsperiode. Vi går derved fra en situation hvor en timeout-guard f.eks. er villig til at vente i fem sekunder, til en situation hvor den er villig til at vente i 5 tidsskridt. Denne ændring virker umiddelbart simpel, men introducerer et problem i forhold til hvornår timeout-guarden vælges: En proces kan nu ønske at kommunikere i indeværende tidsskridt, men ikke i det efterfølgende. Vi kan dog ikke i tidsskridtet evaluere om kommunikation er muligt. Dette skyldes, at tiden står stille, mens processerne er aktive, så selvom kommunikation ikke er muligt på ét tidspunkt i tidsskridtet, så kan en efterfølgende begivenhed i samme tidsskridt muliggøre kommunikation.

En løsningsmodel på denne problemstilling kunne være at lade processer i timeout-guards vente helt frem til næste tidsskridt, og så her lade dem vælge SKIP-guarden. Dette efterfølgende tidsskridt kan enten være et ekstra skridt, vi introducerer specifikt for at håndtere disse timeout-guards, eller det kan være det næste tidsskridt, hvortil der er planlagt en begivenhed. Såfremt vi introducerer et kunstigt tidsskridt, skal dette specificeres i definitionen af timeout-guarden. Uanset hvor lille et tidsskridt vi specificerer, kan vi ikke garantere, at der ikke efterfølgende i samme tidsskridt vil blive planlagt en begivenhed, der skal indtræffe mellem indeværende tidsskridt og det kunstige tidsskridt, vi specificerede i timeout-guarden. Herved er der risiko for, at rækkefølgen af begivenheder, der skal udføres, bliver ændret, hvilket er uacceptabelt. Alternativt kan man vælge at lade tiden springe til den næste begivenhed, der er planlagt, og der som det første vælge SKIP-guarden. Her vil man ikke risikere at ændre på rækkefølgen, men derimod at springe for langt frem i tiden. Dette kan være et problem, hvis en proces, der tager en SKIP-guard, efterfølgende planlægger en begivenhed. Begge muligheder har grundlæggende den svaghed, at oprindeligt ønskede man kun at kommunikere i det indeværende tidsskridt og ikke i hverken et vilkårligt lille tidsrum eller i et tilfældigt tidsrum frem til en efterfølgende begivenhed.

En sidste løsningsmodel er at vente til lige før, tiden tælles op, og der kalde de ventende processers SKIP-guards. For at kunne adskille hvilke processer, der har en begivenhed til et tidsskridt, og hvilke, der venter på kommunikation, kan vi benytte os af edge-triggering til at dele hvert tidsskridt op i to grupper, henholdsvis wake-first og wake-last. I wake-first-gruppen udføres de processer, der har tilknyttet en begivenhed til det givne tidsskridt, mens man i wake-last-gruppen aktiverer SKIP-guarden for de processer, der venter på en timeout.

Edge-triggering er den bedste løsningsmodel af de beskrevne, da man her har mulighed for at udføre alle begivenheder, som måske resulterer i kommunikation, og først derefter aktivere SKIP-guards for de processer, der har en timeout til samme tidsskridt. Ulempen ved denne metode er at det kræver en mere kompleks

implementering, da der nu findes to seperate måder at vente for hhv. begivenheder og timeouts.

3.2.4 Planlægning af begivenheder i fremtiden

Vi har valgt at anskue planlægningen af en begivenhed til et bestemt tidspunkt, sådan at den proces der skal udføre begivenheden venter indtil tiden for begivnheden er nået, og først her begynde udførslen. Dette vil i praksis være det samme som en planlægning til tidspunktet men det letter implementeringen da vi ikke behøver nogen viden om specifikke begivenheder i vores skemaplanlægger.

kodeuddrag 3.12: Et yield i SimPy (taget fra Bank05.py i eksemplet fra SimPy)

```
   def visit(self,timeInBank):
12     print now(), self.name," Here I am"
       yield hold,self,timeInBank
14     print now(), self.name," I must leave"
```

I programmeringssproget SimPy benytter man også denne metode med at lade en proces vente. Dette gøres ved at foretage kaldet `yield`, som sørger for, at processen ikke fortsætter, før et defineret tidsrum er gået. Kodeuddrag 3.12 på denne side viser, at en kunde er ankommet til banken, hvorefter kunden printer tiden, foretager et `yield`, printer tiden igen og afslutter. Når processen har kaldt `yield`, er tiden steget med værdien af `timeInBank`. Grunden til at SimPy kan bruge `yield`, der er indbygget i Python og at dette kald afgiver kontrol over processen i et tidsrum, knytter sig til deres implementering af SimPy, hvor de bruger `corutine` som underliggende teknologi. Som standard kan `corutine` afgive kontrollen med en proces via `yield`, og SimPy behøver derfor kun at sikre, at tiden er talt tilstrækkeligt op, før de returnerer til processen.

Vi ønsker i PyCSP at have en lignende mulighed for at lade en proces vente. Med `greenlet`-modulet af brugertråde kan vi ikke bruge `yield`, da denne er specifik for `co-rutines`, men funktionaliteten er allerede delvist introduceret via `timeout` til `alternation` i greenlets-versionen af PyCSP. Vi kan derfor bygge videre på denne funktionalitet med funktionen `Wait`, der fungerer som timeout, men som kan kaldes af processerne på et vilkårligt tidspunkt.

Funktionen `Wait` står for at kalde den interne funktion, der er lavet til timeouts, kaldet timer_wait, hvorved processen lægges i `timers`-listen, og herefter sørge for først at returnere, når tiden er steget til det krævede. `Wait` er reelt det

Kodeuddrag 3.13: Wait i simulering-versionen.

```
20 def Wait(seconds):
     Simulation().timer_wait(Simulation().current, seconds)
22   t = Now()+seconds
     while Now()<t:
24     p = Simulation().getNext()
       p.greenlet.switch()
```

eneste værktøj, der skal til for at vente, og dermed planlægge begivenheder ud i fremtiden, og vi har på nuværenede tidspunkt en simpel DES, der kører i realtid.

3.2.5 Timers

I PyCSP bruges listen `timers` til at placere processer, der venter på en timeout i `alternation`. Dette er en niche feature ved PyCSP, som sjældent bruges, og hvor der sjældent er samlet mange processer på en gang. I afsnit 3.2.4 på forrige side beskriver vi hvordan processer, der ønsker at vente lægges på `timers`-listen, og i afsnit 3.2.3 på side 22 beskriver vi, hvordan skemaplanlæggeren kun tæller tiden op, når ingen processer kan foretage sig mere i et givent tidsskridt. Når tiden tælles op, vil alle processer enten vente på kommunikation eller befinde sig i listen `timers`. De processer, der venter i en `alternation` og har tilknyttet en timeout, ligger begge steder. Gennemsnitslængden af listen vil derfor stige voldsomt i vores version og dermed ændres kravene til hvilken datastruktur der er bedst egnet. Med en kort, sjældent brugt liste vil omkostningerne til oprettelse og vedligeholdelse af en avanceret datastruktur være større end fordelene. Til vores skemaplanlægning vil en min-hob være det åbenlyse valg, da man kan indsætte elementer i konstant tid og fjerne det mindste element i $O(log\ n)$. Skemaplanlægning er specifikt nævnt i introduktionen til Pythons implementering[9] som eksempel på anvendelsesmulighederne for en hob.

Da implementeringen af en hob allerede findes i Python i modulet `heapq`, som er effektivt implementeret i C, vælger vi at bruge denne. Den eneste handling, der ikke er implementeret som standard, er fjernelsen af et arbitrært element fra hoben. Dette sker i den eksisterende løsning, når en proces aktiverer et andet valg i `alternation` end timeout. I dette tilfælde skal processen ikke vente på sin timeout, og derfor skal elementet fjernes fra `timers` listen. Her må man, som i en normal liste, lave en lineær søgning i hoben og derefter genoprette hob-egenskaben i listen. Det vil dog ikke tage længere tid, da fjernelse af en timeout i greenlets-versionen på nuværende tidspunkt bruger en lineær søgning, til at finde elementet der skal fjernes, og genoprettelsen af hob-egenskaben også kan gøres i

lineær tid.

Det kræver ikke meget omskrivning for at konvertere en liste til en hob, hvilket ses ved at sammenligne kodeuddrag 3.8 på side 23 linje 205 med kodeuddrag 3.9 på side 23 linje 126.

3.2.6 Annekteret kode fra SimPy.

I vores implementering er der en del overlap med SimPy, da det har været en inspirationskilde til hvordan et simuleringssprog kan udvikles i Python. En del af arbejdet med SimPy har vi kunne bruge direkte i vore implementering efter devisen om ikke at genskrive eksisterende god kode. SimPy har en `Monitor`-klasse, der kan benyttes til dataindsamling, bearbejdning og visualisering. Denne klasse har vi i stor udstrækning genbrugt. Den fungerer ved at gemme en liste af tid/-værdi par. Dermed kan man efter endt simulering, analysere hvordan værdierne har ændret sig over tid. Klassen `Monitor` kan bruges direkte af udvikleren, hvor de selv står for at at gemme værdier på passende tidspunkter igennem kørslen af programmet. Man ønsker tit at kende længden af en kø, der som oftest er implementeret via en liste. Vi har derfor lavet vores egen liste der kan indeholde en `Monitor`. Når længden af listen ændres, gemmes længden af listen i en montor til brug for senere analyse, uden udvikleren selv skal stå det. Alternativt kan man lave en separat proces, hvis eneste formål er med en given frekvens at gemme listens længde. Fordelen ved denne løsning er at intervallet er jævnt fordelt, og man derfor lettere kan foretage tidsspecifik statistik.

3.3 Evaluering

Vi har det foregående afsnit beskrevet hvad der skal til for at implementere DES i PyCSP. Vi vil i dette afsnit evaluere vores løsning, med udgangspunkt i de eksempler vi tidligere har opstillet.

3.3.1 Test af korrekthed

Vi har igennem designet og udviklingen af simulerings-versionen brugt Test Driven Development (TDD). I TDD starter man med at skrive tests til den nye egenskab, der skal udvikles. Efterfølgende skrives koden, så testen kan køres uden at fejle, og per defintion er designet implementeret korrekt, når alle tests kan gennemføres korrekt. Dette medfører, at vi løbende i forbindelsen med udviklingen af simulerings-versionen har skrevet tests.

For yderligere at teste den udviklede kode har vi desuden integreret alle tests fra greenlets-versionen ind i simulerings-versionen. Således er alle de tests, der

er skrevet til greenlets-versionen også er med til at teste simulerings-versionen. Resultaterne af de udførte tests kan ses i bilag A.1 og viser, at alle tests forløb tilfredstillende.

3.3.2 Eksempler

For at evaluere fordele og ulemper af simulerings-versionen af PyCSP, har vi genimplementeret eksemplerne fra afsnit 3.1 på side 12, med brug af vores udviklede kode. Vi kan derfor i det følgende sammenholde de to versioner og se på fordele og ulemper ved simulerings-versionen.

Hajer og fisk på Wa-Tor

Implementeringen af dette problem i et simuleringssprog har ikke medført den store omskrivning. Dette skyldes, at eksemplet er en kontinuerlig simulering, og at alle fisk og hajer ønsker at interagere med omverdenen i hvert tidsskridt. I processen `visualize` ser man tydeligst forskellen mellem standard PyCSP med dens brug af barrierer og simulerings-versionen med dens brug af tid vha. funktionen `Wait`. Hvor standard PyCSP må kalde en barriere tre gange for hver iteration (kodeuddrag 3.14 på denne side), kan man i simulerings-versionen blot angive, at visualiseringen ønsker at vente tre tidsskridt (kodeuddrag 3.15 på modstående side).

kodeuddrag 3.14: greenlets-versionen af visualize

```
   @process
158 def visualize(barR,barW):
      for i in xrange(iterations):
160     barW(1)
        barR()
162     barW(1)
        barR()
164     pygame.display.flip()
        barW(1)
166     barR()
      poison(barW,barR)
```

I arbejderprocessen (kodeuddrag 3.16 på næste side) kan man også se hvordan brugen af tid sikrer adgang til en delt ressource. I greenlets-versionen skulle barrieren kaldes flere gange i træk for at sikre, at en delt datastruktur ikke blev brugt af processen. Med simulerings-versionen kan man, som det ses på linje 140 i

kodeuddrag 3.15: simulerings-versionen af visualize

```
144 @process
    def visualize():
146   for i in xrange(iterations):
        Wait(3)
148     pygame.display.flip()
        print "%d: vizualized"%Now()
```

kodeuddrag 3.16: Uddrag af arbejderprocessen i simulering

```
130 Wait(1)
    for i in xrange(iterations):
132   #Calc your world part:
      main_iteration()
134   Wait(1)
      #Calc the two shadowrows
136   print "%d: shadow row "%Now()
      for i in range(world_height):
138     for j in range(2):
          element_iteration(Point(right_shadow_col+j,i))
140   Wait(2)
```

kodeuddrag 3.16 på denne side, nøjes med at kalde `Wait` og vente i to tidsskridt. `Wait` sikre hermed mod at processerne kan skrive samtidigt til den samme data da arbejderprocessen ligger og venter i kø på `timers` mens visualiseringsprocessen tilgår data. Brugen af tid kan på også med fordel bruges hvis der er endnu flere processer skal have eksklusive rettigheder. Dette løses ved blot at øge antallet af tidsskridt, hver proces venter. Hvis dette skulle opnås med barrierer, skulle man lave en løkke, der et antal gange lod processen gå igennem barrieren.

Kunder i en bank

De to bankeksempler har krævet en betydelig omskrivning for at udnytte den nyudviklede simulerings-version. I generatorprocessen ses tydeligst forskellen på de to versioner. Greenlets-versionen bruger jævnfør kodeuddrag 3.17 på næste side, 20 linjer kode, hvor kodeuddrag 3.18 på den følgende side viser at simulerings-versionen kun bruger 6 linjer.

Den store forskel på de to funktioner er muligheden for at hoppe fra et tidspunkt til det næste i simuleringen. Dermed skal funktionen ikke være aktiv i hvert tidsskridt, men kun i de tidsskridt, hvor der skal produceres en kunde. Når processen kun er aktiv i de relevante tidsskridt, kan vi undgå hjælpevariablerne `t_event, time` og `numberInserted`.

I greenlets-versionen kan man af kodeuddrag 3.17 på næste side, linje 38 til

41 se at generatorprocessen forbliver aktiv efter den er færdig og ikke vil sende flere kunder. Dette gøres fordi antallet af processer der tilgår barrieren skal være det samme igennem hele kørslen (se afsnittet "Barrierer" i kapitel 3 på side 11). Denne begrænsning slipper man for i simulerings-versionen, hvor processen derfor kan stoppe, når den har sendt alle sine kunder.

kodeuddrag 3.17: Generatorprocessen for greenlets-versionen

```
@process
def Generator(i,number,meanTBA, meanWT,
              customerWRITER,barrierWRITER,barrierREADER):
  t_event = 0
  time = 0
  numberInserted = 0
  while numberInserted<number:
    if t_event<=time:
      customerWRITER(Customer(name = "Customer%d:%02d"%
                     (i,numberInserted),meanWT=meanWT))
      t_event = time + round(expovariate(1/meanTBA))
      numberInserted+=1
    barrierWRITER(0)
    barrierREADER()
    time+=1
  retire(customerWRITER)
  try:
    while True:
      barrierWRITER(0)
      barrierREADER()
      time +=1
  except ChannelPoisonException:
    return
```

Generatorfunktionen er omskrevet, så koden er mere relateret til funktionaliteten og mindre til synkronisering. I resten af koden opnår vi lignende kodekoncentration, men vi har også udvidet eksemplet, så vi gemmer antallet af kunder, der befinder sig i banken, i en `Monitor`. Dette er ikke strengt nødvendigt og gøres ikke i det originale eksempel fra SimPy, men er et godt eksemple på brugen af en `Monitor` til udtræk af data fra simuleringen. Det var forventet, at koden kunne forbedres, da eksemplet er et typisk DES problem, men det er tilfredsstillende, at vi opnår de forventede forbedringer i implementeringen af simuleringen i forhold til greenlets-versionen.

kodeuddrag 3.18: Generatorprocessen for simulerings-versionen

```
@process
def Generator(i,number,meanTBA, meanWT, customerWRITER):
  for numberInserted in range(number):
    customerWRITER(Customer(name = "Customer%d:%02d"%(i,numberInserted),
                            meanWT = meanWT))
```

```
      Wait(expovariate(1/meanTBA))
26  retire(customerWRITER)
```

kodeuddrag 3.19: Generator funktion for SimPy

```
  def generate(self,number,meanTBA,resource):
12    for i in range(number):
          c = Customer(name = "Customer%02d"%(i,))
14        activate(c,c.visit(b=resource))
          t = expovariate(1.0/meanTBA)
16        yield hold,self,t
```

En sammenligning af generatorfunktionen i SimPy med generatorprocessen i simulerings-versionen viser ikke den store forskel. I SimPy aktiverer man kunden direkte, mens man i simulerings-versionen sender kunden over en kanal. Der findes derimod en større forskel på de to implementeringer i kundedelen. Dette skyldes, at i simulerings-versionen findes der en bankproces, som er delt på tværs af alle kunder, mens man i SimPy har en kundefunktion, der er unik for hver kunde. Denne forskel medvirker til, at bankprocessen i SimPy skal vedligeholde afgangstiden for samtlige kunder i banken hvilket kræver mere kode. Dette er implementeret i kodeuddrag 3.20 på denne side på linjerne 46-53 med en `alternation`. Her venter processen hele tiden på enten at modtage en ny kunde, eller på at en kunden ønsker at forlade banken.

kodeuddrag 3.20: Uddrag af bank processen i simulation-versionen

```
      while True:
40      msg = customerREADER()
        print "%94.0f: %s enter bank"%(Now(),msg.name)
42      heappush(customers,(Now()+msg.waittime,msg))
        mon.observe(len(customers))
44      while len(customers)>0:
          print "%94.0f: B: timeout is:%f"%(Now(),customers[0][0]-Now())
46        (g,msg) = Alternation([(customerREADER,None),
                                 (Timeout(seconds=customers[0][0]- Now()),None)
48                             ]).select()
          if g == customerREADER:
50          heappush(customers,(Now()+msg.waittime,msg))
            print "%94.0f: %s enter bank"%(Now(),msg.name)
52        else:
            ntime,ncust = heappop(customers)
54          print "%94.0f: %s left bank"%(Now(),ncust.name)
```

kodeuddrag 3.21: Funktionen `visit` i SimPy

```
20      def visit(self,b):
            arrive = now()
22          print "%8.4f %s: Here I am     "%(now(),self.name)
            yield request,self,b
24          wait = now()-arrive
            print "%8.4f %s: Waited %6.3f"%(now(),self.name,wait)
26          tib = expovariate(1.0/timeInBank)
            yield hold,self,tib
28          yield release,self,b
            print "%8.4f %s: Finished      "%(now(),self.name)
```

kodeuddrag 3.22: Bankprocessen hvor banken er en begrænset ressource.

```
    @process
34 def Bank(customerREADER):
    try:
36      while True:
          print "%94.0f: B: waits for customer"%Now()
38        customer = customerREADER()
          print "%94.0f: B: adding customer %s to queue"
40              %(Now(),customer)
          Wait(customer.waittime)
42        print "%94.0f: B: customer  %s exits queue"
                %(Now(),customer)
44  except ChannelRetireException:
        print "%94.0f: B: got retire"%(Now())
```

I det avancerede eksempel som vist i kodeuddrag 3.22 på denne side indeholder bankprocessen mindre kode end i det simple eksempel. Det skyldes, at banken ikke længere skal holde styr på samtlige kunder, men blot skal håndtere en kunde ad gangen, mens resten af kunderne befinder sig i køen. Kundefunktionen i SimPy og bankprocessen i det avancerede eksempel minder derfor meget om hinanden og en fordel ved simulerings-versionen er, at man kun foretager en `Wait` for at indikere, at den begrænsede ressource er optaget, hvor man i SimPy versionen skal foretage tre kald. Først et kald for at få adgang til den begrænsede ressource, så et kald for at holde ressourcen i et tidsperiode, og til sidst et kald for at slippe ressourcen.

3.4 Fremtidigt arbejde

Vi har præsenteret en fungerende DES løsning som opfylder de primære formål til fulde. Sammenholdt med andre lignende løsninger kunne vi udvide vores løsning med funktionalitet til at foretage løbende evaluering af en given kø. Eksempelvis

kunne man forstille sig en simulering af en bank med to skranker med hver sin kø. En kunde ankommer og stiller sig i den korteste kø. Efter noget tid er den modsatte kø blevet kortest og kunden ønsker nu at skifte kø. Alternativt kunne kunden vurdere at ventetiden har været for lang og ønsker at forlade banken uden at komme frem til en skranke. Disse muligheder til at evaluere køer løbende er ikke implementeret på nuværende tidspunkt men vi mener at det kunne være interessant at kigge nærmere på, da det vil give mulighed for mere komplicerede kø-strukturer og mere effektiv udnyttelse af ressourcer.

