Autonome agenter som programmel-arkitektur på mobile robotter

Ken Frøslev

LYNGBY 2001
EKSAMENSPROJEKT
NR. 888

IMM

Trykt af IM, DTU

Forord

Dette projekt er skrevet som eksamensprojekt på *CST* som er en del af *Informatik og matematisk modellering* ved Danmarks Tekniske Universitet. Projektet er skrevet med professor Jørgen Fischer Nilsson og lektor Hans Bruun som vejledere. En mindre del af projektet er udført ved *Automation sektionen* under *Ørsted-DTU*. Herfra har Ole Jannerup, Nils Andersen, og Ian David Braithwaite været kontakt personer. Desuden har Lene, Chris, Pia og Bjarke været behjælpelige på hver deres måde.

Tak til alle de ovenstående.

Ken Frøslev, august 2001

Abstract

The aim of this project is to design and implement a suitable autonomous agent architecture for a collection of physical robots, based on a symbolic/logic representation and reasoning system.

The robots sensors and actors are controlled throw a software interface, which only gives primitive control of the robot. On top of this interface is implemented an extension module, which provides a better level of abstraction to the robot. The module contains position maintaines, noise reduction for the distance measurements equipment and some algorithms for controlling the robots wheels.

After a series of experimentations with the robots the possibilities and limitations for the robot are sketched . Which leads to a description of some experimental-scenarios. For each scenario is discus of possible solutions.

From the discussions of the scenarios is designed a agent-architecture based on a BDI-model. Beside the three component *belief*, *desire* and *intention* is a communication component and a reactive component found useful.

Unfortunately the robots have not been operational doing the last period of the project. So test and demonstration is quite limited. Therefore the conclusion is based on a discussion of the architecture.

Key words: Intelligent Robots, Intelligent Agents, Autonomous Agents, Implementing Agents.

Indhold

Kapitel 1

Indledning

I dette projekt ønskes at udforske mulighederne for at anvende en agent-arkitektur på mobile robotter. Som udgangspunkt haves en konkret robot, som arkitekturen skal implementeres på.

Arkitekturen skal såvidt muligt designes til en bred klasse af mobile robotter, men skal naturligvis indeholde løsninger til de konkrete problemstillinger for denne robot.

Der lægges vægt på at nå hele vejen fra analyse og design af en arkitektur til implementeringen af denne, således at designet kan demonstreres på robotten.

1.1 Hvad er en agent?

En agent er et individ som fungerer interaktivt i et miljø. Et individ kan i denne forbindelse være mange ting som for eksempel: Et menneske, et dyr, en robot eller et program. En autonom agent er således en agent, som ud fra sine mål og observationer af omgivelserne, kan foretage autonome handlinger i det omgivende miljø. I resten af rapporten ses dog udelukkende på software-agenter.

En software-agent er et computer-program, som besidder nogle særlige egenskaber. Hvilke egenskaber der gør forskellen mellem et "almindeligt"

program og en software-agent er der ikke nogen fast definition på. I [13] beskrives en software-agent, som et program der er er opbygget omkring nogle mentale tilstande som **viden**, **tro**, **intention** og **forpligtigelse**. Af andre egenskaber er blandt andet foreslået: **autonom**, **måldreven**, **fleksibel**, **kommunikerende**, eller **reaktiv**.

Ud fra denne "definition" af en agent kan der ses på forskellige velkendte program-typer som for eksempel operativsystemer, epost-klienter eller epost-servere.

En epost-klient modtager en besked (eller brev) fra en bruger og sender det til en epost-server. Desuden hentes breve fra epost-serveren, som vises for brugeren. Hvilken epost-server der skal anvendes er den del af epost-klientens viden. Klienten er reaktiv da den "opfanger", hvornår brugeren ønsker at sende beskeden, og den er kommunikerende, da den kommunikerer med epost-serveren. Alligevel betragter man ikke en epost-klient som en agent.

En epost-server har nogle af de samme egenskaber som en epost-klient. Men den er væsentligt "mere kommunikerende" da den kommunikerer med mange andre epost-servere og -klienter. Desuden er serveren fleksibel da den ofte er i stand til at sende post ad alternative ruter, hvis nogle systemer er "nede". Med lidt god vilje kan man også betragte epost-serveren som måldrevet, idet den som mål har at videre sende post (muligvis med forskellige prioritet) mellem forskellige klienter og servere. Men igen en epost-server betragtes almindeligvis ikke som en agent.

På samme måde kan de fleste af ovenstående agent egenskaber sættes i forbindelse med et operativsystem. Operativsystemet udfører "ønsker" fra brugere og ofte mange ønsker fra flere brugere. Flere oplysninger findes automatisk over netværket (*nameserver*, *browsemaster* m.m.). Og ofte er operativsystemer i stand til at starte forskellige "vedligeholdelses-programmer" på tidspunkter, hvor det antages ikke at ville forstyrre brugerne. Selv om operativsystemer måske betragtes mere som agenter end epost programerne, så opfatter man alligevel ikke helt et operativsystem som en agent.