Det vil ydermere være interessant at kigge på mulighederne for at implementere en struktur til at reservere flere ressourcer som skal bruges samtidigt, som f.eks. ved Dining Philosophers problemstillingen. Dette er et klassisk eksempel på hvordan deadlocks kan opstå og hvis vi kunne lave en konstruktion der kan hjælpe udviklere med at udgå deadlocks i den forbindelse ville det kunne lette udviklingen af simuleringer med DES i PyCSP.

I SimPy har man mulighed for at angive et tidspunkt som et stopkriterium. Dermed skal simuleringen køre frem til dette tidspunkt, uden at hver proces skal håndtere et stopkriterium. Det ville øge brugbarheden af DES i PyCSP hvis det også havde denne funktionalitet.

3.5 Opsummering

Vi har i dette afsnit gennemgået DES og redegjort for dens anvendelsesområder, og kommet frem til at en DES grundlæggende skal have tre egenskaber: En repræsentation af diskret tid, der kan styres af simuleringen, en liste over fremtidige begivenheder og muligheden for at opsamle statistisk data.

På baggrund af de tre grundlæggende egenskaber har vi defineret to eksempler der viser behovet for en bedre metode til at foretage DES i PyCSP. De to eksempler er begge implementeret med og uden brug af vores DES løsning. Vi har redegjort for at uden brug af DES, er det nødvendigt at benytte barrierer til at repræsentere tid, og at denne metode bryder med det grundlæggende princip i CSP om ikke at have delte datastrukturer.

På baggrund af eksemplerne og de grundlæggende egenskaber, har vi lavet en fuldstændig implementering af DES i PyCSP. Vores implementering løser problemet med at tid kræver en delt datastruktur, og derudover illustreres det, at det er simpelt for en udvikler at benytte vores implementering. Vi har vist at en implementering af eksemplerne med brug af vores løsning kræver væsentligt mindre kode, end hvis de var implementeret direkte i PyCSP. Generelt resulterer vores løsning derfor i en implementering der er mere overskuelig og hvor koden primært

udtrykker det konkrete problem der skal løses.

De to eksempler er af grundlæggende forskellig karakter, og taget med for at vise hvordan vores løsning håndterer to meget forskellige simuleringer. Der er forskel på hvor meget de respektive eksempler drager nytte af vores implementering, men begge eksempler kan repræsenteres bedre med, end uden, vores implementering.

Kapitel 4

Realtidsplanlægning

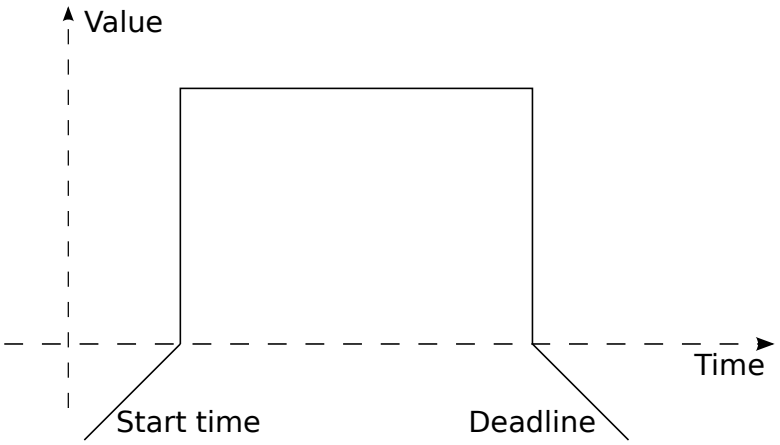
Det andet anvendelsesområde vil vi behandle, med henblik på at inføre tid i PyCSP, er Real-time planlægning (RTP). Vi vil i dette kapitel gennemgå RTP samt diskutere hvordan det kan implementeres i PyCSP.

RTP er baseret på den kendsgerning at nogle begivenheder i programmer kan være vigtigere at få udført end andre indenfor en afgrænset tidsperiode. Dette kan f.eks. være interrupts, input- eller output enheder eller interne processer. Med RTP tilknytter man en deadline for hver begivenhed, som bruges til at planlægge rækkefølgen for afvikling af begivenhederne. Formålet er at optimere antallet af begivenheder der når at blive udført inden deres deadline er overskredet. Normalt anses alle begivenheder for at være lige vigtige, og de planlægges ud fra en optimal udnyttelse af processoren. Denne optimale processorudnyttelse kan man være nødt til at gå på kompromis med hvis man ønsker at bruge RTP og derved prioritere visse begivenheder højere end andre. Man kan forestille sig en situation hvor man ikke starter en begivenhed med lav prioritet selv om den er klar, hvis man ved at en begivenhed med høj prioritet er klar kort tid efter, og venter derfor på at den er klar og igangsætter begivenheden med høj prioritet med det samme. RTP benyttes meget i specialiserede indlejrede systemer til f.eks medicinsk udstyr, kontrol af luftrummet, på rumstationen ISS[1] og mange andre steder. Det er dog også anvendeligt i mere gængse applikationer, typisk i forbindelse med en eller anden form for interaktion med den virkelige verden.

I litteraturen omkring RTP bruges begreberne hard- og soft deadlines samt hard- og soft real-time systemer forskelligt, så vi vil i det følgende gennemgå hvordan vi definerer disse begreber. Vi har valgt at illustrere deadlines ved hjælp af time-value funktioner, hvor "værdien" indikerer det bidrag begivenheden bidrager med til systemets overordnede mål.

Kritisk deadline

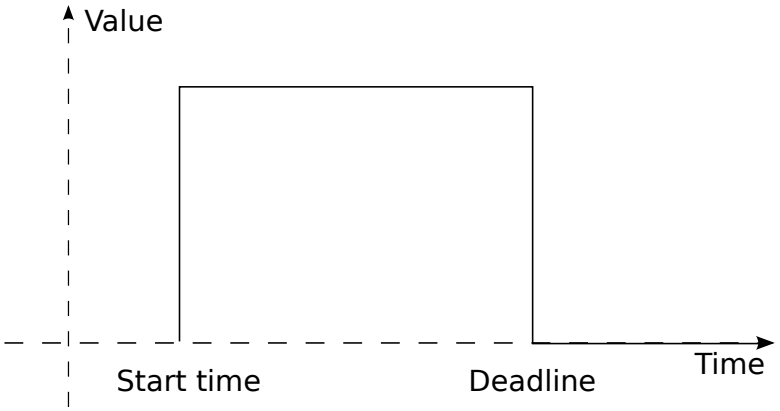
En kritisk deadline er en deadline som under alle omstændigheder skal overholdes for at opretholde systemets integritet. Såfremt en kritisk deadline overskrides vil der påføres skader på systemet som kan forårsage at systemets tilstand bliver udefineret. En kritisk deadline er illusteret på figur 4.1 på denne side.



Figur 4.1: Begivenhed med kritisk deadline.

Hard deadline

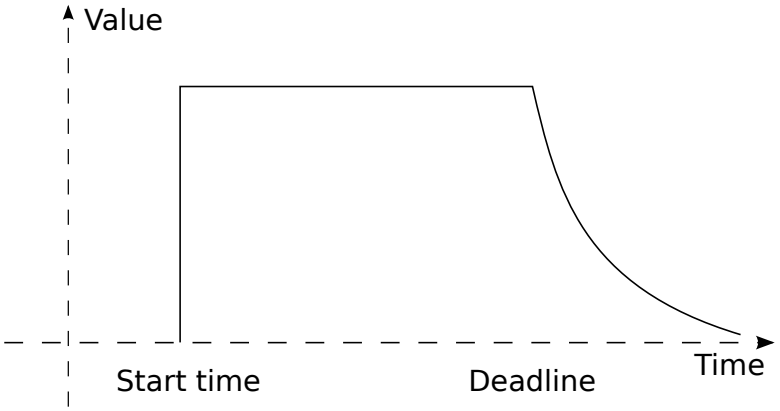
Vi definerer en begivenhed til at have en hard deadline såfremt en færdiggørelse af begivenheden efter deadlinen ikke tilfører systemet nogen positiv værdi. Modsat en kritisk deadline kan en overskridelse af en hard deadline accepteres. På figur 4.2 på denne side vises en hard deadline for en begivenhed.



Figur 4.2: Begivenhed med hard deadline.

Soft deadline

Færdiggørelse af en begivenhed med en soft deadline, før dens deadline tilføjer den samme værdi til systemet som hvis den havde haft en hard deadline. Forskellen ligger i den tilførte værdi såfremt deadlinen overskrides. Hvor en hard deadline ikke tilføjer nogen værdi ved en overskridelse, vil en overskridelse af en soft deadline stadig tilføre en reduceret værdi ved færdiggørelse. Den tilførte værdi vil være omvendt proportional med længden af overskridelsen. Figur 4.3 på denne side illustrerer en soft deadline.



Figur 4.3: Begivenheden med soft deadline.

Hard real-time system

Et hard real-time system er defineret ved et system der har begivenheder med hard deadlines, og kan garantere at disse ikke overskrides. Ydermere skal systemet være derterministisk, så denne garanti kan gives på forhånd. Det giver ikke mening at snakke om kritiske deadlines i hard realtime systemer da de kun adskiller sig fra hard deadlines med henblik på konsekvensen af en overskreden deadline, hvilket per definition ikke må ske i et hard real-time system.

Soft real-time system

Et soft real-time system kan indeholde alle typer deadlines men opstiller ikke nogen garantier for at de overholdes. Det vil typisk bruge en algoritme til at op- og nedprioritere hvilke deadlines der skal overholdes såfremt alle deadlines ikke kan overholdes.

Generelt vil der i et real-time system ikke være alle begivenheder der har den samme type af deadline. Nogle begivenheder har ingen deadline, nogle har en soft deadline, og få har en hard eller eventuelt en kritisk deadline.

4.1 Planlægning af begivenheder

I RTP tager planlægningen af hvilke begivenheder, der skal køres hvornår, udgangspunkt i det overordnede formål med systemet. Generelt vil formålet være at minimere antallet af overskredne deadlines, men dette er sjældent det eneste kriterium, der planlægges efter. Ofte vil prioriteten for en begivenhed, og tiden den tager at udføre, indgå i planlægningen. Eksempelvis vil man ofte prioritere at nå en kritisk deadline selv om det betyder at man overskrider to soft deadlines.

For at foretage en planlægning af begivenheder, der opfylder systemets formål bedst muligt, skal vi have så meget information om begivenhederne som muligt - jo mere vi ved om dem, jo bedre en planlægning kan der foretages. De relevante informationer i forhold til planlægningen er, hvornår en begivenhed forekommer, hvor lang tid den tager at udføre samt hvilken prioritet den har.

Ud fra den tilgængelige viden kan der foretages en statisk eller dynamisk planlægning[7]. Statisk RTP kræver, at vi har alle de nævnte informationer om alle begivenheder. Herved kan vi på forhånd foretage en fuldstændig planlægning og allerede inden start have klarlagt, om nogen deadlines vil blive overskredet. Såfremt vi ikke har alle informationer til rådighed, er vi nødsaget til at foretage en dynamisk planlægning. Dette vil ofte skyldes, at vi enten ikke ved hvornår en begivenhed forekommer, eller at vi ikke har information om hvor lang tid der tager at udføre den. I praksis vil det være svært at opstille eksakte værdier for hvor længe en begivenhed er om at blive udført, og der benyttes derfor ofte estimater.

I systemer, hvor begivenheder ikke er isolerede, men kan have interne afhængigheder, kan der opstå det problem, der hedder prioritetsinvertering[20]. Det opstår, hvis en begivenhed med høj prioritet er afhængig af en begivenhed med en lavere prioritet. Ikke nok med at begivenheden med høj prioritet må vente på begivenheden med lav prioritet, den må også vente på alle begivenheder med medium prioritet. Derved bliver begivenheden i praksis kørt med den lavere prioritet, da den ikke er klar, før begivenheden med lav prioritet er færdig. Problemstillingen kan vises klart med følgende eksempel. Vi har tre begivenheder (B_0, B_1, B_2) med prioriteterne $Pr_0 > Pr_1 > Pr_2$. Først udvælges B_0 da denne har højst prioritet, men stopper, da den er afhængig af kommunikation fra B_2. Den næste begivenhed, der udvælges vil være B_1, og dermed bliver B_0 unødigt forsinket mens B_1 kører.

Overordnet set findes der to metoder til at minimere prioritetsinvertering. Enten kan man prøve at forhindre de opstår eller minimere/kontrollere prioritetsinverteringen. Hvis prioritetsinverteringen kommer som en konsekvens af brugen af kritiske regioner, kan man hindre processer i at blive afbrudt mens de er i de kritiske regioner. Dermed vil begivenheden med høj prioritet ikke skulle vente

på begivenhederne med medium prioritet, men omvendt skal den altid vente på processen med lav prioritet, selvom den ikke ønsker at tilgå den kritiske region.

Er begivenhederne alle regelmæssige og det er kendt hvor lang tid hver begivenheden tager, kan man nægte at lade begivenheder tilgå kritiske regioner, hvis der er mulighed for de så blokere for begivenheder med højere prioritet.

Prioritetsnedarvning er en tredje metode til at kontrollere prioritetsinvertering[20]. Ved at benytte denne løsning vil en begivenhed med lav prioritet, som en begivenhed med høj prioritet er afhængig af, nedarve prioriteten fra den begivenhed der venter på den. Herved sikres det at begivenheder med høj prioritet ikke kommer til at vente unødigt på begivenheder med lavere prioritet.

4.1.1 Metoder til skemaplanlægning

Der findes adskillige metoder til at planlægge rækkefølgen af begivenheder. De adskiller sig fra hinanden med henblik på hvilke informationer de benytter og hvad de optimerer efter. Vi har valgt at kigge på "Rate monotonic algorithm"[15, 16] inden for statis skemaplanlægning, og "Earliest deadline first"[16] og "Least Laxity" indenfor dynamisk skemaplanlægning.

Hvor begivenheder isoleret set har behov for at udføre deres opgave inden deres deadline, har skemaplanlæggeren behov at foretage en vurdering af en proces' prioritet og organisere dem så den, til enhver tid, kan vælge hvilken proces, der skal udføres som den næste. Hovedformålet for en skemaplanlægger er derfor at gå fra en række processer med tilknyttet deadline og eventuelt andre egenskaber til en prioriteret liste.

Der fokuseres i litteraturen på om en algoritme er stabil eller ej. En stabil algoritme er defineret som at det altid er den begivenhed med lavest prioritet, der overskrides, hvis det ikke kan undgås at overskride en deadline. Dette er en ønsket egenskab ved en skeamaplanlægningsalgoritme, og kan bla. bruges til at definere en delmængde af begivenheder, for hvilke man kan garantere at de ikke overskrider deres deadline.

Rate monotonic algorithm (RM)

RM er en statisk skemaplanlægningsalgoritme, der tager udgangspunkt i at processer udføres periodisk. Den udregner fra start en prioriteret liste på baggrund af frekvensen af processerne, hvorved processer der ofte skal udføres vil få en højere prioritet end processer med lav frekvens. RM er todelt og i første del, før selve simuleringen, udregnes prioriteten for processerne, og der foretages en udvælgelse af hvilke processer der kan medtages i selve udførslen. Anden del står for udvæl-

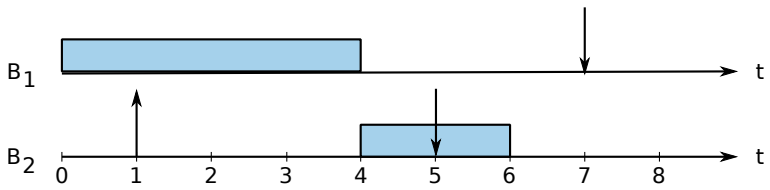
gelsen af processer under simuleringen, og her vælges simpelt den proces med den højeste prioritet.

Et problem for RM er at ved udvælgelsen af processer der kan medtages har man ikke en optimal udnyttelsen af processorkraft. Lehoczky, Sha og Ding er kommet frem til at "worst-case" er udnyttelsen 69% [15]. Et større problem i relation til implementering i PyCSP er dog at RM er dog at den er statisk. Til gengæld er algoritmen stabil ved en overskridelse af deadline for en proces.

Earliest deadline first (EDF)

Som alternativ til den statiske skemaplanlægger, hvor man ikke kan ændre prioriten af processen løbende under udførslen, findes de dynamiske skemaplanlæggerer, hvor EDF er et eksempel. Her evalueres prioriterne af processerne dynamisk under udførslen og evaluerer dermed løbende hvilke processer der skal udvælges. I EDF har den proces hvis deadline ligger først højest prioritet og den proces hvis deadline ligger længt ude i fremtiden har lavest prioritet. I EDF er antallet af prioriteter ikke fastlagt på forhånd og antallet, samt spændet mellem højeste og laveste prioritet, kan ændre sig dynamisk. Aperiodiske processer kan i EDF indgå på lige fod med de periodiske da man til hvert processkift evaluerer hvilken proces, der har den nærmeste deadline, som både kan være en periodisk og aperiodisk proces.

Udnyttelsen af processorkraft kan i EDF komme op på 100%, da alle processer bliver planlagt løbende i modsætning til RM der foretager et valg om en given proces kan planlægges. Ulempen ved EDF er den ikke er stabil idet vi ikke har kontrol over hvilke processers deadline der bliver overskredet. Dette er specielt et problem hvis man har en en uhomogen samling processer hvor en mindre del er kritiske. Såfremt skemaplanlæggeren ikke kan foretage preemptive skift mellem begivenheder kan EDF ikke give nogen garanti om at alle deadlines overholdes, selv om der findes en rækkefølge af begivenheder der sikrer dette. Dette er illustreret på figur 4.4 på denne side.



Figur 4.4: Overskridelse af deadline med EDF såfremt der ikke kan udføres preemptive kontekstskift. Begivenheden B_2 ankommer til tiden t_1 og har deadline ved t_5. Den kan dog ikke igangsættes før B_1 er færdig, hvorved B_2 overskrider sin deadline. Figuren er kopieret fra [6, s. 56]

Least Laxity(LL)

LL er en modifikation af EDF. LL kigger på hvor lang tid en begivenhed tager at udføre sammenholdt med hvor lang tid der er til deadline for begivenheden. Laxity er defineret som deadline minus tiden det tager at udføre begivenheden. Laxity bliver altså et udtryk for hvor presserende det er at igangsætte en begivenhed for at den kan nå sin deadline. I LL bruges dette til at prioritere de begivenheder der har mindst laxity højest når begivenhederne planlægges.

4.2 Realtidsplanlægning i PyCSP

Da vi ønsker at introducere RTP i PyCSP, er der er række forhold, som vi skal tage højde for. Vi vil i dette afsnit tage udgangspunkt i de ovenstående opstillede muligheder og sammenholde dem med PyCSP.

I PyCSP kan vi anskue processerne som begivenheder og derved bruge skemaplanlæggeren i greenlets-versionen til at styre hvilken proces og dermed hvilken begivenhed, der udføres. Vi har ikke umiddelbart informationer om, hvor lang tid en proces er om at blive udført, og det vil kræve en analyse af den enkelte proces at udlede estimater for det. Vi har valgt at fokusere på selve planlægningen af processerne og ikke på analysen. Dermed har vi ikke mulighed for at benytte RM og LL algoritmerne, da de begge benytter information om udførselstiden for en proces. Derved har vi EDF tilbage som mulighed, hvilket vi i det følgende vil implementere. En udvidelse af PyCSP til at benytte EDF er forholdsvis ligetil. Der skal laves en mulighed for, at udvikleren kan angive en deadline til en proces, og vi skal ved kontekstskift sikre, at vi aktiverer den proces, der har den førstkomne deadline. Såfremt en deadline overskrides, skal vi have funktionalitet, til at håndtere dette. I forbindelse med kommunikation, skal vi se hvordan processer med deadlines bliver påvirket af any-to-any kanalerne og det prioriterede valg i `alternation`.

Den eneste form for intern afhængighed der findes mellem processer i PyCSP er per definition i forbindelse med kommunikation over kanaler, og vi kan derfor opleve problemer med prioritetsinvertering. Den eneste metode til håndtering af dette, der kan bruges sammen med PyCSP, er prioritetsnedarvning, da de andre metoder forudsætter prioritetsinverteringen sker som resultat af tilgang til kritiske regioner og ikke pga. kommunikation. Vi skal derfor også implementere prioritetsnedarvning for at sikre os mod prioritetsinvertering.

Da vi benytter os af greenlets-versionen af PyCSP arbejder vi med en skemaplanlægger, der ikke kan foretage preemptive kontekstskift. Dette kan lede til problemstillinger som det er illustreret på figur 4.4 på forrige side. En metode til

at mindske denne type problemer er at lade de enkelte processer afgive kontrollen med jævne mellemrum. Herved vil der blive foretaget en ny evaluering af hvilken proces, der skal aktiveres, og hvis der er processer, der har en nærmere deadline, som er blevet klar, aktiveres en af disse. Dette beror helt og holdent på at hver enkelt proces frivilligt afgiver kontrollen og gør det med jævne mellemrum, når den er aktiv. Dette betyder, at det er overladt til udvikleren at indsætte det på relevante steder i processens kode.

4.2.1 Tilknytning og overskridelse af deadlines

Som udgangspunkt skal vi kunne håndtere, at en proces kan have forskellige typer deadlines. Umiddelbart er det oplagt, at der for hver proces tilknyttes en deadline samt hvilken type, det er. Dette giver mulighed for, at vi kan differentiere i den måde vi håndterer en overskreden deadline. F.eks. kan vi stoppe systemet helt i tilfælde af overskridelse af en kritisk deadline, kaste en exception ved en hard deadline og blot registrere overskridelser af soft deadlines. Uanset hvilken handling vi vælger at tilknytte til de respektive overskridelser, vil der altid være situationer, hvor den valgte handling ikke er optimal.

Vi har derfor valgt en anden løsning, hvor en proces blot kaster en exception, hvis den overskrider en deadline. Dette overflødiggør, at der tilknyttes en specifik type deadline til en proces, da håndteringen af den kastede exception overgives til udvikleren. Det er herved op til den enkelte udvikler at definere, hvad der skal ske i hver enkelt proces, såfremt den overskrider en deadline. Dette giver den største frihed til at tilpasse håndteringen til den enkelte proces og applikation.

4.2.2 Udvælgelse af proces

Udvælgelsen af hvilken proces der skal aktiveres, ved et kontekstskift, er givet ud fra vores valg af EDF. Som beskrevet i "EDF" i afsnit 4.1.1 på side 42, specificerer EDF, at det altid processen med den nærmeste deadline der skal aktiveres. For at opnå dette kan vi benytte stort set samme metode som vi brugte i DES. I DES sorterede vi processerne efter hvilket tidskridt de skulle aktiveres i, her kan vi sortere dem ud fra deres deadline.

4.2.3 Kanalkommunikation

I litteraturen for RTP fokuseres der kun på udvælgelsen og planlægningen af processer i skemaplanlæggeren. I PyCSP er kommunikation blokerende, hvilket betyder at en proces er blokeret fra den ønsker at kommunikere, til kommunikationen er gennemført. Ved one-to-one kanaler, er der kun en proces i hver ende,

og derfor er processen garanteret at gennemføre kommunikationen når modparten er klar til at kommunikere.

I PyCSP er one-to-one kanalerne blevet fjernet, og erstattet af any-to-any kanaler, der ud fra et teknisk synspunkt er mere generelle og kan alt som one-to-one kanalerne kan. I any-to-any kanalerne er man ikke begrænset til kun at have en proces i hver ende af kanalen, men der kan være et vilkårligt antal processer. Et problem ved any-to-any kanaler, er man ikke som ved one-to-one kanaler er sikret på at kommunikere blot fordi der er en partner, da parringen mellem processer der ønsker at kommunikere sker tilfældigt. For at prioritere de processer der har højst prioritet, ønsker vi at det altid er den proces med højst prioritet der kommer til at kommunikere først. Det skulle eliminere problemet med den tilfældige parring af processerne.

4.2.4 Prioritetsnedarvning

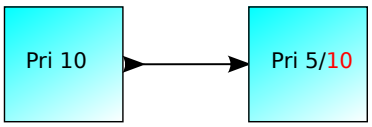
At introducere prioritetsnedarvning i PyCSP virker umiddelbart ligetil, idet vi kan se de indbyrdes afhængigheder klart ud fra forbindelserne gennem kanalerne. Der kan forekomme andre afhængigheder, som hvis man introducere delt data, men vi mener ikke man bør dele data i velskrevne CSP applikationer, og vi har derfor valgt at begrænse os til afhængigheder repræsenteret vha. kanaler. På trods af den umiddelbare simplicitet skal vi overveje hvornår, hvem og hvor længe, der skal arves i forbindelse med prioritetsnedarvning.

Ændring af prioritet

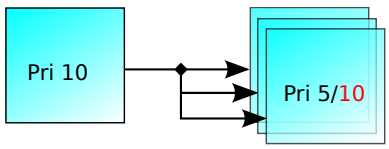
Vi arbejder med et dynamisk system af processer, hvor en proces, på baggrund af sin tilstand, er afhængig af forskellige andre processer for at kunne arbejde videre. Hvis vi højner prioriteten på alle processer, der er forbundet til en proces med høj prioritet, vil mange af processerne, der arver den høje prioritet, reelt ikke kunne bidrage til udførelsen af den proces, der starter med høj prioritet. Det er en bedre løsning, at det kun er den eller de processer, der kan sikre videre udførsel af den aktuelle proces, der tildeles højere prioritet. Hvis eksempelvis en proces med høj prioritet ønsker at skrive på en kanal, skal alle processer, der læser på den kanal, arve den høje prioritet, men de processer, som ønsker at skrive til en anden kanal, som processen med høj prioritet læser fra, ikke skal arve den høje prioritet. Generelt set betyder det, at der kun skal udføres prioritetsnedarvning, såfremt en højt prioriteret proces venter på at kommunikere, såfremt der ikke er andre processer, der er klar til at indgå i den ønskede kommunikation.

Vi har nu begrænset os til, at det kun er de processer der kan indgå i kommunikation over en kanal, der arver en prioritet. Med any-to-any kanaler findes der

dog et vilkårligt antal processer i hver kanalende, og man risikerer en prioritetsdevaluering ved at lade flere processer arve en høj prioritet. I klassisk CSP findes der modsat hertil one-to-one kanaler, hvor man er sikret at det kun én proces' prioritet der bliver højnet ved prioritetsnedarvning. Forskellen illustreres af figur 4.5 og figur 4.6 på denne side. På figur 4.5, bliver en proces' prioritet hævet fra fem til ti. Modtageren kan uden afbrydelser arbejde hen mod at kunne kommunikere, da afsenderen venter. I figur 4.6 bliver alle tre processers prioritet hævet til ti og de vil skulle kæmpe mod hinanden for komme frem til en tilstand hvor de ønsker at kommunikere.



Figur 4.5: Procesnetværk med en afsender og en modtager. Afsenderen har en prioritet på 10, mens modtageren har en initiel prioritet på fem. Modtageren får via prioritetsnedarvning hævet sin prioritet til 10. (Højere er bedre)



Figur 4.6: I dette eksempel findes der tre modtagere, som alle får hævet deres prioritet til 10.

Prioritetsnedarvning i et miljø med any-to-any kanaler har dermed en risiko for at medføre prioritetsdevaluering. Man kan forstille sig forskellige metoder til at eliminere eller minimere problemet f.eks ved at begrænse antallet af processer, der kan modtage en prioritetsnedarvning. Dette kunne gøres ved kun at lade en enkelt proces arve prioriteten, men så skal man tage stilling til, hvilken proces, der skal udvælges. Dette kunne være den proces, der er tættest på at indgå i kommunikationen. Udvælgelsen af processer må dog bero på en analyse af den enkelte applikation og dens aktuelle tilstand. Da vi ikke ønsker at begynde på en analyse af udviklerens kode, må vi nødvendigvis sende prioritetsnedarvningen til alle processerne på kanalen. For at løse problemet vil vi i stedet fokusere på at minimere antallet af processer, der starter prioritetsnedarvningen. Dette vil vi gøre ved bla. at se på om processens deadline allerede er overskredet. Vi vil ligeledes minimere det tidsrum processerne har arvet en prioritet, ved straks efter at kommunikationen har foregået at fjerne prioritetsnedarvningen, for alle processerne der havde arvet en prioritet.

Når kommunikationen på kanalen er gennemført, befinder processens sig i en anden tilstand og er afhængig af noget andet for at komme videre i sin udførsel. Når det midlertidige afhængighedsforhold ophører skal processer der har arvet en prioritet miste denne. Der skal selvfølgelig tages højde for at en proces kan arve forskellige prioriteter fra forskellige andre processer, så det skal være muligt at falde tilbage til den næsthøjeste arvede prioritet, i stedet for blot at skifte tilbage til den oprindelige prioritet.

4.2.5 Alternation

Det er dog ikke kun i forbindelse med almindelig kommunikation vi skal forholde os til introduktionen af processer med prioritet. I kodestrukturen `alternation` har en udvikler mulighed for at foretage et prioriteret valg mellem flere forskellige kanaler. Når man kommunikerer vil man opfylde en proces' ønske om kommunikation, og vi ønsker at opfylde kommunikationen for den proces der har den højest prioritet. Det prioriterede valg i en `alternation` kan derfor komme i konflikt med dette ønske.

Burns og Wellings beskriver og illustrerer præcist denne problemstilling i „The notion of priority in real-time programming languages"[5]. Til at illustrere problemet beskriver de et eksempel, som er vist i kodeuddrag 4.1 på denne side. Dette kodeuddrag viser et prioriteret valg mellem kanalerne A1 og A2. Til hver kanal er tilknyttet en proces, P1 og P2. Disse to processer har en prioritet tilknyttet på hhv. Pri1 og Pri2. Tabel 4.1 på denne side viser hvilken kanal der bliver valgt, afhængig af processernes prioritet.

Kodeuddrag 4.1: (priority) select. Eksemplet er kopieret fra [5]

```
(priority) select
   A1 -- Process P1
or
   A2 -- Process P2
end select
```

	Pri1 > Pri2	Pri1 = Pri2	Pri1 < Pri2
priority select	A1	A1	?

Tabel 4.1: Konflikten ved brug af prioriteret valg og procesprioriteter. Eksemplet er kopieret fra [5, s. 160]

Man kan af tabel 4.1 på denne side konstatere at der opstår en konflikt i kolon-

ne tre, hvis udvikleren foretrækker en kanal, hvor den tilknyttede proces' prioritet er lavere end den anden proces' prioritet. Man vil ikke kunne opfylde begge krav om både at kommunikere med den proces med højst prioritet, og lade udvikleren bestemme kanalen. For Burns og Wellings er løsningen to "orthogonal solution", der håndterer begge typer prioriteter. De ønsker overordnet set en weak- og strong select. Weak select sorterer primært efter processernes prioritet og sekundært efter det prioriterede valg. Strong select udvælger udelukkende processer efter det prioriterede valg. De forstiller sig, at weak select skal bruges som den primære metode, men i specielle tilfælde skal en udvikler have mulighed for at tvinge et prioriteret valg igennem.

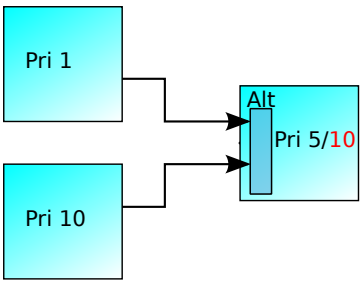
Artiklen beskæftiget sig kun med one-to-one kanaler, og det er denne antagelse der medfører at et valg af kanal medfører et valg af en proces og dennes prioritet. Dette er ikke en mulighed i PyCSP hvor kanalerne er af typen any-to-any. Et valg af en kanal, medfører derfor ikke en direkte kobling til en proces, men til vilkårligt mange processer, der kan have forskellige prioriteter. I afsnit 4.2.3 på side 44 kommer vi frem til at kommunikationen altid skal sørge for at det er den højst prioriterede proces der kommunikerer. Dette kan bruges i `alternation` til at finde den højst prioriterede proces, for hver kanal, der er klar til at kommunikere og udvælge den kanal der har den højst prioriterede proces.

På baggrund af artiklen har vi valgt at `alternations` udfører en weak select, da denne version egner sig bedst til RTP. Vi vil i vores version ikke inkludere en strong select, ud fra en betragtning om at den kun bør bruges i sjældne tilfælde, da den vil modarbejde ideen bag RTP. Hvis en udvikler alligevel ønsker denne funktionalitet, vil han godt kunne opnå det i PyCSP, men vi ønsker ikke at tilskynde dette, ved inkludere det i RTP versionen af PyCSP.

Prioritetsnedarvning i `alternation`

Vi har som nævnt i afsnit 4.2.4 på side 45 en klar kæde af afhængigheder i PyCSP, men vi skal være opmærksomme på ikke at højne processers prioritet unødigt i forbindelse med prioritetsnedarvning. Dette kan let blive tilfældet såfremt vi ikke holder ordentligt styr på, hvorfor en proces har den prioritet, den har, om den er sat af udvikleren, eller den er nedarvet. Man kan forestille sig en situation, hvor et uddrag af et proces-neværk består af en generator-forbruger-model med to generatorer og en enkelt forbruger. De to generatorer er forbundet til forbrugeren vha. en `alternation`, og har henholdsvis høj og lav prioritet. Eksemplet er illustreret på figur 4.7 på næste side. Forbrugeren vil i dette scenarium arve den høje prioritet fra den tilsvarende generator, men utilsigtet vil den høje prioritet derefter også propagere fra forbrugeren til generatoren med lav prioritet. Dette

er ikke hensigtsmæssigt, da de to generatorer nu har lige høj prioritet og ikke det forhold, som udvikleren oprindeligt har angivet. Vi kan dog indse, at dette ikke bliver et problem, idet vi kun udfører prioritetsnedarvning i det tilfælde, hvor der ikke er nogen processer, der er er klar til at indgå i ønsket kommunikation. I det opstillede tilfælde vil forbrugeren korrekt modtage en prioritetsnedarvning, men vil aldrig selv foretage en yderligere prioritetsnedarvning på generatoren med lav prioritet. Dette skyldes at generatoren med høj prioritet nødvendigvis må være klar til at sende data, siden den selv tidligere, har givet sin prioritet til forbrugeren.



Figur 4.7: Prioritetsnedarvning i `alternations`.

4.3 Slagterieksempel

Til at illustrere brugen af RTP, vil vi tage udgangspunkt i et aktuelt problem fra Danish Crown slagteriet. Slagteriet skal udvikle en beslutningsmodel for, hvordan en robot, skal udskære hver gris, men denne beslutning skal foretages inden grisen når robotten, hvorfor vi har et helt klassisk RTP problem. På Danish Crown slagteriet i Horsens foretages udskæringen af grise, som beskrevet på deres hjemmeside:

> *"Grisen [. . .] skal nu skæres i mindre, håndterbare stykker. Det sker i en meget avanceret maskine – en såkaldt tredeler – hvor hver halvdel af grisen deles i tre stykker: bov, mellemstykke og skinke.*
>
> *Robotten starter med at fotografere hver halvdel. Dataene fra billedet kombineres med ordren og kundens ønsker, hvorefter stykket deles i tre - nøjagtigt afpasset kundens ønsker."* Danish Crowns hjemmeside[1]

Et billede af den automatiske tredeler er vist på figur 4.8 på den følgende side. Problemet med den nuværende maskine er, at den ikke kan tage højde for

[1] http://www.danishcrown.dk/custom/horsens/3772.asp

den enkelte gris' anatomi. Omtrent 10% af grisene har et ekstra sæt ribben, og slagteriet vil derfor gerne ændre deres udskæring til at tage højde for om dette ribben er til stede.

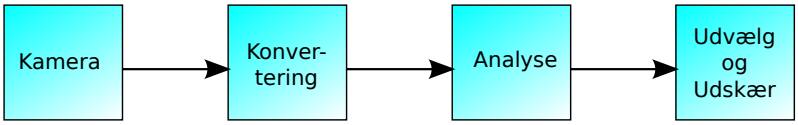
Figur 4.8: Billedet viser i forgrunden et foto taget af tredeleren til brug for analyse. I baggrunden ses transportbåndet, hvor de halve grise venter på på at blive udskåret af den automatisk tredeler.

Slagteriet har placeret kameraet i starten af et transportbåndet mens udskæringsrobotten findes i den anden ende. Der kan være flere grise på transportbåndet på samme tid, og det fremfører grisene i et fast tempo. Dette giver et fast tidsrum fra grisen passerer kameraet til det passerer robotten. Vi har hermed et klassisk RTP system, hvor robotten skal foretage et valg under en hard deadline, da der skal foretages en udskæring. Før vi begynde implementeringen skal vi opdele den samlede proces i logisk afgrændsede skridt.

Vi har opdelt Arbejdsgangen i følgende skridt:

1. Et billede bliver taget af grisen mens den passerer kameraet.
2. Billedet konverteres til en 3D-model af grisen.
3. 3D-modellen analyseres.
4. Robotten udvælger hvor udskæringerne skal være på baggrund af analysen, ordren og kundens ønske.
5. Robotten udskærer grisen.

Man kan se at arbejdsgangen indeholder en række klart afgrænsede arbejdsområder, som med fordel kan modelleres som selvstændige processer i PyCSP. Vi har derfor valgt implementere følgende processer: Kamera, Billedekonvertering, 3D-analyse og en Udskæringsproces, hvilket leder til et procesnetværk som vist i figur 4.9 på denne side.

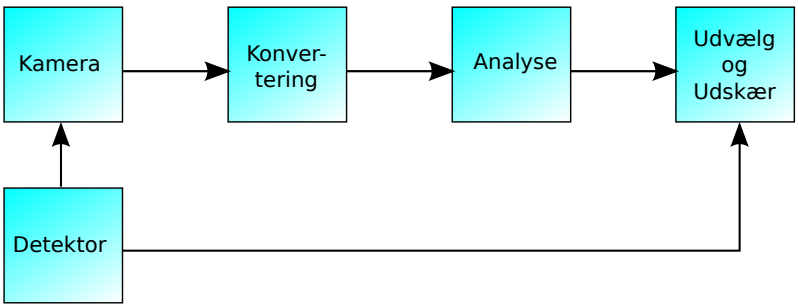


Figur 4.9: Procesnetværk til udskæring af grise på et slagteri.

Implementering

Til at implementere eksemplet i PyCSP, kan vi oprette hver gris som et objekt og tilknytte et tidspunkt som vi kan bruge som en deadline. Med denne kan hver proces evaluere om grisen har overskredet sin deadline, og i det tilfælde fjerne griseobjektet, og stoppe den videre behandling af det. Da det er ikke angivet hvordan hele processen startes, antager vi der findes en form for detektor foran kameraet, der opfanger når en gris passerer og som dermed starter processen.

Når detektoren starter hele processen, opretter den griseobjektet, som den sender til kameraprocessen samt sender en kopi direkte til udskæringsprocessen. Dermed ved processen, at der ankommer en gris, som den skal udskære, og hvis den inden deadline, får en analyse af grisen, kan den træffe et begrundet valg om hvordan udskæringen skal foretages. Hvis ikke denne analyse når at blive klar, bruges blot standardmodellen til at udskære grisen. Figur 4.10 på denne side viser et netværk, hvor detektoren er introduceret, og som sender data til hhv. kameraprocessen og til udskæringsprocessen.



Figur 4.10: Procesnetværk med detektor til initiering af hver gris.

Vi har holdt os tæt op af den virkelige verden, i designet af implementeringen

men da dette er et delvist tænkt eksempel, har vi i sagens natur ikke adgang til slagteriet og deres maskiner, eller præcis data om grisene. Derfor må vi nødvendigvis simulere store dele af eksemplet.

De enkelte processer foretager derfor ikke et konkret stykke arbejde, men har tilknyttet et tal der repræsenterer det tidsrum, som vi forventer arbejdet i processen vil tage. Hver proces simulerer i stedet arbejdet, i et normalfordelt tidsrum omkring det tilknyttede tidsrum.

Detektoren starter processeringen af hver gris. Derfor står detektoren for oprettelsen af griseobjektet, hvori vi også definerer om den har et ekstra sæt ribben.

Det er i eksemplet krævet, at grisen bliver udskåret mens den er indenfor robottens rækkevidde, hvorfor det er uacceptabelt hvis ikke udskæringsprocessen er aktiv i tidsrummet hvor grisen er inden for rækkevidde af robotten.