Grunden til at det virker forkert at bruge agentbegrebet om de tre ovenstående eksempler er, at de fleste kender den funktionelle virkemåde af programmerne (ihvertfald på at et vist abstraktionsniveau.), og har man allerede en forståelse af programmet, virker det overflødigt at indføre agentbegrebet. Desuden findes agentbegrebet hverken for computerens hardware eller for oversætteren som oversætter programmet. Agentbegrebet anven-

des altså udelukkede for at man kan beskrive de mekanismer, der sker i et program.

Det er derfor oplagt kun at bruge begrebet software-agent om et program, når det hjælper til forståelsen af programmets funktionalitet og opbygning.

Et program eller et system er altså IKKE en agent, hvis det kan beskrives simplere uden agentbegrebet, og en agent er et program, hvis funktionalitet nemmest beskrives med agentbegrebet.

Denne definition gør det nemt at udelukke programmer som værende en agent, men der mangler stadig en idé om hvilke programmer, der er nemmest at beskrive som agent. Det nærmeste man kommer på dette må være programmer, som er opbygget med en eller flere af de tidligere nævnte agent-egenskaber.

1.1.1 Er en robot en agent?

En robot er mange ting. I industrien er robotter mest brugt til samlebånds produktion, f.eks som svejserobotter. I dette projekt betragtes robotter, som kan bevæge sig fysisk rundt i "verden".

Typisk består en robot af:

1. Noget mekanik, som udgør den fysiske struktur, og forskellige bevægelige dele.
2. Nogle motorer (elektriske, hydrauliske el. lign.), som får robotten til at bevæge sig.
3. Noget elektronik som styrer motorerne.
4. Noget software som styrer elektronikken (grænsen mellem hvad der er elektronik og software kan være noget flydende).

Hvis softwaren er en software-agent, er hele systemet så en agent, eller er det en robot med agent-styring? Det svarer til at diskutere, om en agents aktorer og sensorer er en del af agenten, eller om de er en del af det system agenten fungerer i.

Ses de fire ovenstående dele som en enhed, hvor softwaren er opbygget som en agent, så vil den samlede enhed have agentegenskaber. Derfor vælges at betragte det samlede system (robot + agent-software) som en agent, og en robot betragtes som bestående af delene 1, 2 og 3.

1.2 Dansk oversættelse af *belief*, *desire* og *intention*

Et speciel klasse af agenter er BDI-agenter. BDI er en forkortelse for **B***elief*-**D***esire*-**I***ntention*. Som på dansk kan oversættes til *tro*, *mål* og *intention*.

Der er dog et problem med denne dansk oversættelse, efter som de danske ord kan have flere betydninger, som ikke svarer helt til de engelske ord. Derfor er her en kort forklaring til oversættelsen. Betydningerne af de danske ord er fra "Politikkens Retskrivnings- og betydningsordbog (1996)".

Belief oversættes til det danske ord *tro*. Problemet ved dette er at *tro* ofte relateres med en "religiøs tro". Tro har da også to betydninger:

- 1: "Det at være sikker på noget uden at have vished for det eller overbevisning"
- 2: "Religiøs eller overtroisk overbevisning. Eks. den kristne tro; troen på nisser."

og i denne sammenhæng er det den første betydning der henvises til. *Belief* kunne således også være oversat til *overbevisning*.

Desire kan oversættes til *ønske*, *lyst* eller *begær*. Men ordet *mål* findes her mere passende. Også ordet mål har flere betydninger, men betydningen der søges her er:

- "hensigt"eller "noget man ønsker skal blive opfyldt"

Intention kan oversættes til hensigt eller intention. Betydningen af *hensigt* er "plan, ide eller formål", mens betydningen af *intention* er "hensigt". Oversættelsen *intention* vælges da hensigt nemmere forveksles med mål.

I resten af rapporten vil denne klasse af agenter dog stadig blive kaldt BDI-agenter. Men *Belief*, *Desire* og *Intention* oversættes til *Tro*, *mål* og *intention*. Ofte kaldes disse entiteter for **mentale attributter**, men her vælges at betragte dem som **komponenter**.

1.3 Indhold i rapporten

Kapitel 2 beskriver baggrundsteori om agenter og teknikker, der typisk anvendes ved design og opbygning af agenter. Herefter i kapitel 3 beskrives

den robotplatform, som anvendes som forsøgsplatform i projektet. Kapitel 4 undersøger hvilke forsøgsscenarier, der er mulige med robotterne og diskuterer mulige løsninger til disse scenarier. Ud fra disse scenarier designes en agent-arkitektur, som beskrives i kapitel 5. Implementeringen af denne arkitektur beskrives i kapitel 6, og endelig i kapitel 7 demonstreres implementeringen.

Kapitel 2

Agenter

I dette kapitel gennemgås teorien for opbygning og implementering af agenter. Specielt ses på klassen af agenter som kaldes BDI-agenter.

2.1 Logik sprog

En agent har typisk brug for et logiksystem til at lagre tro, viden, mål og lignende, og for at kunne ræsonnere over denne information. Et sådant system kaldes i [8] for et repræsentations- og ræsonneringssystem. Systemet kan opbygges omkring mange forskellige teorier. Eksempelvis med første-ordens prædikatlogik, som en relationsdatabase med nogle passende operationer, eller af nogle datastrukture udtrykt i et passende programmeringssprog.

Her vælges at se på DATALOG, selv om det ikke er lige så udtryksfuldt som ex. første-ordens prædikatlogik, men tilgengæld er det mere implementerings venligt.