Hvis eksemplet implementeres i greenlets-versionen, kan kun en proces være aktiv ad gangen, og processer kan være aktive, så længe de ønsker. Dermed skal kamera-, konverterings- og analyseprocesserne frivilligt stoppe deres arbejde mens udskæringsprocessen arbejder. Vi har dog ikke kendskab til hvordan de processer kommer til at arbejde, og kan dermed heller ikke sige om det vil være muligt at afgive kontrollen med regelmæssige mellemrum. Selv hvis dette er muligt, findes der ikke i PyCSP mulighed for at processen stopper sit arbejde for at andre processer kan komme til. Det bedste man vil kunne gøre, er at benytte en `alternation` sammen med en `timeout`, for at tvinge processen til at vente et tidsrum, hvor andre processer så vil kunne komme til. Dette vil dog samtidigt sænke ydelsen af programmet da processen dermed er tvunget til at vente i tidsrummet angivet i denne `timeout`, selvom der ikke er andre processer klar. Hvis man vil undgå brugen af `timeout`, kan processerne deles op i to separate applikationer. En applikation skrevet i PyCSP der skal stå for konvertering og analyse mens en anden skal styre robotten. De to applikationer skal kunne kommunikere og udveksle data, f.eks. igennem en database, harddisk eller anden delt datastruktur. Hvis analysen bliver færdig gemmes den i den delte datastruktur og applikationen der styrer robotten kan udnytte analysen. Hvis ikke den er klar bruges i stedet standardmodellen. Som beskrevet i kapitel 2 og som vi også kom ind på i implementeringen i afsnit 3.1, strider en delt datastruktur dog mod ideerne i CSP, hvorfor det ikke er optimalt, men dog bedre end at sænke ydelsen af systemet ved at indsætte tvungne `timeouts`.

I stedet for at vælge greenlets-versionen kunne man vælge processes-versionen. Hermed vil man kunne køre processer samtidigt, og udnytte operativsystemet til at foretage preemptiv afbrydelse af processerne, således at alle processer samtidigt får en del CPU-tid. Dermed kan alle processer kører samtidigt og robotten

kan foretage udskæringen. Processes-versionen risikerer dog at operativsystemet foretager et preemptivt processkift, mens udskæringsprocessen kører, således at konvertering og analyseprocessen også kan arbejde. Potentielt vil dette kunne resultere i at grisen ikke bliver udskåret, selvom griseobjektet ankommer rettidigt til udskæringsprocessen.

Med tanke på de ovenstående fordele og ulemper der findes ved hhv. greenlets-versionen og processes-versionen, har vi valgt at implementere begge versioner, for i evalueringen, at sammenligne dem med RTP-versionen.

I greenlets-versionen er udskæringsprocessen reduceret til en IO-proces der modtager griseobjekter og gemmer dem, så en anden applikation kan tilgå dem. Til dette har vi valgt at bruge en `dictonary` datastruktur til at gemme hver griseobjekt under deres unikke id. Første gang processen modtager et griseobjekt vil det være ubehandlet, men gemmes i datastrukturen for at indikere at et tilsvarende objekt er ved at blive behandlet i proces-netværket. Såfremt griseobjektet bliver færdigbehandlet indenfor sin deadline, vil det færdigbehandlede griseobjekt modtages fra analyseprocessen, og overskrive det ubehandlede griseobjekt. Hvis det ikke når at blive færdigbehandlet, vil den anden applikation stadig kunne tilgå det ubehandlede griseobjekt, og derved vide at en gris skal udskæres.

Processes-versionen, er magen til greenlets-versionen, men den praktiske forskel at processerne kører som selvstændige python-processer og derfor har muligheden for parallel kørsel.

4.4 Implementering

Vi vil i dette afsnit beskrive hvilke ændringer og tilføjelser vi skal foretage i PyCSP, for at implementere RTP. Ændringerne vil tage udgangspunkt i de emner, vi har diskuteret i foregående afnit med fokus på de problemstillinger der skal tages højde for ved implementeringen af dem.

4.4.1 Overskredne deadlines

Vi har i foregående afsnit argumenteret for, at alle overskredne deadlines bør resultere i en exception. Dette er oplagt at implementere i skemaplanlæggeren, så det checkes ved kontekstskift, om en deadline for den proces der skiftes fra, er overskredet, og i givet fald, kaster en exception. Vi ønsker dog at få kastet vores exceptions så hurtigt som muligt, for derved at gøre opmærksom på den overskredne deadline. Derfor checker vi yderligere for overskredne deadlines, når der kommunikeres på en kanal, og når der foretages et valg i en `alternation`.

4.4.2 Ændringer i skemaplanlæggeren

I greenlets-versionen af skemaplanlæggeren findes der som nævnt i afsnit 3.2.2 på side 20 tre lister af processer: `new`, `next` og `timers`. De tre lister er prioriteret således, at der først kigges på processer fra `timers`, dernæst fra `new` og til sidst kigges der i `Next`.

I RTP er det ikke hensigtsmæssigt at inddele processerne i disse tre kategorier. Vi skal derimod have et miljø, der gnidningsløst tillader processer både med og uden deadlines, samt at de dynamisk kan ændres. Skemaplanlæggeren skal i forbindelse med processkift hurtigt kunne finde den næste proces, der skal udføres.

Vi har derfor valgt at fjerne de tre lister og erstatte dem med `has_priority`, `no_priority` og `timers`. `has_priority` og `no_priority` benyttes til aktive processer, der ønsker at blive udført, mens `timers` er en kopi af DES versionen.

Det er vigtigt at bemærke ifht. processer der ligger i `timers`, at udvikleren ikke kan forvente at de bliver aktiveret på de eksakte tidspunkt han har defineret. Dette er kun muligt i DES versionen hvor vi kan kontrollere tiden. Den eneste garanti der gives, når vi arbejder med realtid, er at de tidligst aktiveres på det angivne tidspunkt. I greenlets-versionen aktiveres først processer fra `timers` listen. Dette gøres fordi processer i denne version kun kommer på denne liste via `timeout`. En udvikler vil forvente at processen venter i præcist det tidsrum man har angivet for så at fortsætte. For at emulere dette krav om kun at vente et præcist tidsrum foretrækkes derfor processer fra denne liste fremfor processer der bare ønsker at bliver kørt. I RTP antages det, at der findes en mængde processer, der skal gennemføres inden en deadline, hvorfor de må kæmpe om CPU-tid. En proces, der har ventet i `timers` listen skal derfor ikke nødvendigvis udføres med det samme, da det hele tiden bør være den proces med den højeste prioritet der skal udføres, uafhængigt af processerne i `timers` hoben. Processerne, der ikke længere skal vente på timeout, bliver derfor planlagt og udvalgt på lige fod med andre processer der er klar til at blive udført.

Til at implementere `has_priority` bruger vi også en hob, men da modulet `heapq` kun understøtter min-hobe kan vi ikke lave en klassisk prioritetshob, da den skal kunne udtrække processen med maksimal prioritet. Vi har dermed to muligheder, enten kan vi lave vores egen implementering af en maks-hob, eller også kan vi ændre vores prioriteter internt, så en lav værdi angiver en høj prioritet. Med en egen implementering har vi en logisk opbygning af prioriteter, men vi får ikke fordelen ved den underliggende implementering direkte i C, som man opnår ved brug af modulet heapq. Vælger vi at bruge dette, skal vi invertere prioritetsbegrebet, så det er den laveste prioritet, der udvælges først. Dette viser sig dog ikke at være et problem i vores tilfælde, da vi ønsker at benytte os af

en EDF algoritme og derfor nemt kan opnå den ønskede effekt ved at bruge en proces' deadline som dens prioritet. Her vil en lav deadline betyde, at processen snart skal være færdig, hvilket resulterer i en høj prioritet. Vi kan derfor blot benytte en proces' deadline som dens prioritet og benytte en min-hob.

4.4.3 Preempting

Som vi har beskrevet i afsnit 4.2 på side 43, kan man uden preempting risikere, at en proces med lav prioritet kan blokere for en proces med høj prioritet. Her konkluderede vi at det er udviklerens opgave at processen afgiver kontrol, og derfor skal det være nemt at afgive kontrollen for processen. Til dette har vi lavet funktionen `Release()`, der minder om `Yield` for `co-rutiner`.

Implementeringen er meget simpel og er blot en wrapperfunktion, da den underliggende funktionalitet allerede eksisterer. Den aktive proces stopper og bliver genplanlagt til senere kørsel af skemaplanlæggeren. Dermed lægges processen på den relevante kø, og skemaplanlæggeren får mulighed for at vælge en ny proces der skal udføres. Er der ikke kommet nye processer, vil det stadig være den originale proces, der vælges og kan fortsætte sin kørsel. Hvis der derimod er ankommet en eller flere nye processer i mellemtiden, som har højere prioritet, vil disse blive valgt i stedet.

Problemet ved denne tvungne procesafgivelse er, at det kan tage lang tid at lægge processerne i en min_hob, som vil være spildt, hvis den alligevel med det samme fjernes fra køen. Man vil derfor nok i en senere version kunne optimere hastigheden af `Release()`.

4.4.4 Udvidelse af `Process`

Hver proces skal kunne tilknyttes en deadline, som er et tidsstempel, der angiver det tidspunkt, som udvikleren ønsker at processen skal være færdig inden. Desuden skal hver proces tilknyttes en prioritet. Denne prioritet bruges af skemaplanlæggeren til at udvælge hvilken proces der skal eksekveres. I EDF er prioriteten og deadline for en proces som den samme, og prioriteten er derfor også et tidsstempel.

I forbindelse med prioritetsnedarvning kan en proces midlertidigt få ændret sin prioritet, hvilket vi diskuterer yderligere i afsnit 4.4.6 på side 59. For at kunne adskille prioriteten, der ikke altid er sat af udvikleren, og en deadline, der altid er sat af udvikleren, har vi valgt at holde de to variable adskilt.

For at en udvikler kan tilknytte en deadline til en proces har vi introduceret funktion `Set_deadline`. Denne funktion har to parametre, et tal, der angiver tiden til deadline og en proces. Processen er en valgfri parameter, da funktionen antager at det er processen selv der ønsker at sætte en deadline. Hvis en ud-

vikler ønsker at sætte en deadline for en proces umiddelbart efter oprettelsen af processen, bruges samme funktion, men med processen som en parameter.

Til at angive at processen har nået sin deadline introducere vi funktionen `Remove_deadline`, der står for at fjerne en givet deadline fra enten processen selv, eller den proces der gives som en valgfri parameter.

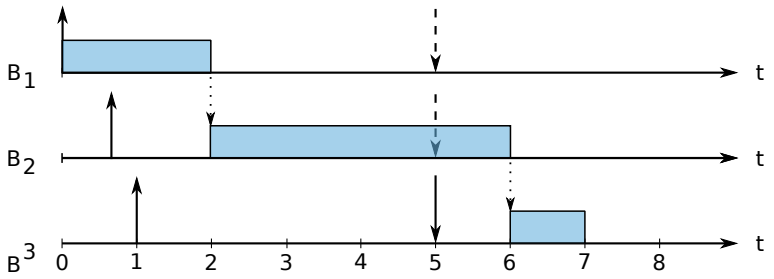
Selvom det ikke umiddelbart er nødvendigt har vi også introduceret funktionen `Get_deadline`, der som navnet antyder returnere den givne deadline for en proces.

Når en proces bliver udvalgt til at arve en prioritet gennem prioritetsnedarvning, skal skemaplanlæggeren planlægge processen ifht. den nye prioritet. Den nye prioritet er også et tidsstempel, og hvis ikke processen er færdig, inden denne prioritet er overskredet, vil en anden proces' deadline være overskredet. Vi kan derfor vælge at `RTP` skal kaste en `deadlineException`, hvis prioriteten overskrides. Ved at kaste en `deadlineException` i processer hvor prioriteten er overskredet, kan udvikleren se præcist hvilken proces, der var aktiv, og dermed se hvorfor den originale proces også kaster en `deadlineException`.

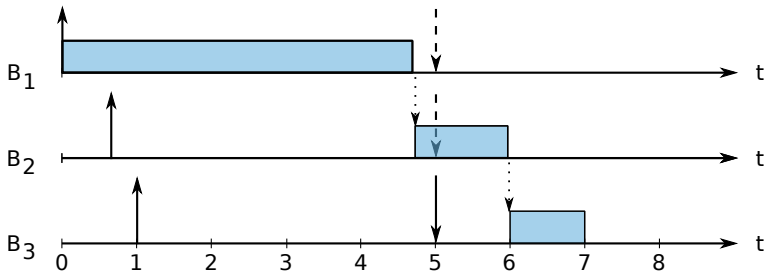
Figur 4.11 på modstående side viser et tidsdiagram for et generator-arbejder-forbruger-netværk bestående af tre processer B_1, B_2 og B_3. B_3 har den højeste prioritet, og B_2 arver denne prioritet. Hvis B_2 også kaster en `deadlineException` vil det tydeligt fremgå for udvikleren, at det er B_2 der bærer skylden for, at B_1's deadline ikke blev overholdt.

En anden begrundelse for at lade en nedarvet prioritet medføre en `deadline-Exception` er hvis processerne er afhængige af hinanden. I figur 4.11 på næste side nedarver arbejderprocessen(B_2) og generatorprocessen (B_1) en prioritet fra forbrugerprocessen (B_3). Hvis denne deadline ikke nås, er det data som arbejderprocessen bearbejder ikke længere relevant, og arbejderprocessen kan med fordel stoppe det irrelevante arbejde. I eksemplet ville arbejderprocessen (B_2) kunne stoppe sit arbejde til tiden $t = 5$ i modsætning til at fuldføre arbejdet, og stoppe i tiden $t = 6$.

Der er dog ikke sikkert at en deadlineException i processer der har nedarvet en prioritet, er med til klarlægge hvilke processer der har brugt al tiden, og derfor bærer skylden for at en deadline ikke blev overholdt. Figur 4.12 på modstående side viser et eksempel på dette. Netværket er opsat som i figur 4.11, men i dette tilfælde bruger generatorprocessen (B_1) al tiden, og data bliver først sendt fra B_1 umiddelbart før en overskridelse af deadlinen. For en udvikler vil det fremgå, som var det arbejderprocessen (B_2), der er ansvarlig for overskridelsen ligesom i figur 4.11, og ikke generatorprocessen B_1, som i dette eksempel brugte det meste af tiden. Dermed mister `deadlineException` sin troværdighed, og brugbarhed



Figur 4.11: Et generator/arbejder/forbruger -netværk. Kasserne repræsentere det tidsrum hvor processerne bliver bearbejdet. En pil op indikere hvornår processen er klar til at blive eksekveret. En pil ned indikere en deadline for processen. De stiplede pile i proces B_1 og B_2 til tiden t= 5 viser en kunstig prioritet på baggrund af B_3's deadline. Den lille stiplede pil mellem B_1 og B_2 i t= 2 og mellem B_2 og B_3 i t= 6 viser kommunikation mellem processerne.



Figur 4.12: Samme netværk som i figur 4.11, men i dette tilfælde venter B_2 på data fra B_1 i hovedparten af tiden inden en deadline.

til at identificere hvor i netværket tiden bruges.

Et andet problem ved at lade den aktive proces kaste en `deadlineException` er, at det vil pålægge udvikleren et væsentligt større arbejde med at håndtere disse exceptions. Såfremt vi implementerer det, kan enhver proces, der kan arve en prioritet via prioritetsnedarvning, kaste en exception. Det er ikke nødvendigvis klart gennemskueligt hvilke processer det vil være, hvorved udvikleren kan have svært ved at sikre ordentlig fejlhåndtering. Yderligere vil det medføre at excpetions kan kastes udenfor den kontekst de er relateret til, hvorved det kan være umuligt at håndtere dem korrekt.

På baggrund af de opstillede fordele og ulemper, har vi valgt at kun processer med en eksplicit deadline, har mulighed for at kaste en `deadlineException`. Processer, der nedarver en prioritet, bliver planlagt i henhold til den højeste prioritet, de har, og vil altid gøre arbejdet færdigt. En proces skal dermed kunne adskille sin egen deadline fra den prioritet, som den skal planlægges med, selv om de to værdier i en stor del af tiden vil være det samme.

En deadline er dermed en variabel der kun kan sættes af udvikleren og det er kun på baggrund af denne deadline at processen skal kaste en `deadlineException`.

For at skemaplanlæggeren kan udvælge processer introduceres prioritet, der som standard er det samme tal som deadlinen. For at kunne håndtere flere niveauer af prioritetsnedarvning, gemmes prioriteten i en liste kaldet `inherit_priotity`. Denne liste af prioriteter indeholder indledningsvis kun en prioritet som er deadlinen sat af udvikleren. Når andre processer midlertidigt ønsker at ændre en proces' prioritet, tilføjes den til listen. Ved at bruge en liste i stedet for blot en variabel, har processen mulighed for at blive opprioriteret flere gange og derefter trinvist vende tilbage til de tidligere niveauer.

Når skemaplanlæggeren placerer processen i hhv. `has_priority` og `no_priority` hobene, bruges blot den mindste prioritet i listen af nedarvede prioriteter ihht. vores implementering af skemaplanlæggeren. Dette medfører, at når en proces efterfølgende får ændret sin liste af prioriteter, skal processen genplanlægges for at sikre, at den placeres korrekt i min-hoben i skemaplanlæggeren.

4.4.5 Kanaler

I PyCSP findes der kun kanaler af typen `Any-To-Any`, og derfor kan der altid være et vilkårligt antal kanalender i hver ende af kanalen, der kan være klar til at kommunikere. Vi skal derfor foretage en ændring, så kommunikationen mellem kanalenderne altid foregår mellem de højst prioriterede processer.

I greenletsversionen foregår udvælgelsen af kanalender til kommunikation ved hjælp af funktionen `match`, der udnytter at hver kanal vedligeholder to lister af processer for hhv. de processer, der ønsker at sende, og modtage data på kanalen. Når en proces eks. ønsker at modtage data, tilføjer den sig selv til listen af processer, der ønsker at modtage, og prøver derefter i `match` funktionen at finde en proces, der vil sende data. Er der ingen processer, der venter på at sende data, venter processen på, at en proces melder sig klar til at sende data, ved at kalde `match`. Til hver vellykket kommunikation af data vil `match` altid blive kaldt to gange, hvor kun den sidste vil resultere i at kommunikationen lykkes.

Ideen bag funktionen `match` er enkel og udnytter, at greenlets-versionen er enkelttrådet, så hver proces kan løbe listerne igennem, uden andre processer ændre på listernes tilstand. Vi er kommet frem til, at en simpel sortering af listerne ud fra processernes interne prioritet vil resultere i, at det altid er den højst prioriterede proces der indgår i kommunikationen. Den ændrede `match`, hvor der foretages en sortering af de to lister, kan ses i kodeuddrag 4.2 på næste side, hvor det kun er linje 119 og 120 der er ændret.

kodeuddrag 4.2: Funktionen `match` der sorterer kanalrequests.

```
    def match(self):
118     if self.readqueue and self.writequeue:
            self.readqueue.sort(key=lambda channelReq:channelReq.process.internal_priority)
120         self.writequeue.sort(key=lambda channelReq:channelReq.process.internal_priority)
            for w in self.writequeue:
122             for r in self.readqueue:
                    if w.offer(r):
124                     return
```

Funktionen `match` vil blive kaldt en gang for hver proces der ønsker at kommunikere, og derfor vil det kun være det sidste element i listen som ikke er sorteret korrekt ved hver kald af `match`. Desuden vil der altid i den ene liste maksimalt være på et element. Bemærk desuden at listerne er sorteret så værdien af den interne prioritet er stigende, og derfor er det processen med lavest værdi, der først bliver udvalgt til et match, i overensstemmelse med repræsentationen af prioriteter som nævnt i afsnittet "Ændringer i skemaplanlæggeren" på side 54.

4.4.6 Prioritetsnedarvning

Prioritet i et RTP system skal ses i forhold til alle processers prioritet. En proces kan derfor ikke i sig selv have en absolut høj prioritet, men kun have høj prioritet ifht. de andre processers prioritet. Ved at give en høj prioritet til en proces, vil dette dermed indirekte sænke de andre processers prioritet, et fænomen vi vil kalde "prioritetsdevaluering".

For at minimere prioritetsdevaluering i forbindelse med prioritetsnedarvning, ønsker vi at minimere den tid en proces har en kunstigt høj prioritet, og at minimere antallet af processer, hvis prioritet øges.

Som vi er kommet frem til i afsnit 4.2.4 på side 45, skal der foregå prioritetsnedarvning i forbindelse med kommunikation, hvis der ikke findes nogle processer, der umiddelbart er klar til at kommunikere. I PyCSP kan man umiddelbart evaluere, om der er processer klar til at kommunikere over en given kanal. Det skyldes, at processer der ønsker kommunikation befinder sig i listerne `readqueue` og `writequeue`. Hvis ingen processer ønsker at kommunikere, kan man dog ikke finde de processer som potentielt kan indgå i kommunikation. Vi må derfor udvide kanalerne i RTP versionen med to lister, `readerprocesses` og `writerprocesses`, der består af de processer, der potentielt kan sende og modtage data over kanalen. Vi håndterer vedligeholdelsen af disse lister, ved at hver proces ved opstart tilføjer sig selv til de kanaler, den har mulighed for at kommunikere over. Et oplagt sted at implementere denne funktionalitet er i processens

`__init__` funktion, da alle kanalender som denne proces potentielt kan kommunikere over, findes som argument til `__init__` funktionen. Kodeuddrag 4.3 på denne side viser udvidelsen af funktionen, hvor argumenterne gennemløbes, mens der ledes efter kanaler, som processen skal registreres i.

Kodeuddrag 4.3: Uddrag af `Process`' `__init__` funktion

```
    for arg in args:
30      if isinstance(arg, pycsp.greenlets.channelend.ChannelEndRead):
            arg.channel._addReaderProcess(self)
32      if isinstance(arg, pycsp.greenlets.channelend.ChannelEndWrite):
            arg.channel._addWriterProcess(self)
```

Kanaler kender nu både de processer, der på et specifikt tidspunkt ønsker at kommunikere vha. listerne `readqueue` og `writequeue`, og de processer, der potentielt vil kunne kommunikere vha. listerne `readerprocesses` og `writerprocesses`. Processer der ønsker at kommunikere kan, som normalt umiddelbart evaluere om det er muligt; såfremt det ikke er muligt, kan den nu evaluere hvilke processers prioritet den kan øge, for at bringe dem i en tilstand hvor de kan indgå i den ønskede kommunikation.

Funktionaliteten til prioritetsnedarvning skal implementeres i de to interne kommunikationfunktioner `_read` og `_write`. Fordelen ved at placere prioritetsnedarvning i disse to funktioner er, at de bruges af processerne både i forbindelse med normal blokerende kommunikation og i forbindelse med kommunikation i `alternation`. Vi har udvidet funktionerne med følgende liste af begivenheder:

- Undersøg om processen opfylder kriterierne for at starte en prioritetsnedarvning.
- Forhøj prioriteterne for de potentielle processer i enten `readerprocesses` eller `writerprocesses`.
- Umiddelbart efter kommunikationen nedprioriteres de processer, man midlertidigt har øget prioriteterne på.

Som beskrevet er det vigtigt, at vi igennem hele designet forsøger at begrænse mængden af prioritetsnedarvningen, og derfor har vi tilføjet en række egenskaber, der skal være indfriet, før prioritetsnedarvning forsøges. Disse er: processen skal have en prioritet, enten direkte eller efter en nedarvning; kanalen må ikke være klar til kommunikation, hvilket vil sige, at hvis processen ønsker at skrive, må der ikke findes en proces, der er klar til at modtage data; endeligt skal processen ikke have overskredet sin egen deadline, da denne til slut blot vil kaste en exception, og hele prioritetsnedarvningen vil være irrelevant.

Selve prioritetsforhøjelsen og den senere nedprioritering er simpel, da processen blot sender sin prioritet til alle processerne i den relevante liste dvs. `writerprocesses` for `_read` funktionen og vice versa. Hvis en proces modtager en lavere prioritet end dens egen prioritet, ses der bort fra hhv. op- og nedjusteringen, så en prioritetsnedarvning ikke resulterer i en forringelse af prioritet.

4.4.7 `Alternation`

Som nævnt i afsnit afsnit 4.2.5 på side 47 har vi behov for at kunne tilknytte en prioritet til en kanal for at kunne håndtere udvælgelse i `alternations`. Vi har allerede prioriteter for processer og ønsker, at kanalernes prioritet skal defineres på baggrund af hvilke processer, der er tilknyttet kanalen. Vi skal kunne håndtere både input- og output-guards og ønsker seperate prioriteter for disse. Vi tilknytter derfor to prioriteter til hver kanal. De to prioriteter er sat som de højst prioriterede processer, der er klar til at hhv. modtage og sende data. En kanals prioritet er derfor ikke fast som for processerne, hvor de får sat en prioritet (der dog kan ændres med prioritetsnedarvning), men nærmere en emuleret prioritet, som ændre sig baseret på alle processernes tilstand.

Til at implementere de to prioriteter introduceres to hjælpefunktioner, der løber hhv. `readqueue` og `writequeue` igennem og finder den højst prioriterede proces, der er villig til hhv. at sende og modtage data. Når `alternation` ønsker at finde prioriteten for en kanal, kigger den på om kanalen i `alternation` er tilknyttet en output- eller inputguard og finder den korrekte prioritet.

4.5 Evaluering

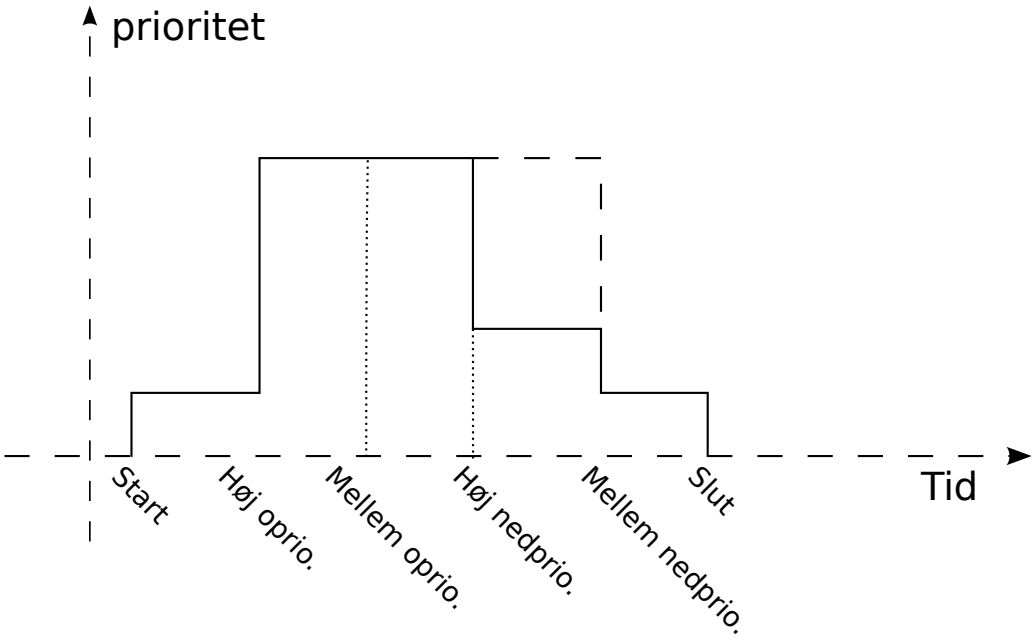
Vi har i dette afsnit beskrevet hvad der skal til, for at implementere RTP i PyCSP. Vi vil i dette afsnit gennemgå hvordan RTP kan implementeres, for derefter at evaluere vores løsning med udgangspunkt i slagterieksemplet.

4.5.1 Test af Korrekthed

Vi har som i DES i afsnit 3.3 på side 29 løbende skrevet tests, før vi implementerede hver ny funktion i RTP-versionen. Bilag A.2 viser testresultaterne for de tests der er lavet specifikt for RTP-versionen.

Alle tests undtagen én fungerer korrekt. Testen, der fejler hedder test_xreset_inheritance_from_two_step og viser en situation, hvor den samme proces får løftet sin prioritet to gange i træk, først med en høj prioritet og efterfølgende med en mellemprioritet. Efterfølgende skal processen sænke sin prioritet, først til mellemprioriteten og til slut til sin originale prioritet. Her viser det sig, at

vi har lavet en fejl i implementeringen, således at prioriteten ikke bliver nedsat til mellemprioriteten. Figur 4.13 på denne side viser prioriteten, mens processen bliver op- og nedprioriteret. Vi har ikke prioriteret at løse dette problem, men det kan løses ved at kræve, at når en proces opprioriteres, gemmes oplysningen om, hvilken proces der står bag, så når en proces ønsker at fjerne sin opprioritering fra andre processer, er det kun sin egen prioritet, den fjerner.



Figur 4.13: Figuren viser hhv. forventet og faktisk prioritetsarvning. Der hvor den faktiske og forventede opførsel adskiller sig er forventet den fuldt optrukne streg, mens den stiplede streg er den faktiske opførsel.

4.5.2 Slagterieksempel

Vi vil nu sammenligne implementeringerne i de tre eksisterende versioner af PyCSP mod henholdsvis hinanden og RTP-versionen. Ud fra denne sammenligning vil vi se på fordele og ulemper ved de forskellige versioner.

Ved at benytte PyCSP til slagterieksemplet opnår man et modulært design, der nemt kan udvides hvis de fysiske rammer for slagteriet ændrer sig. Viser det sig f.eks at kameraet holder den samlede produktivitet af netværket tilbage, kan slagteriet nemt tilføje endnu et kamera, og udvide procesnetværket med endnu en kameraproces, som kan arbejde samtidigt med det første kamera.

En kørsel af de tre eksisterende versioner giver et meget forskelligt resultat, som vist i tabel 4.2 på modstående side. I denne tabel har vi foretaget en simu-

lering af 100 grise. Denne simulering er kørt 10 gange, og tabellen viser gennemsnittet og standardafvigelsen. Af tabellen kan man se at versionen som forventet har betydning for antallet af grise der rettidigt kan nå at blive bearbejdet.

Proces-versionen er ikke som forventet bedre en de andre versioner til at bearbejde griseobjekterne rettidigt. Dette skyldes at testen er gennemført på en maskine, med kun en kerne. Dermed kan processes-versionen ikke drage nytte af flere kerner, men er begrænset til kun at kunne køre en proces af gangen. I processes-versionen, foregår kontekstskiftene i operativsystemets skemaplanlægger, der er væsentligt langsommere end i `greenlets`- og RTP-versionen, hvor konktekstskiftene sker i vores skemaplanlægger.

I vores arbejde med anvendelsesområdet interaktiv planlægning i kapitel 5, har vi implementeret et prioritetssystem for processer, da vi netop indenfor dette anvendelsesområde, har tænkt på det som en mulig udvidelse. Vi har derfor medtaget resultater hvor vi har højere prioriteter på de essentielle processer. Umiddelbart ser det ud til at forværre ydelsen, hvilket må tillægges at håndteringen af prioriteter skaber et overhead, sammenholdt med, at alle processer i denne version af eksemplet er lige vigtige.

Version	**Samtidige grise**	**Succesrate (%)**	**Standard Afvigelse (SA)**
Processes	1	16	2
Processes	2	0	0
Processes	3	0	0
Processes	5	0	0
Threads	1	34	6
Threads	2	1	1
Threads	3	1	0
Threads	5	1	1
Greenlets	1	44	7
Greenlets	2	49	1
Greenlets	3	32	0
Greenlets	5	20	0
RTP u. prioritet	1	42	6
RTP u. prioritet	2	44	2
RTP u. prioritet	3	31	2
RTP u. prioritet	5	19	0
RTP m. prioritet	1	44	10
RTP m. prioritet	2	21	5
RTP m. prioritet	3	21	7
RTP m. prioritet	5	7	3

Tabel 4.2: 10 kørsler hvor 100 grise bliver sendt igennem procesnetværket.

For at få mere samtidigt arbejde har vi ændret indstillingerne for simuleringen, så transportbåndet er længere, og dermed giver mere tid til at nå deadlinen, men samtidig har vi flere grise samtidigt i simuleringen. Resultaterne af dette fremgår af tabel 4.2 på foregående side. Når antallet af grise, der skal bearbejdes samtidigt stiger, øges antallet af processer, der må kæmper mod hinanden for CPU-tid. Dette er ens for de tre eksisterende versionerne af PyCSP.

I Proces-versionen medfører det muligheden for, at griseobjekterne kan blive bearbejdet parallelt, hvis computeren har denne mulighed. Ofte vil der være flere griseobjekterne end processorer, hvorfor griseobjekterne stadig vil skulle kæmpe mod hinanden, for at komme igennem netværket. Når griseobjekterne skal kæmpe for at komme igennem netværket, sker det, i denne version, uden hensyn til hvilken gris der er nærmest robotten. Man risikerer dermed en situation hvor griseobjekterne kan overhale hinanden, så det ikke er grisene nærmest robotten der først bliver bearbejdet.

`RTP`-udvidelsen bygger på greenlets-versionen, og vil derfor have de samme begrænsninger som denne, som diskuteret i implementeringen i "Implementering" i afsnit 4.3 på side 51. Grisseobjekter kan ikke på samme måde overhale hinanden i RTP-versionen, da hver proces har tilknyttet griseobjektets deadline. Dermed sikres, at det altid er griseobjektet tættest på sin deadline, der sendes igennem netværket først.

I RTP-versionen, slipper de enkelte processer desuden for at holde styr på tiden, og vurdere om det enkelte griseobjekts deadline, er overskredet. Når de starter, har de en deadline svarende til det tidspunkt robotten senest skal have griseobjektet. Denne sættes via funktionen `Set_deadline`. `Set_deadline` kan resultere i en exception, og derfor skal hver proces kunne håndtere en `DeadlineException`, som de i dette eksempel blot kan håndtere, ved at smide griseobjektet væk. Det kan de gøre, da robotten på dette tidspunkt tager sin beslutning, og derfor vælger at udskære grisen uden specialviden. Griseobjektet er dermed ikke længere relevant og kan smides væk, så processen kan gå i gang med modtage et nyt griseobjekt.

I greenlets-versionen kom vi ind på at processerne frivilligt skal afgive kontrollen, før robotten kan foretage udskæringen, men at der ikke findes en metode til midlertidigt at afgive kontrollen. Med RTP-versionen og funktionen `Release`, har alle processer mulighed for at afgive kontrollen, så robotten rettidigt kan foretage selve udskæringen. Hermed skal vi ikke introducere en delt datastruktur, men kan lade det være op til robotprocessen at igangsætte robotten.

Efter at have implementeret RTP-versionen, og implementeret vores eksempel i denne version, kan vi i tabel 4.2 på foregående side se at vores løsning ikke

er mærkbart bedre en greenlets-versionen. Dette mener vi skyldes, at det valgte eksempel er for simpelt og derfor ikke lader RTP-versionen komme til sin ret. I eksemplet har alle processer den samme deadline, i forhold til hvornår de ankommer, hvorfor de alle er lige begrænset. At udvælge et griseobjekt frem for et andet, har ikke indflydelse på hvor mange griseobjekter vi i alt kan nå, men ændre blot hvilke af griseobjekterne, der bliver bearbejdet rettidigt. Det ekstra arbejde vi foretager i skemaplanlæggeren gavner ikke slagterieksemplet, men begrænser tværtimod RTP-versionen i tiden processerne har til at bearbejde hvert griseobjekt.

Med udgangspunkt i at eksemplet ikke er optimalt til at vise vores løsning, ønsker vi at ændre eksemplet, for at se hvordan RTP-versionen bruges, hvis ikke alle processerne har en deadline. Vi har udvidet slagterieksemplet med en ekstra proces. Denne dummyproces har ingen deadline, og kører i en fast løkke, der estimerer π. Formålet med processen er at simulere et arbejde der også ønskes udført, men som ikke er tidskritisk. Når greenlets-versionen udvælger hvilke processer der skal køres, gøres det tilfældigt, hvorfor processen uden deadline kan risikere at blokere for de tidskritiske processer. I RTP-versionen bør tiden vi bruger i dummyprocessen til gengæld begrænses, når der er tidskritiske processer, der skal bearbejdes.

Ved at introducere en dummyproces, oplevede vi indledningsvis at greenlets-versionen, ikke blev færdig, men befandt sig i en livelock. Grunden til dette skyldtes en blanding af vores implementering af dummyprocessen, og implementeringen af greenlets-versionen. Som det ses af kodeuddrag 4.4 på denne side, bruger vi en `alternation` for midlertidigt at afgive kontrollen så den senere kunne genplanlægges, samt at detektere hvornår vi skal slutte, da kanalen så vil være blevet forgiftet. Denne løsning placerer processen på `timers` listen, men da skemaplanlæggeren foretrækker, at planlægge processer fra denne liste frem for `next`-listen, og dummyprocessen altid er klar til at blive udvalgt, vil det kun være dummyprocessen der bliver bearbejdet. Det samme problem gør sig ikke gældende i RTP-versionen, da processer i `timers`-listen bliver planlagt på lige fod med alle andre processer.

kodeuddrag 4.4: Uddrag af Dummyproces

```
while True:
    Alternation([{Timeout(seconds=0.005):None}, {dummy_in:None}]).select()
    time_spent -= time.time()
    dummywork(work)
    time_spent += time.time()
```

For ikke unødigt at give vores version en fordel har vi valgt at ændre dummyprocessen, så også greenlets-versionen kan blive færdig. Vi har ikke mulighed for i greenlets-versionen manuelt at stoppe eksekveringen af processen, og lade den blive placeret på `next`-listen. Den eneste måde for en proces at komme på denne liste er efter at have kommunikeret. Vi ændrer derfor dummyprocessen til et dummynetværk bestående af to processer, der hver foretager et stykke arbejde og kommunikerer, som vist i kodeuddrag 4.5 på denne side. I dette netværk findes der to dummyprocesser, der skiftes til at sende hinanden den akkumulerede tid de har været aktive. Fordi de efter hver periode, sender den akkumulerede tid til den anden dummyproces, vil den ene hele tiden skulle vente, for så at blive lagt på `next`-listen. Dermed indgår dummyprocesserne på lige fod med alle andre aktive processer i netværket.

kodeuddrag 4.5: Uddrag Dummynetværk

```
while True:
    time_spent = dummy_in()
    time_spent -= time.time()
    dummywork(work)
    time_spent += time.time()
    dummy_out(time_spent)
```

Version	Tid i dummyproces(s)	SA.	Succesrate (%)	SA.
Greenlets	1.29	0.61	13	2
RTP	1.05	0.28	24	6
RTP m. prioritet	0.74	0.34	42	13

Tabel 4.3: 10 * 100 grise køres igennem procesnetværket. Desuden er der tilknyttet en dummyproces, uden en deadline, der hele tiden ønsker at arbejde. For hver kørsel udregnes hvor meget kørselstid dummyprocessen har kørt, samt antallet af grise der bliver rettidigt bearbejdet.

Tabel 4.3 på denne side viser den gennemsnitlige tid hver version har brugt i dummyprocessen, samt hvor mange grise, der er blevet bearbejdet rettidigt. Som det ses, bruges der mindre tid i dummyfunktionen i RTP-versionen end i greenlets-versionen. Dette medfører at antallet af grise, der rettidigt bliver bearbejdet, er højere i RTP-versionen. Denne test viser dermed, at RTP-versionen formår at udvælge de tidskritiske processer, på bekostning af processer uden deadline, og at denne udvælgelse medfører en øget succesrate, på trods af den øgede kompleksitet i RTP-versionen. Vi ser yderligere at den ekstra funktionalitet til at håndtere prioritet spiller en stor rolle, når vi introducerer dummyprocesserne. Det sker

fordi vi nu har en et scenarium, hvor vi kan differentiere i i prioriteten af de enkelte processer, og sikre at de højt prioriterede processer afvikles når der er brug for det.

4.6 Fremtidigt arbejde

Vi har i dette kapitel opstillet en basal model for en RTP-implementering i PyCSP. Implementeringen er foretaget med henblik på blot at vise muligheden for en sådan model, og der er flere udviddelser som vi mener vil kunne forbedre modellen såfremt de kan implementeres. Vi vil i dette afsnit gennemgå nogle forbedringer som vi mener er interessante, men som ikke er inddraget i den basale implementering.

Evaluering af effektivitet

Vi har med eksempler og test vist, at den implementerede løsning fungerer teoretisk korrekt. Vi har dog ingen reelle målinger af, hvor meget tid vi bruger på at evaluere hvilken proces der skal aktiveres, samt opretholde de metadata der skal til for at foretage denne vurdering. En grundig analyse af tidsforbruget i de administrative dele af vores implementering ville derfor være interessant at udføre, så man bl.a. kan udlede generelle retningslinier for, hvor ofte det vurderes hvilken proces der skal køre og hvor beregningstung hver proces bør være for at opnå den bedste ydelse.

Estimater for udførselstid

Den primære begrænsning i vores løsning er manglen på at kunne evaluere hvor lang tid en proces eller dele af en proces tager at udføre. Hvis vi kunne udvikle en løsning der kunne foretage estimater af processers udførselstid ville vi kunne vælge en anden udvælgelsesalgoritme, som f.eks LL frem for EDF. Derved ville udvælgelsen af hvilken proces der skal aktiveres blive mere præcis. Ligeledes vil muligheden for at vurdere hvor langt en proces er nået i sin udførsel også være meget brugbart i forbindelse med prioritetsnedarvning. I vores implementering nedarver vi prioritet til alle processer som har mulighed for at opfylde en afhængighed. Såfremt vi kan vurdere hvilken proces der er tættest på at kunne opfylde afhængigheden, kan vi nøjes med at nedarve prioriteten til denne proces. Dette vil afhjælpe problemet med prioritetsdevaluering som nævnt i "Ændring af prioritet" i afsnit 4.2.4 på side 45.