2.1.1 DATALOG

DATALOG er en forkortelse af *Database logic*. DATALOG er beskrevet i [8] og [6].

DATALOG kan bruges som repræsentations- og ræsonneringssystem i en agent, hvis agentens verden kan beskrives under følgende tre antagelser:

- **Individer og relationer**: Verden kan beskrives som individer og relationer mellem disse. Individer kan være stort set alt som kan navngives. F.eks.: personer, farver, tal, følelser osv.
- **Definit viden:** Agentens viden (herunder tro o.a.) består af definitte og positive udsagn. Hvilket vil sige at man ikke kan have udsagn af type: "RobotA er **ikke** i rum 2" eller af typen: "RobotA er i **enten** rum 1 **eller** i rum 2". Definit viden er af typen: "RobotA **er i** rum 1".
- **Endeligt domæne:** Der er et endeligt antal individer (af interesse) i agentens verden, som alle kan tildeles et unikt navn.

Klausuler i DATALOG

DATALOG består af *definitte klausuler*, som er klausuler på formen:

$$A \leftarrow B_1 \wedge B_2 \wedge . \cdot \cdot \wedge B_n \tag{2.1}$$

hvor A og B_i er prædikater af formen:

$$p(t_1, \ldots, t_k) \tag{2.2}$$

og t_1 til t_k er termer og p er prædikatets navn. En term kan være en variabel, en konstant eller en funktor. En funktor er en sammensat term:

$$f(t_1, \ldots, t_k) \tag{2.3}$$

hvor t_1 til t_k igen er termer.

En klausul med $n = 0$ kaldes for et *atomisk udsagn*, og hvis $n > 0$ kaldes klausulen for en *regel*. A kaldes klausulens *hoved*, mens $B_1 \wedge \ldots \wedge B_n$ kaldes *kroppen*. En klausul, atomisk udsagn eller en *regel*, siges at være på grundform, hvis den ikke indeholder variabler. Standardkonventionen er at konstanter starter med et lille bogstav eller et tal (eks. *agentX, f12, 16*), mens variable starter med stort bogstav (eks. *X, Agent, Felt*). Prædikatnavne skrives med små bogstaver. Hvis et prædikat ikke har nogle termer, dvs. $k = 0$ undlades parentesen typisk. Eksempelvis skrives p i stedet for $p()$.

En regels hoved er gyldigt såfremt reglens krop er gyldigt. Reglens hoved kan altså udledes med:

$$\frac{hoved \leftarrow krop \qquad krop}{hoved}$$

Eller for formen som 2.1:

$$\frac{(A \leftarrow B_1 \wedge \ldots \wedge B_n) \qquad (B_1 \wedge \ldots \wedge B_n)}{A} \tag{2.4}$$

En viden-database, eller videnbase, beskrevet i DATALOG er en mængde af definitte klausuler. En videnbase, KB_{robot}, som beskriver noget af en simpel "robot/kasse" verden kunne være:

$$\begin{aligned} KB_{robot} = \quad \{ \quad & robot(r_1), && (2.5)\\ & box(b_1), && (2.6)\\ & moveable(X) \leftarrow robot(X), && (2.7)\\ & moveable(X) \leftarrow box(X), && (2.8)\\ & can\_carry(X,Y) \leftarrow robot(X) \wedge box(Y), && (2.9)\\ \} & \end{aligned}$$

2.5 og 2.6 er atomiske udsagn som angiver at henholdsvis r_1 og b_1 er en robot og en kasse. 2.7 er en *regel* som angiver at X er et flytbart individ, hvis X er en robot, mens 2.8 angiver at X er flytbar, hvis X er en kasse. Endelig angiver reglen 2.9 at X kan bære Y, hvis X er en robot, og Y er en kasse.

For at kunne anvende videnbasen er det nødvendigt at kunne stille forespørgsler (som for en almindelig database). I DATALOG har en forespørgsel ($query$) formen:

$$?query$$

hvor *query* har formen $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ og hvor p_i er atomiske udsagn. Svaret på en forespørgsel på grundform er *sand* (forespørgslen er en logisk konsekvens af videnbasen) eller *fail* (forespørgslen kan ikke udledes af videnbasen). For en forespørgsel, som ikke er på grundform, er svaret enten *fail* eller en variabel tildeling, som gør forespørgslen *sand*. At et udtryk g er en logisk konsekvens af videnbasen KB udtrykkes som:

$$KB \models g$$

Hvis forespørgslen $?robot(r_1)$ stilles til ovenstående videnbase, KB_{robot}, er svaret *sand* da $KB \models robot(r_1)$ På forespørgslen:

$$?moveable(X)$$

findes to forskellige svar:

$$X = r_1 \quad \text{og} \quad X = b_1$$

da der gælder både $KB_{robot} \models moveable(r_1)$ og $KB_{robot} \models movable(b_1)$.

For at kunne "svare" på en forespørgsel er det nødvendigt med et *bevissystem*. Dette kan gøres med enten en *bottom-up* (data-dreven) eller en *top-down* (mål-dreven) procedure. *Bottom-up* procedure er den mest simple, men den har sine begrænsninger pga. kombinatorisk eksplosion. *Top-down* er mere komplicere og er den procedure, der benyttes i prolog-systemer. Hvis et bevissystem kan bevise g fra videnbasen KB skrives:

$$KB \vdash g$$

Et bevissystem er *sundt* hvis:

$$KB \vdash g \Rightarrow KB \models g$$

og er *komplet* hvis:

$$KB \models g \Rightarrow KB \vdash g$$

Almindeligvis vil man kræve at et bevissystem er både sundt og komplet.