Håndtering af forskellige typer deadlines

I den nuværende løsning håndterer vi alle deadlines ens. Det er der fordele og ulemper ved, hvor en fordel er, at udvikleren får maksimal kontrol over hvad der skal ske såfremt en given deadline overskrides. Vi håndterer overskridelser af deadlines ved at kaste en exception når det sker. Dette er måske ikke altid ønskværdigt hvis det er en soft deadline der overskrides, da processen derved afbrydes. Det kunne tænkes at der er tilfælde hvor det er mere hensigtsmæssigt at udføre processen helt, og først her give besked om at den ikke nåede sin deadline.

Udviklerbestemte prioriteter

På nuværende tidspunkt har alle processerne den samme prioritet inden de planlægges, og deres prioritet afhænger udelukkende af deres deadline. Det kunne være interessant at undersøge om man kan bruge en anden skemaplanlægningsalgoritme, der kan håndtere at processerne har forskellige prioriteter inden de blev planlagt.

Hvis muligheden for at differentiere processerne blev implementeret, kunne det være spændene hvis man kunne udvide skemaplanlægger, så udvikleren kunne angive et kritisk sæt af processer, for hvilke det kunne garanteres at de ikke ville overskrider deres deadline.

4.7 Opsummering

Vi har i dette kapitel beskrevet RTP og sat det i kontekst med PyCSP. Vi har set på forskellige typer af deadlines, præsenteret flere skemaplanlægningsalgoritmer og ud fra krav og muligheder valgt EDF.

Denne algoritme er blevet implementeret i PyCSP. Vi har i den forbindelse implementeret prioritetsnedarvning, og intelligent udvælgelse af hvilke processer der først skal kommunikere, hvis der er flere processer med forskellig deadline tilknyttet den samme kanal. RTP-versionen kræver kun at en udvikler benytter yderligere tre funktionaliteter: `Now, Wait` og `DeadlineException` udover PyCSP. Vi mener derfor det er simpelt for en udvikler, der allerede kender PyCSP, at bruge vores udvidelse.

Vi har vist brugen af udvidelsen, med et eksempel, der er inspireret af en reel problemstilling, på et slagteri. Eksemplet er implementeret både ved hjælp af de eksisterende versioner af PyCSP samt vores udviklede metode, for derved bedst at kunne vise styrker og svagheder, ved vores implementering. Vi kan på baggrund af evalueringen konkludere at vores løsning til dette eksempel ikke levere nævneværdigt bedre resultater end de eksisterende versioner. Dette mener vi kan

føres tilbage til, at eksemplet ikke er velvalgt, og vi har modelleret det for simpelt med for få processer, der alle har en deadline, og med den samme laxity. Vi har derfor udvidet eksemplet med processer uden deadlines, og kan se at i disse tilfælde er RTP mellem to og tre gange så effektiv som greenlets-versionen, til at få ført griseobjekter rettidigt gennem procesnetværket.

Kapitel 5

Interaktiv planlægning

Interaktiv planlægning (IP) er det sidste anvendelsesområde, vi vil belyse med henblik på at indføre tid i PyCSP. Det har dog vist sig, at emnet kun er sparsomt berørt i litteraturen, og der er ikke en klar definition på, hvad det er, og hvor det anvendes.

Vi forventede, at dette anvendelsesområde var brugt i forbindelse med computerspil, men efter at have studeret litteraturen og have rådført os med Lektor Kenny Erleben, der underviser på Det Danske Akademi for Digital Interaktiv Underholdning (DADIU), er vi kommet frem til, at der ikke findes en udbredt definition af IP. Han fortæller desuden, at en af begrundelserne for, at computerspilfirmaerne ikke er interesserede i at oplyse om deres metoder til at planlægge begivenheder i spil, er at det betragtes som en forretningshemmelighed.

Vi vil derfor i stedet på baggrund af en række praktiske eksempler indkredse, hvad vi forventer, IP skal kunne bruges til, og på baggrund af eksemplerne se på muligheden for at implementere en model i PyCSP. Dette kapitel adskiller sig dermed væsentligt fra de foregående to kapitler, i og med anvendelsesområdet ikke baserer sig på en fast definition givet af litteraturen, og vi ikke bygger på kendt viden.

5.1 Eksempler

Vi har valgt to scenarier, som vi forventer med fordel kan benytte IP. Det første er en repræsentation af et ur, hvilket vi mener, er det simpleste eksempel, der kan gøre brug af IP. Det andet eksempel er en del af et computerspil, som er vores forventede primære anvendelsesområde for IP.

5.1.1 Et ur

Det første eksempel på IP er en repræsentation af et digitalur. Uret består af seks cifre, og en gang i sekundet skal sekunderne tælles op. Det er et krav, at en opdatering skal sætte uret til at vise det korrekte tidspunkt. Vi forestiller os, at uret er en del af et større system, hvor der er andre ressourcekrævende begivenheder, der er vigtigere at få udført end opdateringen af uret. Opdatering af uret foregår ved at planlægge en begivenhed til hvert sekund, der specificerer, hvad uret skal vise. Da det har en lav prioritet, er der ikke nogen garanti for, at denne begivenhed indtræffer i det sekund, den er planlagt til. Derfor skal begivenheden bortkastes, såfremt den overskrider sin deadline. Deadlinen er et nyt sekunds begyndelse, for at sikre den krævede korrekthed.

Uret er repræsentativt for IP, da der planlægges en række begivenheder, der skal foregå i fremtiden og hver især har tilknyttet en deadline.

5.1.2 Computerspil

Et andet eksempel, vi mener repræsenterer IP, er mere realistisk og bunder i vores oprindelige forventning om, at IP var defineret af computerspilindustrien, som en metode til at planlægge, hvordan spil kan forløbe. Uden deres definition af IP vil vi i stedet opstille et hypotetisk eksempel.

Vi forstiller os et computerspil, der er skrevet i PyCSP, og hvor hvert element i spillet er en selvstændig proces. I dette computerspil skal der være en fugl, der jævnligt flyver på tværs af skærmen. Fuglen skal starte på et given tidspunkt og med en fast hastighed bevæge sig over skærmen. Fuglens bane over skærmen udregnes af to typer processer, baseret på en model, der minder om videokomprimering. Den første procestype er en højprioritetsproces, der står for at udregne positionen af fuglen med et fast interval. Den anden procestype kan bestå af mange lavprioritetsprocesser. Hver lavprioritetsproces står for en egenskab ved fuglen, som f.eks. optimere animationen af fuglen, tilknytte fuglekvidren og andre ikke essentielle egenskaber. Den højtprioriterede proces udføres sjældent, men er essentiel at få udført, mens lavprioritetsprocesserne blot skal udføres, hvis der er mulighed for det og ellers skal droppes.

5.2 Beskrivelse

Vi kan se på hvilke egenskaber eksemplerne har, og hvilke krav de derved stiller for at kunne håndteres i et programmeringssprog. Først og fremmest ligger eksemplerne inden for realtidsmodellen. Ligeledes skal der være mulighed for at tilknytte en deadline til en begivenhed. Dette vil i vores opstillede eksempler være

hard deadlines, men vi kan ikke udelukke at der findes eksempler hvor andre typer deadlines vil være fordelagtige. Eksemplet med computerspillet viser at der er behov for at tilknytte en prioritet, der er uafhængig af deadline, til en begivenhed. Dette har vi diskuteret som en mulig udvidelse til RTP i afsnit 4.6 på side 67. Disse egenskaber minder alle om dem der er er givet for RTP. Ud over disse skal vi også kunne planlægge en begivenhed. Det lægger sig mere op af DES med det forbehold at vi i IP ikke garanterer at en begivenhed sker på et bestemt tidspunkt, men tidligst på det angivne tidspunkt.

Vi kan dermed se IP som en blanding af RTP og DES, med tidsmodellen og deadlines fra RTP, og planlægningen af begivenheder fra DES.

5.3 Design og implementering

Da kravene til IP rent praktisk ligger meget tæt op af de løsninger, vi tidligere har beskrevet, i RTP, vil vi ikke implementere en selvstændig IP-version. Vi vil i stedet udvide RTP-versionen med den krævede funktionalitet så RTP kan foretage både RTP og IP. Vi skal derfor udvide RTP så man kan planlægge begivenheder der skal foregå ud i fremtiden, samt sætte en prioritet på en proces uafhængigt af processens deadline.

5.3.1 Funktionerne `Now` og `Wait`

Vi argumenterer i kapitel 3 for, at planlægning af begivenheder til et givet tidspunkt kan tolkes som venten indtil tidspunktet. Vi vil derfor også i forbindelse med IP introducere de to globale funktioner `Now` og `Wait`, der hhv. returnerer den aktuelle tid og lader en proces vente i et givent tidsrum. Vi har til IP ændret den interne implementering af funktionerne, så de bruger realtid. Ved at bruge de samme funktioner, sikres en ensartet implementering af tid på tværs af Timed-PyCSP, og man kan i vores øjne med fordel tilføje funktionen `Now` til alle PyCSP versionerne for på den måde at ensrette de forskellige implementeringer. Hvis `Now` blev inkluderet i greenlets-versionen, kunne den fjernes helt fra denne version, da de baserer sig på den samme tidsmodel, nemlig realtid.

Forskellen i implementering mellem versionen, der benytter realtid som tidsmodel, ifht. til versionen der bruger diskrettid som tidsmodel, er simpel. I stedet for at skemaplanlæggeren kender tiden, og at det derfor er dennes tid, vi returnerer i diskret tid, bruger vi nu Pythons `time`-modul. Genimplementeringen består derfor i at ændre funktionen `Now` til at bede `time`-modulet om det nuværende tidspunkt. Implementeringen af `Wait` benytter sig af `Now` til de tidsspecifike operationer, hvorfor vi kan genbruge denne funktion, uden en eneste ændring.

5.3.2 Udvikler-prioriteter

I computerspilseksemplet, har processerne forskellige prioritet dikteret af udvikleren. Denne prioritet er ikke det samme som den prioritet, der allerede findes i RTP, da denne er beregnet af skemaplanlæggeren. For at adskille dem vil vi kalde udviklerens prioriteter for udvikler-prioritet. Vi skal udvide RTP således at det kan håndtere processer, der både kan have deadlines og udvikler-prioriteter tilknyttet. Dette har vi allerede beskæftiget os med i afsnit 4.6 på side 67, som en fremtidig mulighed for RTP.

Først skal det fastlægges i hvilket interval udvikler-prioriteter kan antage. Vi ønsker ikke at begrænse udvikleren ved at have for få prioriteter, men omvendt risikere man også at introducere en prioritetsskrue, hvis man har et stort antal prioriteter. Dette sker hvis en udvikler flere gange undervejs i udviklingen af et program øger den maksimale prioritet, da han mener den nuværende proces er den vigtigste uden at gennemtænke det i relation til alle andre processer. Dermed risikerer man at udvande prioriteten for de allerede udviklede processer. Der skal derfor være en maksimal prioritet, og intervallet man kan angive prioriteter må ikke animere til en prioritetsskrue.

Præcis hvor stort intervallet skal være for udvikler-prioriteter, kræver dog en bredere analyse af flere projekter, og vi vil derfor begrænse os til at lave et foreløbigt interval på ti, man senere nemt vil kunne ændre baseret på en bedre analyse.

I RTP-versionen er skemaplanlæggeren en EDF algoritme, som kun planlægger processerne på basis af deres deadline. Vi kan derfor ikke bruge den samme skemaplanlægningsalgoritme, men skal udvide skemaplanlæggeren. For at udvide skemaplanlæggeren skal vi definere den indbyrdes relation mellem deadlines og udvikler-prioriteter. Baseret på computerspil-eksemplet kan vi se, at de processer, der har tilknyttet en udvikler-prioritet, skal gå forud for både de processer, der har tilknyttet en deadline, og dem der hverken har prioritet eller deadline.

I RTP indeholder skemaplanlæggeren alle processer med en prioritet i en enkelt hob kaldet `has_priority`. Vi kan udvide denne hob til en sorteret liste af hobe, en for hver prioritet. Med en liste af hobe vil hver hob være lille, og dermed vil de enkelte operationer på hoben være hurtigere. Skemaplanlæggeren kan desuden bruge EDF på hver hob da alle processerne i hver hob har samme prioritet. For at udvælge hvilken proces der skal aktiveres vælges det første element i den første ikke-tomme hob. Alternativt kan vi udvide den ene hob, der allerede findes, så alle processer både med og uden udvikler-prioritet befinder sig i den samme hob. Med denne metode kan man ikke bruge EDF, men der skal udvikles en funktion, der kombinerer udvikler-prioriteten og deadlinen til en endelig prioritet. Man

vil så kunne bruge en Least Priority First (LPF) algoritme, der fuldstændigt svarer til EDF, men baserer udvælgelsen på prioritet. I sagens natur vil denne hob være større end, hvis man brugte en liste af hobe. Dette er dog ikke et stort problem, da hoben gemmer data i en træstruktur, og søgetiden vokser derfor logaritmisk til antallet af elementer. Vi antager derfor, at forskellen mellem de to fremgangsmåder ikke vil være nævneværdig, og under alle omstændigheder vil den ikke være anderledes end i RTP-versionen, hvor der også kun findes en hob. Fordelen ved kun at bruge en hob er, at vi kan genbruge `has_priority` hoben og ikke skal lave en større omskrivning af RTP-versionen. Desuden skal vi ikke lineært gennemløbe en liste af hobe for at finde den første hob med elementer, men kan med det samme starte i den korrekte hob. Endnu en fordel ved denne metode er, at vi nemt kan ændre vores oprindelige antagelse om, at prioritet skal gå forud for deadlines, blot ved at ændre på den eksterne funktion, der kombinerer de to parametre.

Vi mener, at fordelene ved kun at have en hob og muligheden for at kunne ændre på vægtningen af forholdet mellem udvikler-prioritet og deadline, er større end besparelsen i tid ved at have flere hobe, der skal vedligeholdes. Derfor vælger vi at genbruge `has_priority`, som den eneste hob til processer, der har deadlines og udvikler-prioritet. Med kun en hob skal vi derfor udvikle en ekstern funktion, der kombinerer to tal til en endelig prioritet. Denne funktion skal sikre at at prioriteten sat af udvikleren altid vil være dominerende. Det bør nævnes at jo højere udvikler-prioriteten er, jo vigtigere er processen. I implementeringen af `has_priority` bruges der en min-hob, hvorfor vi internt inverterer udvikler-prioriteten, så 0 er den højeste prioritet, og processer, hvor udvikleren ikke har angivet en prioritet, sættes til 10.

Vi har valgt at implementere en simpel løsning, hvor de to tal lægges efter hinanden. Hvis man eksempelvis har en udvikler-prioritet på 2 og en deadline på 20, sættes de efter hinanden, så den endelige prioritet bliver 220. Dette kan vi gøre, da antallet af cifre i hhv. deadlines og udvikler-prioritet ligger fast, så vi risikerer ikke en forskydning, når de lægges efter hinanden. For at sikre at processer, der har en udvikler-prioritet, og ingen deadline, også kan planlægges, bruges der en kunstig høj deadline. Ønsker man en anden løsning for at sammenholde prioriteter og deadlines, kan det nemt opnås blot ved at ændre i en enkelt funktion.

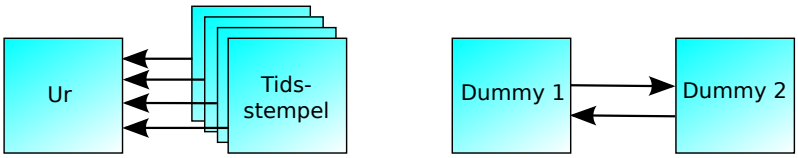
5.4 Evaluering

Vi har i det foregående afsnit beskrevet hvad der skal til, for at implementere IP i PyCSP. Vi vil i dette afsnit gennemgå hvordan IP kan implementeres, for derefter

at evaluere vores løsning med udgangspunkt i ureksemplet.

Vi har som i de foregående versioner løbende skrevet tests for at evaluere korrektheden, og resultaterne findes i afsnit 3.3 på side 29. Da denne version ikke kræver nogle store ændringer i implementeringen, er der ikke behov for at en stor mængde test, men dem der er skrevet fungerer alle korrekt.

For at kunne evaluere vores udvidelse har vi valgt at implementere ur-eksemplet, da tiden ikke har tilladt os at implementere fugle-eksemplet. Til ur-eksemplet er netværket er struktureret som vist på figur 5.1 på denne side. Som vi kom ind på tidligere, har vi en ur-proces, en række tidsstempel-processer, og nogle dummy-processer. Ur-processen modtager et tidstempel fra alle tidsstempel-processerne, og står for at at opdatere uret ud fra disse. Tidsstempel-processerne modtager fra start, hvert sit tidsstempel, som de planlægger at sende i fremtiden når tidspunktet er korrekt. Hver tidsstempelproces venter indtil dette tidspunkt, for så at sende tidspunktet til ur-processen. Samtidigt med at uret skal holdes opdateret, køres der også en række dummyprocesser i par, som kommunikerer med hinanden. Hver dummyproces udregner 50.000 iterationer i en simpel estimering af π, og på testmaskinen tager dette i gennemsnit 0.07707 sekunder med en standardafvigelse på 0.01245 sekunder. Dummyprocesserne brugte vi også i slagterieksemplet.



Figur 5.1: Procesnetværk med flere konverteringsprocesser, og analyseprocesser. Dummyprocessernes kørsel foregår uafhængigt af uret, og tidsstempelprocesserne, men alle processerne køres samtidigt.

I greenlets-versionen har vi ikke mulighed for direkte at planlægge en begivenhed i fremtiden. Vi kan dog oprette en separat tråd via `IO`-dekoratøren. I den separate tråd kan man så vente via funktionen `sleep`. Dette er den eneste metoden for processer at afgive kontrollen i et tidsrum i greenlets-versionen uden at blokere for hele programmet, og metoden er beskrevet på hjemmesiden for PyCSP. Metoden medfører dog et overhead, da der skal oprettes en ny tråd blot for at vente. Med introduktionen af `Wait`, bliver processen lagt på `timers`-hoben, og det er derfor ikke længere krævet at oprette en separat tråd til en proces der ønsker at vente. En udvikler der skal planlægge en proces til senere kørsel, slippe derfor i RTP for at skulle introducere koden vist i kodeuddrag 5.1 på modstående side før en proces kan planlægges.

kodeuddrag 5.1: Funktion der venter et antal sekunder

```
  @io
2 def sleep(n):
      import time
4     if n>0: time.sleep(n)
```

Ved implementeringen af greenlets-versionen, opstår der et problem i opdateringen af uret. Begrænsningen om kun at opdatere uret så længe tidsstemplet er korrekt, viser sig nemlig problematisk at implementere. I den første version fulgte vi eksemplet og implementeret kontrollen i tidsstempel-processen, med en `alternation`, og en `timeout`. Tidsstempel-processen, skal derfor inden et sekund have sendt beskeden til uret, og vil ellers helt droppe forsendelsen. Resultatet af denne implementering kan man se af linje 1 i tabel 5.1 på denne side. Den gennemsnitlige forsinkelse på 1.529 sekunder burde være umulig, da opdateringer på over et sekund bør falde bort. Grunden til, at forsinkelsen ikke falder bort, er at tidsstempel-processen korrekt får overført tidsstemplet til urprocessen inden for et sekund. Ur-processen bliver efterfølgende lagt på `next` listen da den er klar til at blive kørt, og har modtaget tidsstemplet. Nu skal den så slås, for at blive udvalgt af skemaplanlæggeren, på lige fod med de dummy-processer, der også er klar til at blive kørt. Når ur-processen bliver udvalgt er tiden forpasset og uret bliver opdateret for sent. Vi har derfor lavet endnu en version, hvor ur-processen også kontrollere, at tiden ikke er forpasset. Resultatet af denne ændring ses af linje 2 i tabellen, hvor vi kan se, at uret ikke når at blive opdateret en eneste gang, inden tiden er forpasset.