2.1.2 *Bottom-up* bevisprocedure

Teknikken for et mål-drevet *bottom-up* bevissystem er at udlede alt hvad der kan bevises fra en given videnbase. Dette gøres ved at udvælge en regel af formen 2.1 hvor $n \geq 1$, hvor kroppen b_1 til b_n allerede findes som

atomiske udsagn i videnbasen. Kan sådan en regel ikke findes, er der ikke flere udsagn, der kan udledes. En forspørgsel:

$$?Q_1 \wedge \ldots \wedge Q_n$$

er nu bevist af systemet hvis Q_1 til Q_n alle findes i videnbasen. Er dette ikke tilfældet må bevissystemet svare *fail*. *Fail* betyder altså, at bevissystemet ikke kan bevise forespørgslen.

Denne algoritme virker fint for videnbaser på grundform. Men for videnbaser indeholdende variable, må alle regler instantieres til grundform. Det gøres ved at se på alle kombinationer af variable tildelinger. I videnbasen KB_{robot} findes to konstanter r_1 og b_1, og to variable X og Y. Det giver $2^2 = 4$ forskellige variable tildelinger:

$$\begin{array}{lcl} x = r_1 & \wedge & y = r_1 \\ x = r_1 & \wedge & y = b_1 \\ x = b_1 & \wedge & y = r_1 \\ x = b_1 & \wedge & y = b_1 \end{array}$$

Hvilke for de tre regler 2.7, 2.8 og 2.9 giver følgende instanser:

$$\begin{array}{l} moveable(r_1) \leftarrow robot(r_1), \\ moveable(b_1) \leftarrow robot(b_1), \\ moveable(r_1) \leftarrow box(r_1), \\ moveable(b_1) \leftarrow box(b_1), \\ can\_carry(r_1, r_1) \leftarrow robot(r_1) \wedge box(r_1), \\ can\_carry(r_1, b_1) \leftarrow robot(r_1) \wedge box(b_1), \\ can\_carry(b_1, r_1) \leftarrow robot(b_1) \wedge box(r_1), \\ can\_carry(b_1, b_1) \leftarrow robot(b_1) \wedge box(b_1), \end{array}$$

Generelt gælder at hvis en videnbase indeholder k konstanter, og en regel består af v variable, har denne regel k^v instanser. Det vil sige, at denne mål-drevene algoritme virker fint for "små" videnbaser, men har videnbasen f.eks 100 konstanter og 100 variabler, vil algoritmen få en uoverskuelig kørselstid.

2.1.3 *Top-down* bevisprocedure

SLD-resolution er en *Top-down* eller mål-dreven algoritme, som bruges i prolog. Modsat *bottom-up* algoriten starter *top-down* med forespørgslen - som også kaldes *mål-klausulen*. Først ses på algoritmen for en videnbase på grundform. For mål-klausulen:

$$?Q_1 \wedge \ldots \wedge Q_i \wedge \ldots \wedge Q_n \tag{2.10}$$

Der udvælges, i videnbasen, en klausul på formen:

$$Q_i \leftarrow B_1 \wedge \ldots \wedge B_n \tag{2.11}$$

Det vil sige en klausul, hvis hoved indgår i mål-klausulens krop. Klausulen bruges nu til at substituere Q_i i målklausulen, som herefter er:

$$?Q_1 \wedge \ldots \wedge Q_{i-1} \wedge B_1 \wedge \ldots \wedge B_n \wedge Q_{i+1} \wedge \ldots \wedge Q_n \tag{2.12}$$

Bemærk at hvis den valgte klausul 2.11 har $n = 0$ bliver mål-klausulen 2.12:

$$?Q_1 \wedge \ldots \wedge Q_{i-1} \wedge Q_{i+1} \wedge \ldots \wedge Q_n \tag{2.13}$$

Som er "mindre" end den oprindelige mål-klausul 2.10

Denne substituering fortsættes indtil hele mål-klausulen er bevist.

Det kritiske i algoritmen er at udvælge den "rigtige" regel til at substituere med. Da et forkert valg kan medføre, at algoritmen ikke kan gennemføre et bevis, som kunne have været bevist med et andet valg. Et eksempel på dette er vist på side 51 i [8].

Da algoritmen ikke på forhånd ved hvilken regel, der skal udvælges, behøves en "back-tracking" funktionalitet, som hvis algoritmen fejler, kan springe tilbage til det sted i beviset, hvor der sidst er fortaget et valg. Herfra fortsættes beviset, så med et andet valg. Algoritmen fejler således først endeligt, når alle kombinationer af valg er afprøvet.