Version	Rettidige tidsopdateringer(%)	Gennemsnitlig forsinkelse(s)	Standard Afvigelse
Greenlets ver. 1	0	1.529	0.276
Greenlets ver. 2	0	NaN	0
RTP (100 opdateringer)	80	0.539	0.411
RTP (50 opdateringer)	100	0.077	0.023

Tabel 5.1: Proces-netværk betstående af et ur, 100 opdateringer, og baggrundsprocesser

De to forskellige implementeringer af greenlets-versionen viser tydeligt hvorfor den ikke er egnet til planlægge processer, der skal aktiveres inden for en tidsperiode. Problemet er, at selvom tidsstempel-processerne sender sit data til ur-processen inden for tidsgrænsen, skal ur-processen stadig vente på at blive aktiveret, på lige fod med alle dummy-processerne. I greenlets-versionen aktiverer

skemaplanlæggeren processerne ud fra en FIFO strategi, og ur-processen vil derfor komme sidst i køen af processer, der ønsker at blive aktiveret.

I modsætning til greenlets-versionen, og dens FIFO strategi ser vi i tabellen for RTP-versionen at den når 80% af tidsstemplerne. Værd at bemærke er at den gennemsnitlige forsinkelse i RTP-versionen kun indeholder forsinkelsen for de tidsstempler der ankom rettidigt, hvorfor tallet er svært at sammenligne med greenlets-versionen. Det mest bemærkelsesværdige ved RTP-versionen er at forsinkelsen og standardafvigelsen er meget høj. Dette passer ikke sammen med vores forventning om at ur-processen kommer til som den første proces, og derfor kun bør være forsinket med tiden det tager for kørslen af en dummy-proces.

Efter at have analyseret kørslen er vi kommet frem til at problemet skyldes prioritetsnedarvning, nærmere bestemt at vi ikke korrekt får nedprioriteret ur-processen igen, efter den har modtaget en opprioritering fra en tidsstempel-proces. Den gamle prioritet bliver efterfølgende fejlagtigt overført til tidsstempel-processer der aktiveres på et senere tidspunkt. Som antallet af tidsstempel processer aktiveres, vil tidsstempel-processer modtage flere og flere nedarvede-prioriteter, hvorved mere og mere tid bruges på prioritetsnedarvning. Vi kan konstatere at det er det samme problem som vi kom ind på i afsnit 4.5 på side 61, og vi forventer at en korrektion af prioritetsnedarvning vil løse problemet.

Da problemet vokser med antallet af tidsstempel-processer der startes har vi en forventning om at hvis der startes færre tidsstempel-processer vil fejlen ikke gøre en målbar forskel. Vi foretog derfor en test med kun 50 tidstempel-processer for at minimere problemet, og resultatet ses af linje 4 i tabel 5.1 på foregående side. Her kan vi se at den gennemsnitlige forsinkelse tidstemplerne ankommer med, svarer til kørslen af en dummy-proces, hvilket er forventet, da vi ikke har preemptiv kontekstskift, og derfor nødvendigvis må afslutte en dummy-proces før tidsstempel-processen bliver aktiveret.

På baggrund af testen forventer vi at når fejlen er løst, vil RTP-versionen kunne skalere frit og stadig have en gennemsnitlig forsinkelse og SA der er sammenlignelig med eksemplet hvor der kun er 50 opdateringer.

Til at evaluere effekten af prioriteter har vi ser på slagterieksemplet. Af tabel 4.3 på side 66 kan vi se at når der introduceres prioriteter opnår vi en 100% forøgelse af grise der bliver bearbejdet inden for sin deadline, set i forhold til standard RTP-versionen. Grunden til denne forøgelse i antal grise der bliver bearbejdet rettidigt kan føres tilbage til det samme problem, som vi konstaterede for greenlets-versionen af ur-eksemplet, nemlig at vi godt kan sende inden for en deadline, men at dette ikke betyder at modtageren også er blevet aktiveret inden for samme deadline. Når vi sender griseobjektet fra analyse-processen der har en

deadline til robot-processen, der ikke har en deadline, opfyldes analyses-processens kommunikation korrekt og inden for sin deadline, men da robot-processen ikke har en deadline ligger den på `no_priority` listen, sammen med alle dummy-processerne, og skal der kæmpe om at blive aktiveret. Med en prioritet på robot-processen bliver den aktiveret før dummy-processerne, og vi kan have et vilkårligt antal dummy-processer kørende samtidigt uden at det nævneværdigt har en indflydelse på succesraten af griseobjekter, der ankommer korrekt til robot-processen.

5.5 Opsummering

Vi har i dette kapitel gennemgået to eksempler for at indkredse en definition af interaktiv tid og er kommet frem til at det ikke er et selvstændigt anvendelsesområde, men ligger tæt op af RTP, hvorfor vi har udvidet RTP til at kunne foretage interaktiv planlægning.

Ud fra eksemplerne har vi identificeret at der mangler at blive implementeret planlægning af begivenheder, og prioritet af processer i RTP-versionen. Vi har implementeret den manglende funktionalitet, og for udviklerne introduceret funktionerne `Wait`, `Now` og `Set_priority`. Ved at introducere funktionaliteten direkte i RTP, opnår vi desuden at også RTP-versionen kan drage nytte af denne funktionalitet, som det ses af tabel 4.3 på side 66, hvor dummy-processer blev introduceret til slagterieksemplet.

Implementeringen af eksemplet har vist, at man nemt kan planlægge processer til at have begivenheder, der skal foregå i fremtiden, samt foretage en skemaplanlægning, der sørger for rettidigt, at få eksekveret de processer, der har en deadline. Implementeringen af eksemplet har også afdækket en fejl i vores implementering af prioritetsnedarvning, der medfører, at når antallet af processer stiger, falder effektiviteten. Vi har i den forbindelse fundet frem til, at det er den samme fejl, som vi har identificeret i RTP og forventer at løsning af denne fejl, vil medføre at der kan planlægges et vilkårligt antal processer.

Kapitel 6

Konklusion

Vores mål med dette speciale var at undersøge muligheden for en udvidelse af PyCSP, der muliggør brugen af tid direkte i sproget. Der findes allerede et massivt teoretisk arbejde inden for området, men ingen praktisk anvendelige implementeringer, så vores fokus har været på, at det skulle være praktisk anvendeligt. Dette afspejles i, at vi har valgt at benytte eksempler som omdrejningspunkt for vores analyser af de tre anvendelsesområder.

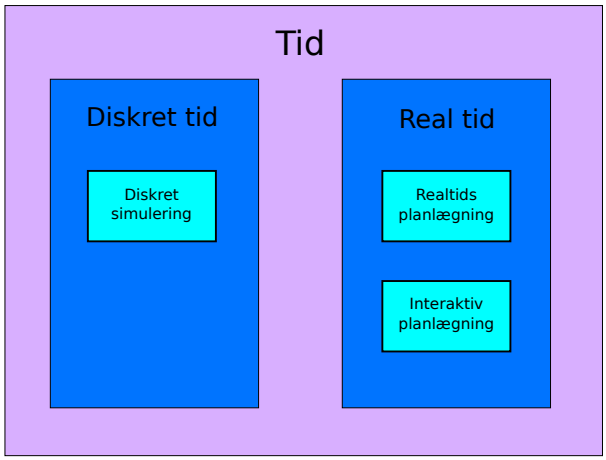
De tre anvendelsesområder vi identificerede i introduktionen var diskret simulering, realtidsplanlægning og interaktiv planlægning. Vi har vha. analyser af krav og eksempler udviklet løsninger der imødekommer alle tre amvendelsesområder.

Inden for diskret simulering har vi udviklet en løsning, der er let at anvende, og som eliminerer kravet om en delt datastruktur for at administrere tid i PyCSP. Dette er gjort ved at introducere en repræsentation af tid i skemaplanlæggeren, samt styring af aktivering af processer ud fra denne tid. Løsningen er let at anvende og kræver væsentligt mindre kode til at administrere tiden, end en tilsvarende løsning lavet i ren PyCSP. Sammenligner vi vores løsning med SimPy, der er et framework til simuleringer, skrevet i Python, mener vi at vores løsning er mere intuitiv og fleksibel at benytte til at modellere et givet problem. Dette skyldes i høj grad, at vi direkte kan benytte PyCSP's processer og kanaler. Et eksempel på hvordan diskret simulering i PyCSP giver udvikleren større frihed til sin modellering finder vi i en kommende udvidelser til SimPy. Der arbejdes på at udvide SimPy til at kunne håndtere reservation af flere begrænsede ressourser, som skal benyttes samtidigt. Dette er allerede muligt i vores løsning, uden det har været vores fokus, da ressourcer blot modelleres som processer i PyCSP. Ønsker man at reservere flere ressourcer, kommunikerer man blot med flere processer, og når alle processerne har svaret, holder man alle de begrænsede resourcer. Arbejdet kan nu udføres, og ressourcerne frigives igen.

I vores løsning til realtidsplanlægning har vi udvalgt earliest deadline first

(EDF), som den grundlæggende skemaplanlægningsalgoritme til at bestemme rækkefølgen for udførsel af processer. Vi har implementeret prioritetsnedarvning for at imødekomme problemer med prioritetsinvertering samt indført en prioriteret udvælgelse i `alternations`, og når der kommunikeres over kanaler. Vores eksempel viser tydeligt, at vores løsning vha. af planlægning, kan øge effektiviteten, såfremt der i programmet kan foretages en differentiering i prioriteten af de processer, der skal afvikles.

Vi er kommet frem til, at interaktiv planlægning ikke kan anses som et selvstændigt anvendelsesområde, men nærmere som en gren af realtidsplanlægning. De benytter begge realtid som tidsmodel, og har begge deadlines og prioriteter. Interaktiv planlægning gør det lettere at udtrykke et starttidspunkt vha. venten, men er ellers ikke fundamentalt forskellig fra realtidsplanlægning. Vi mener derfor at det er mere hensigtsmæssigt at anskue anvendelsesområderne ud fra hvilken tidsmodel, de bygger på. Figur 6.1 på denne side viser således opdelingen af anvendelsesområderne i henholdsvis diskret og realtid.



Figur 6.1: Forholdet mellem de tre anvendelsesområder vi opstillede i introduktionen.

Vi har gennem vores eksempler, vist at vores løsninger er brugbare i den nuværende form, men der kan foretages en række udvidelser, der vil gøre dem endnu mere anvendelige. Inden for diskret simulering synes vi det kunne være spændende at udvide løsningen til at kunne foretage en dynamisk evaluering af køer, for derved at give ekstra fleksibilitet i de udviklede simuleringer. I realtidsplanlægning har vi lavet en basisimplementering, der benytter EDF. Det kunne være interessant at kigge på mulighederne for at bedømme en proces' udførselstid, eller lade udvikleren angive dette. Herved åbnes op for brug af adskillige andre algoritmer end EDF til skemaplanlægningen og udvider anvendelsesområdet.

Kapitel 7

Litteraturliste

[1] N. Audsley og A. Burns. *Real-time system scheduling*. YCS 134 first year report, ESPRIT BRA Project (3092). University of York, 1990.

[2] Fred R. M. Barnes, Peter H. Welch og Adam T. Sampson. „Barrier Synchronisation for occam-pi“. I: *PDPTA*. 2005, s. 173–179.

[3] John Markus Bjørndalen, Brian Vinter og Otto J. Anshus. „PyCSP - Communicating Sequential Processes for Python.“ I: *CPA*. Udg. af Alistair A. McEwan m.fl. Bd. 65. Concurrent Systems Engineering Series. IOS Press, 2007, s. 229–248. ISBN: 978-1-58603-767-3. URL: http://dblp.uni-trier.de/db/conf/wotug/cpa2007.html#BjorndalenVA07.

[4] Alistair A. McEwan, Wilson Ifill og Peter H. Welch, udgivereudg. *C++CSP2: A Many-to-Many Threading Model for Multicore Architectures*. 2007, s. 183–205. ISBN: 978-1586037673.

[5] A. Burns og A. J. Wellings. „The notion of priority in real-time programming languages“. I: *Comput. Lang.* 15.3 (1990), s. 153–162. ISSN: 0096-0551. DOI: http://dx.doi.org/10.1016/0096-0551(90)90008-D.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications*. Nueva York, EUA : Springer, 2005.

[7] S. Cheng, J.A. Stankovic og K. Ramamritham. „Scheduling Algorithms for Hard Real-Time Systems–A Brief Survey“. I: (1987).

[8] Alexander Keewatin Dewdney. „Sharks and fish wage an ecological war on the toroidal planet Wa-Tor“. I: *Scientific American* Dec: 14-22 (1984).

[9] Python Documentation. „Heapq implementation“. I: *http://docs.python.org* (2010). URL: http://docs.python.org/dev/3.0/library/heapq.html (Lokaliseret d. 26. 05. 2010).

[10] Jack J. Dongarra m.fl. *A Proposal For A User-Level, Message Passing Interface In A Distributed Memory Environment*. Tek. rap. TM-12231. Oak Ridge National Laboratory, 1993.

[11] Rune Møllegård Friborg, John Marcus Bjørndalen og Brian Vinter. „Three Unique Implementations of Processes for PyCSP“. I: *Communicating Process Architectures*. (to appear). 2009.

[12] *Greenlet*. URL: http://codespeak.net/py/0.9.2/greenlet.html.

[13] C. A. R. Hoare. „Communicating sequential processes“. I: *Commun. ACM* 26.1 (1983), s. 100–106. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/357980.358021.

[14] http://simpy.sourceforge.net/. „SimPy“. I: *http://simpy.sourceforge.net/* (maj 2010). URL: http://simpy.sourceforge.net/ (Lokaliseret d. 25. 05. 2010).

[15] J. Lehoczky, L. Sha og Y. Ding. „The rate monotonic scheduling algorithm: Exact characterization and average case behavior“. I: (1989), s. 166–171.

[16] CL Liu og J.W. Layland. „Scheduling algorithms for multiprogramming in a hard-real-time environment“. I: *Journal of the ACM (JACM)* 20.1 (1973), s. 61.

[17] David May. „OCCAM“. I: *SIGPLAN Notices* 18.4 (1983), s. 69–79.

[18] Roger McHaney. *Understanding Computer Simulation*. Ventus Publishing ApS, 2009. ISBN: 9788776815059.

[19] Rune Møllegård Friborg (runedrengen). „PyCSP revision r147“. I: *http://code.google.com/p/pycsp* (dec. 2009). URL: http://code.google.com/p/pycsp/source/detail?r=147 (Lokaliseret d. 18. 01. 2010).

[20] L. Sha, R. Rajkumar og J.P. Lehoczky. „Priority inheritance protocols: An approach to real-time synchronization“. I: *IEEE Transactions on computers* 39.9 (1990), s. 1175–1185.

[21] Peter Welch og Neil Brown. *Communicating Sequential Processes for Java (JCSP)*. URL: http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

Appendix A

Testresultater

A.1 Testresultater for DES

Test	Resultat
Doctest: simulation.Simulation	ok
Doctest: simulation.io	ok
Doctest: guard.testsuite	ok
Doctest: alternation.Alternation	ok
Doctest: alternation.testsuite	ok
Doctest: channel.Channel	ok
Doctest: channel.testsuite	ok
Doctest: process.Parallel	ok
Doctest: process.Spawn	ok
Doctest: process.test_suite	ok
test_alternation (test_simulation.SimulationTestCase)	ok
test_buffer (test_simulation.SimulationTestCase)	ok
test_buffered_channels (test_simulation.SimulationTestCase)	ok
test_decompose (test_simulation.SimulationTestCase)	ok
test_io (test_simulation.SimulationTestCase)	ok
test_timers1 (test_simulation.SimulationTestCase)	ok
test_timers2 (test_simulation.SimulationTestCase)	ok
test_timers3 (test_simulation.SimulationTestCase)	ok
test_timers_time_in_past (test_simulation.SimulationTestCase)	ok
test_wait (test_io.TestCase)	ok

A.2 Testresultater for RTP

Test	Resultat
test_Alternation	ok
test_AlternationChoiceReader	ok
test_AlternationChoiceWriter	ok
test_AlternationExecuteReadDeadline	ok
test_AlternationExecuteSkipDeadline	ok
test_AlternationExecuteTimeoutDeadline	ok
test_AlternationExecuteWriteDeadline	ok
test_Alternationchoice1Deadline	ok
test_Alternationchoice2Deadline	ok
test_ChoicemultipleReader	ok
test_ChoicemultipleReader2	ok
test_ChoicemultipleWriter	ok
test_PoisonAndDeadline1	ok
test_PoisonAndDeadline2	ok
test_Reader_Inheritance	ok
test_RetireAndDeadline	ok
test_Writer_Inheritance	ok
test_channelpriority_from_low_deadline	ok
test_channelpriority_from_low_deadline2	ok
test_channelpriority_from_no_deadline	ok
test_channelpriority_from_no_deadline2	ok
test_readDeadline	ok
test_writeDeadline	ok
test_xreset_inheritance	ok
test_xreset_inheritance_from_two_step	FAIL

A.3 Testresultater for IP

Test	Resultat
test_Wait1 (test3.InteractiveTestCase)	ok
test_Wait2 (test3.InteractiveTestCase)	ok
test_Wait3 (test3.InteractiveTestCase)	ok
test_priority1 (test3.InteractiveTestCase)	ok
test_priority2 (test3.InteractiveTestCase)	ok
test_priority3 (test3.InteractiveTestCase)	ok

Appendix B

Eksempler

B.1 Eksempler til DES

Kodeuddrag B.1: WaTor i simulerings-versionen

```
from pycsp.simulation import *
from random import choice as randomchoice
from numpy import *
from pygame.locals import *
import pygame, sys, os

EMPTY, FISH, SHARK = range(3)
TYPE, AGE, MOVED, STARVED  = range(4)

type =  {
       0 : "EMPTY",
       1 : "FISH",
       2 : "SHARK"
       }

color  = {
      "blue": (0,0,205),
      "red" : (255,0,0)
      }

def _element(element):
  return "%s:age:%i moved:%i starved:%i"%(
    type[element[TYPE]],
    element[AGE],
    element[MOVED],
    element[STARVED])

class Point:
  def __init__(self,_x,_y):
    self.x = _x
    self.y = _y
  def __repr__(self):
    return "(%i,%i)"%(x,y)
  def getX(self):
    return self.x
  def getY(self):
```

```
      return self.y
38  def to_gui(self):
      return (self.x*multiplier,self.y*multiplier)
40
  @process
42 def worldpart (part_id):
    #indices for working in a shared matrix:
44  part_width = (world_width/world_parts)-2
    start_col = part_id*(part_width+2)
46  right_shadow_col = (start_col+part_width)%world_width

48  @io
    def main_iteration():
50      for i in range(world_height):
          for j in range(start_col,start_col+part_width+2):
52          f = world[j][i]
            f[MOVED] = 0
54
        for i in range(world_height):
56        for j in range(start_col,start_col+part_width):
            element_iteration(Point(j,i))
58
    def element_iteration(p):
60    f = world[p.getX()][p.getY()]
      if f[TYPE] == EMPTY or f[MOVED] == 1:
62      return
      fish = None
64    if f[TYPE] == SHARK :
        if f[STARVED] >= 3:
66        viz_die(p)
          for i in range(4):
68          f[i] = 0
          return
70      fish = getsurroundings(p,FISH)
      spawn = None
72    if fish:
        spawn = randomchoice(fish)
74    else:
        emptyspaces = getsurroundings(p, EMPTY)
76      if emptyspaces:
          spawn = randomchoice(emptyspaces)
78    if spawn:
        move(p,spawn)
80    else:
          f[AGE] += 1
82        if f[TYPE] == SHARK:
            f[STARVED] += 1
84
    def getsurroundings(p,_type):
86    empty = []
      x = p.getX()
88    y = p.getY()
      if world[x][(y-1)%world_height][TYPE] == _type:
90        empty.append(Point(x,(y-1)%world_height)) #Above
      if world[x][(y+1)%world_height][TYPE] == _type:
92        empty.append(Point(x,(y+1)%world_height)) #Below
      if world[(x-1)%world_width][y][TYPE] == _type:
```

```
94          empty.append(Point((x-1)%world_width,y)) #Left
        if world[(x+1)%world_width][y][TYPE] == _type:
96          empty.append(Point((x+1)%world_width,y)) #Right
        if empty == []:
98        return None
        return empty
100
    def move(f,t):
102       _from = world[f.getX()][f.getY()]
          _to = world[t.getX()][t.getY()]
104       if _from[TYPE] == SHARK:
            if _to[TYPE] == FISH:
106           _to[STARVED] = 0
              viz_die(t)
108         else:
              _to[STARVED] = _from[STARVED] + 1
110
          _to[AGE] = _from[AGE]+1
112       _to[MOVED] = 1
          _to[TYPE] = _from[TYPE]
114       viz_move(_to,f,t)

116       if (_to[TYPE] == FISH and _to[AGE]<3) or _to[AGE]<10 :
            for i in range(4):
118           _from[i] = 0
          else:
120         _to[AGE] = 0
            _from[AGE] = 0
122         _from[MOVED] = 1
            create_gui(_from[TYPE],f)
124       return _to

126   def viz_move(element,_from, _to):
        viz_die(_from)
128     create_gui(element[TYPE],_to)

130   def viz_die(_to):
          screen.fill(color["blue"],(_to.to_gui(),(multiplier,multiplier)))
132
    Wait(1)
134   for i in xrange(iterations):
        #Calc your world part:
136     main_iteration()
        Wait(1)
138     #Calc the two shadowrows
        for i in range(world_height):
140       for j in range(2):
            element_iteration(Point(right_shadow_col+j,i))
142     Wait(2)

144 @process
  def visualize():
146   for i in xrange(iterations):
        Wait(3)
148     pygame.display.flip()

150 def create_gui(type,point):
```

```
    if type == SHARK :
152   screen.blit(shark_img,(point.getX()*multiplier,
                point.getY()*multiplier))
154 if type == FISH :
      screen.blit(fish_img,(point.getX()*multiplier,
156               point.getY()*multiplier))

158 def create(type):
    x = random.randint(0,world_width)
160 y = random.randint(0,world_height)
    while not world[x][y][TYPE] == EMPTY:
162   x = random.randint(0,world_width)
      y = random.randint(0,world_height)
164 age = random.randint(0,9)
    if type == FISH:
166   age = random.randint(0,3)

168 world[x][y][AGE] = age
    world[x][y][MOVED] = 0
170 world[x][y][STARVED] = 0
    world[x][y][TYPE] = type
172 create_gui(type,Point(x,y))

174
  if __name__ == "__main__":
176 #Constants
    world_parts = 4
178 iterations = 40
    pct_fish = 0.76
180 pct_shark = 0.01
    multiplier = 10
182 window_size = 3
    img_size = (multiplier,multiplier)
184
    #Set GUI
186 pygame.init()
    shark_img = pygame.image.load(os.path.join("images","shark2_ms.jpg"))
188 fish_img = pygame.image.load(os.path.join("images","fish2_s.jpg"))
    shark_img = pygame.transform.smoothscale(shark_img,img_size)
190 fish_img = pygame.transform.smoothscale(fish_img,img_size)

192 assert shark_img.get_height() == shark_img.get_width()
    assert fish_img.get_height() == fish_img.get_width()
194 assert shark_img.get_bounding_rect() == fish_img.get_bounding_rect()

196 display = pygame.display.list_modes()[window_size]
    world_width = (display[0]/(multiplier*world_parts))*world_parts
198 world_height = display[1]/multiplier
    if window_size == 0 :
200   screen = pygame.display.set_mode((world_width*multiplier,
                                        world_height*multiplier))
202 else :
      screen = pygame.display.set_mode((world_width*multiplier,
204                                     world_height*multiplier))
    screen.fill(color["blue"])
206
    #Create shared data structure
```

```
208   assert 0 == world_width%world_parts
      starting_fish = int(world_width*world_height*pct_fish)
210   starting_sharks = int(world_width*world_height*pct_shark)

212   #make sure we have room for fish+sharks
      assert world_height*world_width >= starting_fish+starting_sharks
214   world = zeros((world_width,world_height,4),object)

216   #Populate fish
      for i in xrange(starting_fish): create(FISH)
218
      #Populate sharks
220   for i in xrange(starting_sharks): create(SHARK)

222   Parallel(
        [worldpart(i) for i in range(world_parts)],
224     visualize()
      )
```