For videnbaser på grundfor, bliver det lidt mere kompliceret. For at kunne substituere 2.11 i 2.10, kræves at der findes en *most general unifier* for de to

klausuler. En *unifier* er en variabeltildeling som gør to klausuler ækvivalent. Eksempelvis for:

$$p(X, Y, a, b) \quad \text{og} \quad p(X, b, Z, a) \tag{2.14}$$

gælder

$$\begin{aligned} p(X, Y, a, b)\theta &= p(X, c, Z, b)\theta && (2.15)\\ &\Leftrightarrow && \\ p(X, c, a, b) &= p(X, c, a, b) && (2.16)\\ & && (2.17) \end{aligned}$$

for

$$\theta = \{Y \mapsto c, Z \mapsto a\} \tag{2.18}$$

Hvor θ er en variabeltildeling, og $p(\ldots)\theta$ angiver variabeltildelingen påtrykt $p(\ldots)$.

most general unifier (angivet som θ_{mgu}) er **den** variabeltildeling mellem to udtryk, hvorom der gælder at alle andre *unifier* θ' findes en substituation σ, således at:

$$\theta' = \theta_{mgu} \circ \sigma \tag{2.19}$$

hvor $\theta_1 \circ \theta_2$ angiver den samlede substituering er variabeltildelingerne θ_1 og θ_2.

most general unifier fines ved unifikations procedure hvor alle termer i de to udtryk sammenlignes. Dette er beskrevet nærmere i [8] og [6]. En unifikationsprocedure fejler hvis to udtryk ikke har en *unifier*.

2.1.4 *"Negation as failure"*

Ofte vil det være ønskeligt at kunne konkludere at noget er ikke-gyldigt eller falsk. Dette kan gøres ved at indføre en antagelse om, at agentens viden er

komplet. I [8] kaldes denne antagelse for **"closed world assumption"** eller **"complete knowledge assumption"**. Antagelsen betyder, at agenten ved alt, hvad der er værd at vide om verden. For eksemplet, KB_{robot} på side 9 kan udledes:

$$moveable(r_1) \quad \wedge \quad moveable(b_1) \tag{2.20}$$

Såfremt antagelsen om komplet viden er gyldig er r_1 og b_1 de eneste individer hvor om "moveable" gælder. Derfor gælder at:

$$moveable(X) \Leftrightarrow X = r_1 \vee X = b_1 \tag{2.21}$$

og dermed også:

$$\neg moveable(X) \Leftrightarrow X \neq r_1 \wedge X \neq b_1 \tag{2.22}$$

For at vise at $X \neq c$, hvor c er en konstant, kan indføres en ækvivalent-operator, som kan sammenligne termer. En simplere, og ofte dækkende løsning er, at indføre en antagelse om "unikke navne". Antagelsen betyder, at to forskellige konstanter referer til to forskellige individer.

De to ovenstående antagelser betyder, at hvis et *komplet* bevissystem ikke kan udlede en forespørgelse *query* fra en viden base kan konkluderes $\neg$*query*.

2.1.5 Prolog

Som tidligere beskrevet er et logik-udtryks-sprog ikke til den store nytte i en implementering, hvis der ikke er et bevissystem til rådighed. Prolog er et logik programmeringssprog som bl.a. kan ræsonnere over klausuler beskrevet i DATALOG.

Prolog er altså oplagt til agentsystemer, som er baseret på logiske ræsonneringer. Problemet er, at et agentsystem typisk består af andet end logiske ræsonneringer. Eksempelvis består agenter-arkitekturer, til robotter og lignende typisk af en hybrid struktur, med en *delibertiv del* og en *reaktiv del*. Netop for at adskille den overvejende og ræsonnerende del fra det reaktive system som typisk styrer robottens sensorer og aktorer.

GNU Prolog er et prolog-system udviklet under GNU-projektet[1], og er et godt bud på et system der kan bruges til at implementere agenter. (Systemet kan findes på: `http://pauillac.inria.fr/~diaz/gnu-prolog/`). Systemet har flere fordele som kunne udnyttes. For det første kan systemet generere *binær-kode*. Det vil sige prolog-programmet kan oversættes til maskinkode, som kan køres afhængigt af prologsystemet. En anden fordel er, at sproget er udvidet med blandt andet system-kald til operativsystemet og et *socket* interface (til netværkskommunikation). Dette betyder at der kan udvikles nogen grad af "almindeligt programmel" i prolog sproget. Desuden har GNU Prolog et *bidirectional* interface mellem Prolog og C. Hvilket betyder, at man i princippet kan kalde Prolog rutiner fra et C program eller omvendt.

2.2 Kommunikation mellem agenter

Kommunikation mellem agenter består af udveksling af beskeder. Typisk har en besked form som angivet i [10]:

$$I(S, R, \varphi, G, \omega) \tag{2.23}$$

hvor:

- I er beskedtypen, som f.eks er "spørgsmål", "svar" eller "information"
- S og R er henholdsvis sender og modtager af beskeden.
- φ er beskedens egentlige indhold
- G er en begrundelse for φ (Typisk et logisk bevis for φ)
- ω er en priotering af beskeden. Typisk en værdi i intervallet $[0; 1]$.

Der findes et utal af beskrevne sprog til agentkommunikation. [12] giver en kort gemmengang af nogle af disse.

Kommunikation kan være meget simpel, eks. ved at agenterne kun udveksler information igennem forud bestemte spørgsmål/svar. Eller det kan være mere kompliceret, hvor agenterne eksempelvis planlægger i samarbejde igennem kommunikation. [7] beskriver et kommunikationssystem, som lader agenterne forhandle og argumentere, hvor argumenterne er logiske beviser for deres påstande.

[1] Se nærmere på www.gnu.org

2.3 Planlægning for agenter

I dette afsnit beskrives forskellige teorier for planlægning. Planlægning er et vigtigt område for agenter, men er også et meget stort område. Det er derfor ikke muligt at beskrive hele området. De to systemer, der er beskrevet herunder er rimeligt brede og kan anvendes på mange problemer.

2.3.1 STRIPS

Planlægningssystemet *STRIPS* er en forkortelse for "STanford Research Institute Problem Solver" og var et af de første planlægningssystemer. Det er beskrevet bl.a. i [8] og [11]. Der skelnes mellem STRIPS repræsentation og STRIPS planlægningssystemet. Hvor STRIPS repræsentation er en måde at repræsentere aktioner på, og STRIPS planlægningssystemet er en planlægningsalgoritme, som ofte benytter STRIPS repræsentation.

For at kunne benytte STRIPS skal både systemets aktuelle tilstand og målet være beskrevet med atomiske udsagn. Desuden skal systemet have en liste af aktioner, hvis "funktion" er veldefineret.

STRIPS repræsentation

I STRIPS repræsentationen er en aktion beskrevet af et navn og tre mængder: *precondition*, *delete list* og *add list*. *precondition* er en mængde af atomiske udsagn, som skal være opfyldt før aktionen kan udføres. *Delete list* er en mængde af atomiske udsagn som ikke længere er gyldige efter aktionen er udført, og *add list* er de udsagn, som bliver gjort gyldige af aktionen.

STRIPS demonstreres ofte med en stationær robotarm, der skal stable klodser. Sådan et eksempel kan ses i [11]. Her ses på et eksempel med en mobil robot, som kan bære kasser. Udsagnet *nextto(A, B)* angiver at A og B er placeret ved siden af hinanden. Sådan en robot ville typisk have bl.a. de to aktioner: Pickup(X) og Drop(X), som henholdsvis samler et objekt op, og sætter det igen. I STRIPS repræsentationen kunne Pickup(X) aktionen således ud:

```
Pickup(X):
   precondition: box(X), nextto(robot, X)
   add list    : carrying(robot, X)
   delete list : nextto(robot, X)
```

Det vil sige for at en robot kan samle X op, skal robotten være placeret ved siden af X, og X skal være en kasse. Efter aktionen bærer robotten X, og kassen er derfor ikke længere placeret ved siden af robotten. På samme måde kan Drop(X) aktionen beskrives:

```
Drop(X):
   precondition: carrying(X)
   add list    : nextto(robot, X)
   delete list : carrying(robot, X)
```

Robotten skal bære X for at kunne sætte den. Efter aktionen er X placeret ved siden af robotten, og robotten bærer ikke længere på X.

På samme måde kan alle aktioner, der er til rådighed for systemet, beskrives.

STRIPS planlægningssystemet

Planlægning kan ses som en søgning i en tilstandsgraf, hvor knuder i grafen er tilstande, og kanter mellem knuderne er de mulige aktioner. Planen er en rute i grafen fra tilstanden, som svarer til systemets udgangpunkt til en tilstand, der opfylder det eller de ønskede mål. Afhængig af systemets natur kan bruges forskellige graf-søgnings algoritmer. Typisk bruges dybde-først, eller brede-først søgninger. Problemet ved denne grafsøgning er, at for komplicerede systemer kan tilstandsrummet blive meget stort, og en søgning kan derfor tage meget langt tid. En andet problem er, at man skal være i stand til at sammenligne tilstande, for at undgå løkker i grafen.

STRIPS planlægningssystemet fungerer ved at søge baglæns fra målet mod start tilstanden. STRIPS starter med at lægge målet eller målene på en "mål-stak". Fra stakken udvælges en mål, hvortil der findes en aktion som har målet i sin *add list*. Herefter kaldes kaldes STRIPS planlægningssystemet rekusivt med den udvalgte funktion som mål. Resultatet af dette rekusive kald er en plan, som er en del af den endelig plan. Den aktuelle tilstand beregnes som starttilstanden plus summen af *add list* udsagnene fra den netop beregnede delplan, minus summen af *delete list* udsagnene fra samme plan. Dette fortsættes, indtil den aktuelle tilstand opfylder alle målene på mål-stakken.

Som for SLD-resolution indgår et valg i algoritmen, det er derfor nødvendigt med en *backtracking* funktionalitet, så algoritmen ikke fejler på grund af et forkert valg.

Denne STRIPS algoritme har to problemer. Det første er, at den kan havne i en uendelig løkke. Det andet er, at et delmål som allerede er opfyldt kan blive "ødelagt" af senere aktioner. I [11] beskrives en forbedret algoritme, som via en mere intelligent håndtering af mål-stakken, undgår disse problemer.

2.3.2 Situations-kalkule

Situations-kalkule (fra engelsk *Situation Calculus*) er beskrevet bl.a. i [8].