Kodeuddrag B.2: Simpelt bankeksempel i simulerings-versionen

```
1  from pycsp.simulation import *
   from random import expovariate,seed
3  from heapq import *

5  seed(12)
   class Customer:
7    def __init__(self, name="",meanWT=10.0):
       self.name = name
9      self.waittime = round(expovariate(1/meanWT))

11   def __repr__(self):
       return "(%s,%s)"%(self.waittime,self.name)
13
   @process
15 def Generator(i,number,meanTBA, meanWT, customerWRITER):
     """Generaters a customer with a given time difference"""
17   for numberInserted in range(number):
       Wait(expovariate(1/meanTBA))
19     customerWRITER(
         Customer(name = "Customer%d:%02d"%(i,numberInserted),
21               meanWT = meanWT))

23   print "%64.0f: G%d: retires"%(Now(),i)
     retire(customerWRITER)
25

27 @process
   def Bank(meanWait,customerREADER):
29   """Handles the action inside the bank """
     t = False
31   customers =  []
     mon = Monitor()
33
     try:
35     while True:
         print "%94.0f: blocking wait to receive customer"%(Now())
```

```
37        msg = customerREADER()
          print "%94.0f: %s enter bank"%(Now(),msg.name)
39        heappush(customers,(Now()+msg.waittime,msg))
          mon.observe(len(customers))
41
          while len(customers)>0:
43          print "%94.0f: B: timeout is:%f"%(Now(),customers[0][0]-Now())
            (g,msg) = Alternation([(customerREADER,None),
45                                 (Timeout(seconds=customers[0][0]- Now()),None)
                                ]).select()
47
            if g == customerREADER:
49            heappush(customers,(Now()+msg.waittime,msg))
              print "%94.0f: %s enter bank"%(Now(),msg.name)
51
            else:
53            ntime,ncust = heappop(customers)
              print "%94.0f: %s left bank"%(Now(),ncust.name)
55
            mon.observe(len(customers))
57          print "%94.0f: Length of queue in bank %d"%(Now(),len(customers))

59    except ChannelRetireException:
        """All generators have retired just empty the queue"""
61      print "%94.0f: All genreators have retired"%Now()
        while(len(customers)>0):
63          Wait(customers[0][0]-Now())
            ntime,ncust = heappop(customers)
65          mon.observe(len(customers))
            print "%94.0f: %s left bank"%(Now(),ncust.name)
67
        Histo = mon.histogram()
69      plt = SimPlot()
        plt.plotHistogram(Histo,
71                        xlab='length of queue',
                          ylab='number of observation',
73                        title="# customers in bank",
                          color="red",width=1)
75      plt.mainloop()
        return
77
  if __name__ == "__main__":
79  print "main starting"
    nprocesses = 10
81  mon = Monitor()
    customer = Channel(buffer=9,mon = mon)
83  numberCustomersprprocess=10
    meanTBA = 3.0
85  meanWT =  5.0
    b = Bank(meanWT,+customer)
87  Parallel(
      b,
89    [Generator(i,numberCustomersprprocess,meanTBA, meanWT,-customer)
        for i in range (nprocesses)]
91  )
    print b.executed
```

Kodeuddrag B.3: Avanceret bankeksempel i simulerings-versionen

```
from pycsp.simulation import *
from random import expovariate,seed
from heapq import *
seed(12)

class Customer:
  def __init__(self, name="",meanWT=10.0):
    self.name = name
    self.waittime = expovariate(1/meanWT)

  def __repr__(self):
    return "%s,%s"%(self.waittime,self.name)


@process
def Generator(i,number,meanTBA, meanWT, customerWRITER):
  """Generaters a customer with a given time difference"""
  for numberInserted in range(number):
    Wait(expovariate(1/meanTBA))
    customerWRITER(
      Customer(name = "Customer%d:%02d"%(i, numberInserted),
               meanWT = meanWT))

  print "%64.0f: G%d: retires"%(Now(),i)
  retire(customerWRITER)

@process
def Bank(customerREADER):
  """Handles the action inside the bank """
  try:
    while True:
      print "%94.0f: B: waits for customer"%Now()
      customer = customerREADER()

      print "%94.0f: B: adding a customer %s to queue"%(
        Now(),customer)
      Wait(customer.waittime)
      print "%94.0f: B: customer %s exits queue"%(
        Now(),customer)

  except ChannelRetireException:
      print "%94.0f: B: got retire"%(Now())
      pass

if __name__ == "__main__":
  nprocesses = 1
  numberCustomers=5
  queue = Channel(buffer=numberCustomers*nprocesses)
  meanTBA = 3.0
  meanWT = 3.0
  Parallel(
    Bank(+queue),
    [Generator(i,numberCustomers,meanTBA, meanWT, -queue)
      for i in range (nprocesses)]
  )
```

B.2 Eksempler til RTP

Kodeuddrag B.4: Slagterieksemplet RTP-versionen

```
from pycsp.deadline import *
import random, sys, time , heapq, math, scipy

avg_convert_processing = 0.66
avg_camera_processing = 0.135
avg_analysis_processing = 0.45

cam_iter =   200000
conv_iter = 1000000
ana_iter =   700000
dummy_iter =  50000
std = 0.2
concurrent = 5.0
avg_arrival_interval = (avg_camera_processing+
                        avg_convert_processing+
                        avg_analysis_processing)/concurrent

time_to_camera_deadline = (avg_camera_processing+
                           avg_convert_processing)*1.3
time_to_deadline = (avg_camera_processing+
                    avg_convert_processing+
                    avg_analysis_processing)*(1.22*concurrent)

pigs_to_simulate =  100
number_of_simulations = 10

class Pig:
  def __init__(self,_id, arrivaltime,ran,deadline = time_to_deadline):
    self.arrivaltime = arrivaltime
    self.deadline = arrivaltime+deadline
    self.id = _id,
    self.donetime = arrivaltime
    self.done = False
    x = ran.uniform(0,9)

    if x<1: self.normal = False
    else : self.normal = True

    self.wait = []
    self.accum = []

  def __repr__(self):
    sun = 0
    for x in self.wait : sun += x[1]
    return "%s\ttotal time inc queue : %0.3f, total processtime = %0.3f = %s"%(
      self.done,self.donetime-self.arrivaltime,sun, self.accum)

def dummywork(iterations):
    #Estimating Pi.
    temp = 0
    import time
    for k in xrange(int(iterations)):
        temp += (math.pow(-1,k)*4) / (2.0*k+1.0)
```

```
            k +=1

@process
def background_dummywork(dummy, time_out):

    @process
    def internal_dummy(_id,dummy_in, dummy_out,time_out,work = dummy_iter):
        try:
            time_spent=0
            if _id == 0: dummy_out(time_spent)

            while True:
                time_spent = dummy_in()
                time_spent -= Now()
                dummywork(work)
                time_spent += Now()
                dummy_out(time_spent)

        except ChannelPoisonException:
            poison(dummy_in,dummy_out)
            if _id == 0: time_out(time_spent)

    dummyC = Channel()
    Parallel(
        internal_dummy(0,+dummy,-dummyC,time_out),
        internal_dummy(1,+dummyC,-dummy,time_out))

@io
def sleep(n):
    import time
    if n>0: time.sleep(n)

@process
def feederFunc(robot, analysis, dummy,ran, data = avg_arrival_interval):
    NextpigArrival = Now()+ran.gauss(data, data*std)
    ThispigArrival = Now()
    Set_deadline(NextpigArrival-Now())
    for x in xrange(pigs_to_simulate):
        try:
            if x % 10 == 0 : print "\t\t",x
            pig = Pig(x,ThispigArrival,ran)
            robot(pig)
            if pig.arrivaltime+time_to_camera_deadline-Now()>0:
                camchannel = Channel()
                conChannel = Channel()
                feederChannel = Channel()
                cam = cameraFunc(+feederChannel,-camchannel,ran)
                conv = convertFunc(+camchannel,-conChannel,ran)
                ana =  analysisFunc(+conChannel,analysis,ran)
                Set_deadline((pig.arrivaltime+time_to_camera_deadline)-Now(),cam)
                Set_deadline((pig.arrivaltime+time_to_deadline)-Now(),conv)
                Set_deadline((pig.arrivaltime+time_to_deadline)-Now(),ana)
                Spawn(cam,conv,ana)
                Alternation([
                    {((-feederChannel),pig):None},
                    {Timeout(NextpigArrival-Now()):None}
                    ]).execute()
```

```
111
                Remove_deadline()
113             ThispigArrival = NextpigArrival
                NextpigArrival = ThispigArrival+ran.gauss(data, data*std)
115             Set_deadline(NextpigArrival-Now())
                if ThispigArrival>Now() : sleep(ThispigArrival-Now())
117
            except DeadlineException:
119             Remove_deadline()
                NextpigArrival = NextpigArrival+ran.gauss(data, data*std)
121             Set_deadline(NextpigArrival-Now())
        poison(robot)
123     poison(dummy)

125 @process
    def cameraFunc(in0,out0,ran , data = avg_camera_processing):
127     try:
            val0 = in0()
129         if Now() - val0.arrivaltime   < time_to_camera_deadline :
                val0.accum.append(Now()-val0.arrivaltime)
131             dummywork(ran.gauss(cam_iter,cam_iter*std))
                out0(val0)
133             Remove_deadline()

135     except DeadlineException:
            Remove_deadline()
137         poison(in0,out0)

139 @process
    def convertFunc(in0,out0,ran , data = avg_convert_processing):
141     try:
            val0 = in0()
143         if val0.deadline>Now():
                Set_deadline(val0.deadline-Now())
145             val0.accum.append(Now()-val0.arrivaltime)
                dummywork(ran.gauss(conv_iter,conv_iter*std))
147             out0(val0)
                Remove_deadline()
149
        except DeadlineException:
151         Remove_deadline()
            poison(in0,out0)
153
        except ChannelPoisonException:
155         Remove_deadline()
            poison(in0,out0)
157
    @process
159 def analysisFunc(in0,out0,ran , data = avg_analysis_processing):
        try:
161         val0 = in0()
            if val0.deadline>Now():
163             Set_deadline(val0.deadline-Now())
                val0.accum.append(Now()-val0.arrivaltime)
165             dummywork(ran.gauss(ana_iter,ana_iter*std))
                out0(val0)
167             Remove_deadline()
```

```
169     except DeadlineException:
            Remove_deadline()
171
        except ChannelPoisonException:
173         Remove_deadline()

175 @process
    def robotFunc(feeder,analysis,ran, statC, data = time_to_deadline):
177     next_deadline = {}
        try:
179         @choice
            def process_pig(channel_input):
181             if channel_input.id not in next_deadline :
                    next_deadline[channel_input.id] = channel_input
183
            @choice
185         def process_pig2(channel_input):
                if channel_input.deadline>Now():
187                 channel_input.done = True

189             channel_input.accum.append(Now()-channel_input.arrivaltime)
                channel_input.donetime = Now()
191             next_deadline[channel_input.id] = channel_input

193         while True:
                alt = Alternation([
195                 {feeder :process_pig()},
                    {analysis:process_pig2()}
197             ]).execute()

199     except ChannelPoisonException:
            good = 0
201         bad = 0
            normal = 0
203         for key, pig in  next_deadline.items():
                if pig.done : good +=1
205             else : bad +=1
                if pig.normal : normal +=1
207         statC(float(good)/(pigs_to_simulate))

209 @process
    def Work(statC,timeC):
211     start = Now()
        for x in range(number_of_simulations):
213         print x," / ", number_of_simulations
            ran = random.Random(x)
215
            robotC = Channel("robot")
217         analysisC = Channel("analysis")
            dummyC = Channel("dummy")
219
            feed = feederFunc(-robotC,-analysisC, -dummyC, ran)
221         rob = robotFunc(+robotC,+analysisC,ran,statC)

223         Parallel(
              #1*background_dummywork(dummyC,timeC),
```

```
225             feed,
                rob
227         )
        poison(statC, timeC)
229
    @process
231 def Statistic(statC):
        stc = []
233     try:
            while True:
235             stce = statC()
                stc.append(stce)
237
        except ChannelPoisonException:
239         print "number pigs processed in time:"
            print "mean: %0.2f"%scipy.mean(stc)
241         print "std variance : %0.2f\n"%scipy.std(stc)

243 @process
    def StatisticTime(statC):
245     stc = []
        try:
247         while True:
                stac = statC()
249             stc.append(stac)

251     except ChannelPoisonException:
            print "Time spent in dummy:"
253         print "mean: %0.3f"%scipy.mean(stc)
            print "std variance : %0.3f\n"%scipy.std(stc)
255

257 processeschan = Channel()
    timechan = Channel()
259 Parallel(
        Work(-processeschan,-timechan),
261     Statistic(+processeschan),
        StatisticTime(+timechan)
263 )
    print "cam deadline:\t%3f\ndeadline:\t%3f"%(
265     time_to_camera_deadline,time_to_deadline)
    print "avg processing time: ",avg_camera_processing+
267                                 avg_convert_processing+
                                    avg_analysis_processing
269 print "concurrent: ",concurrent
    print "RTP version"
```

B.3 Eksempler til IP

Kodeuddrag B.5: Ureksemplet RTP-versionen

```
from pycsp.deadline import *
import random, sys, time , heapq, math, scipy

class Time:
    def __init__(self,hour, minut,second):
        self.hour = hour
        self.minut = minut
        self.second = second
        self.internal_seconds = ((hour*60+minut)*60+second)
        #print self.internal_seconds
    def __str__(self):
        return "%02.0f:%02.0f:%02.0f"%(
               self.hour,self.minut, self.second)

class Watch:
    def __init__(self):
        self.present_time = None

    def print_time(self):
        offset = Now()-start_time
        print """time in watch is %02.0f:%02.0f:%02.0f
        (real time spent %f = diff %f)"""%(
                 self.present_time.hour,
                 self.present_time.minut,
                 self.present_time.second,
                 offset,
                 offset-self.present_time.internal_seconds)
    def set_time(self,timestamp):
        self.present_time = timestamp
        self.offset = Now()-start_time-self.present_time.internal_seconds

def dummywork(iterations):
    #Estimating Pi.
    temp = 0
    import time
    for k in xrange(int(iterations)):
        temp += (math.pow(-1,k)*4) / (2.0*k+1.0)
        k +=1

@process
def StatisticTime(statC):
    stc = []
    try:
        while True:
            stac = statC()
            stc.append(stac)

    except ChannelPoisonException:
        print "Time spent in dummy:"
        print "mean: %0.5f"%scipy.mean(stc)
        print "std variance : %0.5f\n"%scipy.std(stc)

@process
```

```
54 def background_dummywork(dummy, time_out,stat_out):
       @process
56     def internal_dummy(_id,dummy_in, dummy_out,time_out,work = 50000):
           try:
58             time_spent=0
               n = 0
60             if _id == 0: dummy_out(time_spent)
               while True:
62                 time_spent = dummy_in()
                   n+=1
64                 time_spent -= Now()
                   b4 = Now()
66                 dummywork(work)
                   time_spent += Now()
68                 stat_out(Now()-b4)
                   dummy_out(time_spent)
70         except ChannelPoisonException:
               poison(dummy_in,dummy_out,stat_out)
72             if _id == 0:
                   print time_spent
74
       dummyC = Channel()
76     Parallel(
           internal_dummy(0,+dummy,-dummyC,time_out),
78         internal_dummy(1,+dummyC,-dummy,time_out))

80 @io
   def sleep(n):
82     import time
       if n>0: time.sleep(n)
84

86 @process
   def watch_process(time_in_channel,ack_out_channel):
88     watch = Watch()
       try:
90         offsets = []
           while True:
92             watch.set_time(time_in_channel())
               offsets.append(watch.offset)
94             watch.print_time()
               ack_out_channel(True)
96     except ChannelPoisonException:
           print "mean: %0.3f"%scipy.mean(offsets)
98         print "std variance : %0.3f\n"%scipy.std(offsets)
           print "len: %0.3f, max:%d, min : %f"%(
100                len(offsets),max(offsets),min(offsets))
           poison(time_in_channel,ack_out_channel)
102
   @process
104 def set_time(_id,timeset, time_out_Channel, ack_in_channel):
       try:
106        wakeup_time = timeset.internal_seconds-(Now()-start_time)
           Set_deadline(wakeup_time+1)
108        Wait(wakeup_time)
           woke_up = (Now()-start_time)-timeset.internal_seconds
110        time_out_Channel(timeset)
```

```
            ack = ack_in_channel()
112     except DeadlineException:
            print "skipped one update"
114
    @process
116 def close_watch(time_channel,dummy_channel,time):
        Wait(time)
118     poison(time_channel,dummy_channel)

120 time_channel = Channel()
    ack_channel = Channel()
122 dummy_channel = Channel()
    dummy_timer_channel = Channel()
124 stat_channel = Channel()
    time_steps = 100
126 start_time = Now()
    Parallel(
128     watch_process(+time_channel,-ack_channel)
        ,[set_time(i,Time(i/(60*60),i/60,i%60),-time_channel,+ack_channel)
130         for i in range(time_steps)]
        ,close_watch(+time_channel,+dummy_channel,time_steps)
132     ,10*background_dummywork(dummy_channel,-dummy_timer_channel,-stat_channel)
        ,StatisticTime(+stat_channel)
134 )
```