I situations-kalkule haves en beskrivelse af systemets begyndelses tilstand, kaldet *init*. Alle andre gyldige tilstande beskrives som *do(A, S)*. Der beskriver den tilstand systemet vil være i efter udførelse af aktion A i tilstand S. Foreksempel beskriver $do(move(robot, pos_1, pos_2), init)$ tilstanden for systemet hvor "robotten" har flyttet sig fra position pos_1 i initiel tilstanden til positionen pos_2. Tilstanden efter n aktioner (A_1 til A_n) er således beskrevet ved:

$$do(A_1, do(A_2, do(\ldots, do(A_n, init)\ldots)))$$

Verden beskrives med passende prædikater, hvor sidste "argument" angiver i hvilken tilstand udsagnet er gyldigt. At en *robot* initielt er placeret ved position pos_1 beskrives foreksempel ved $at(robot, pos_1, init)$. Hvis position pos_1 ligger ved siden af position pos_2 skrives $adjacent(pos_1, pos_2, S)$, hvor S er en "fri variabel" da udsagnet gælder for alle tilstande.

De aktioner der er til rådighed for systemet, beskrives med relationen $poss(A, S)$. Som angiver, at aktionen A er mulig i tilstanden S. Eksempelvis vil aktionen *move* beskrives som:

$$\begin{aligned} poss(move(Robot, Pos_1, Pos_2), S) \quad \leftarrow \quad & robot(Robot, S), \\ & adjacent(Pos_1, Pos_2, S), \\ & at(Robot, Pos_1, S) \end{aligned}$$

Altså for at *Robot* kan flytte sig fra Pos_1 til Pos_2 i tilstanden S, kræves det at i denne tilstand gælder at *Robot* er en robot, Pos_1 og Pos_2 er forbundet og at *Robot* er placeret ved Pos_1.

Konsekvenser af aktioner aksiomeres ved at beskrive, hvornår udsagn omkring verden er gyldige. Eksempelvis beskriver:

$$at(Robot, Pos_2, do(move(Robot, Pos_1, Pos_2), S)) \leftarrow poss(move(Robot, Pos_1, Pos_2), S)$$

at *Robot* er placeret ved position Pos_2 efter en *move* aktion, som har flyttet robotten til Pos_2, når blot *move* aktionen er mulig.

På tilsvarende måde beskrives alle de udsagn, der bruges til at beskrive den konkrete verden.

Planlægning med situations-kalkule

Som for STRIPS kan situations-kalkule betragtes som en søgning i en tilstandsgraf. Den mest oplagte måde at bruge situations-kalkule er ved at betragte ovenstående udtryk som klausuler i et DATALOG system. Målet bruges som forespørgsel til SLD-algoritmen. Algoritmen fejler, hvis der ikke findes en plan. Planen er den række af aktioner, som SLD-algoritmen har brugt til at reducere forspørgelsen med.

I [8] argumenteres for at bruge situations-kalkule frem for STRIPS, da alt hvad der kan udtrykkes i STRIPS også kan udtrykkes i situations-kalkule, men ikke omvendt. Samme sted angives en metode til direkte at repræsentere STRIPS aktioner i situations-kalkule.

2.4 BDI-agenter agenter

Man kan beskrive virkemåden af BDI-agenter, som at agenten ud fra sin **tro** skal skabe en **intention**, som fører til udførelsen af agentens **mål**.

I [9] argumenteres for at BDI-agenter skal bestå af præcis de tre komponenter: *tro*, *mål* og *intention*, mens [7] beskriver BDI-agenter som bl.a. har en kommunikationskomponent. Desuden vil fysiske agenter typisk have brug for en eksekveringskomponent og andre komponenter kunne også være praktiske. Forskellen må ligge i hvilke dele af et system man vælger at kalde for "agenten". I det følgende betragtes en BDI-agent, som en agent der som minimum indeholder komponenterne tro, mål og intention.

2.4.1 Karakteristika for BDI-agenter

Gao og Georgeff, som er nogle af bagmændene bag DBI-agenter har i [9] beskrevet en række karaktistika som et real-tidssystem skal besidde for at være velegnet for en BDI-arkitektur.

1. På ethvert tidspunkt kan systemets omgivelser ændre sig i flere forskellige retninger.
2. På ethvert tidspunkt har systemet mange forskellige mulige aktioner, som kan udføres.
3. Det er flere forskellige opgaver, som systemet er bedt om at løse.
4. Hvilken aktion som (bedst) opnår det ønskede resultat er afhængig af omgivelsernes tilstand og uafhængig af systemets tilstand.
5. Systemet kan kun "se" omgivelsernes tilstand lokalt. (Det vil sige systemet kan ikke skabe et komplet billede af omgivelsernes tilstand, på et vilkårligt tidspunkt)
6. Hastigheden hvormed systemet kan beregne og udføre aktioner, er rimelig, sammenlignet med hvor hurtigt systemets omgivelser ændrer sig.

Ud fra disse seks karakteristika beskrives i det følgende funktionen og nødvendigheden af BDI-agentens tre komponenter.

2.4.2 En BDI-agents tro

Kilder til tro

En agents *tro* er den information agenten har. Denne information opbygges af agentens sensorer (her under kommunikation). Som eksempel ses på en agent til at bestemme i hvilken rækkefølge fly skal lande i en luft havn[2]. Når et nyt fly dukker op på agentens radar (sensor) vil agenten få en tro på at flyet eksisterer, og hvor langt væk det er.

Nødvendigheden af tro

Hvis en agent var i stand til når som helst at skabe et komplet billede af verdens tilstand ved brug af sine sensorer, så ville det ikke være nødvendigt

[2] Eksemplet er inspireret fra [9]

med tro for agenten. Med dette er netop ikke tilfældet ifølge karakteristika 5. Agentens verdensbillede må derfor opbygges i takt med, at agenten får mere og mere information. Denne information lagres som tro. Agenten skal tilstræbe at have så meget information om verden som muligt for at kunne vælge den optimale aktion (karakteristika 4) i en given situation.

2.4.3 En BDI-agents mål

Mål er den eller de "opgaver" som agenten skal løse. Agentens mål er dens motivation, og agenten kan have mange mål på samme tid. Hvis en agent har flere mål kan disse være modstridende.

Nødvendigheden af mål

For at en agent kan være mål-drevet, er det naturligvis nødvendigt med repræsentation af dens mål. Desuden bruges en mål-komponent typisk til at generere nye mål, og til at prioritere og udvælge mål.

Kilder til mål

En agent kan selv opstille mål (typisk vil dette være et "del-mål" for at opnå et "overordnet-mål"), eller agenten kan overtage mål fra andre agenter. I ovenstående eksempel er agentens "overordnede-mål" at "bestemme hvilken rækkefølge fly skal lande i". Dette er ikke nødvendigvis direkte beskrevet i agentens målkomponent. Men det ligger indirekte i den måde, hvorpå agenten er designet. Eksempel: Når et fly dukker op på agentens radar kunne agenten opstillet et nyt mål: Bestem ETA[3] for flyet. Eller operatøren af systemet kunne bede systemet om at give landingstilladelse til et bestemt fly, så ville dette blive tilføjet til agents mål. I første eksempel opstiller agenten selv et mål, og i andet eksempel overtages et mål fra en anden agent (operatøren).

2.4.4 En BDI-agents intention

Agentens intention kan betragtes som agentens plan, og hænger derfor nøje sammen men agentens planlægningsmekanisme.

[3]ETA: Estimated time of arrival (estimeret tid for ankomst)

Nødvendigheden af intention

Problemet med planlægningen for et system som opfylder karakteristika 1, 2 og 6 er, at verden kan ændre sig, mens systemet er i gang med planlægningen, eller mens planen er under udførelse. Når en ændring af omgivelserne registreres er der to muligheder. Enten stoppes systemet og en ny planlægning udføres efterfulgt af eksekvering. Eller man kan fuldføre planlægningen og eksekveringen. Førstnævnte giver et ineffektivt system, da hele eller en del af løsningen skal beregnes mange gange. Anden mulighed medfører, at systemet kan udføre ineffektive eller ulovlige aktioner.

Intentionskomponenten repræsenterer den sidst producerede plan. Planen, eller dele af den, fjernes fra intentions-komponenten, når den er udført, når den ikke længere er mulig, eller hvis den er i modstrid med en ny plan. Planlægningsmekanismen aktiveres, enten når en ændring i omgivelserne observeres eller som et led i agentens cyklus.

Intentionskomponenten giver på denne måde et kompromis mellem trofasthed overfor gamle beslutninger, og mulighed for at ændre planen undervejs.

Kilder til intention

Agentens intention bestemmes af dens planlægningssystem, men en intention kan også direkte eller indirekte overtages fra en anden agent. Sidst nævnte som del af agentens samarbejdsevne.

I eksemplet ovenfor er et af agentens mål at give landingstilladelse til et fly. Når agentens planlægningsmekanisme vælger, at dette skal udføres (når landingsbanen er ledig mm.) skabes intentionen "send landingstilladelse til flyet".

2.4.5 Notation

Til at udtrykke tro, mål og intention for en agent bruges de tre prædikater B, D og I. Flere steder (fx i [3] og [7]) benyttes notation $B_a\alpha$ for agent, a, tror udsagnet α. Dette kan også bruges til at beskrive, hvad en agent tror om en anden agent. At agent, a, tror at agent b, har intentionen α kan skrives som:

$$B_a(I_b(\alpha))$$

Hvilket dog giver både syntaktiske og semantiske problemer i forhold til klassisk første-ordens logik. Beskrevet nærmere i [13].

For at undgå problemer som det ovenstående og for at få en notation, der ligger tættere på en implementering vælges notationen:

$$C_i : info$$

Hvilket angiver at informationen, *info*, er lageret i komponenten, C, for agent i. Informationen er beskrevet med de tre prædikater $B(j,u)$, $D(j,u)$ og $I(j,u)$, hvor, u, er et udsagn, som gælder om agenten j. Udsagnet u er typisk beskrevet med første-ordens prædikat logik. Det betyder at prædikaterne B, D og I må betragtes som *meta-level* prædikater. På meta niveau er B, D og I sædvanlige prædikater, mens udsagnet u er et funktionssymbol.

Eksemplet med lufthavns-agenten beskrevet i de ovenstående afsnit kan beskrives for agent, a, med:

$$B_a : B(a,\ distance(plane01,\ 10km)\ )$$

Hvilket betyder at agent, a, tror at afstanden til fly, *plane01* er *10km*.

$$D_a : D(a,\ calculateETA(plane01)\ )$$

$$D_a : D(a,\ giveOKforLanding(plane02))$$

Hvilket betyder at agenten, a, har som mål at beregne ETA for flyet, *plane01* og give landingstilladelse til flyet *plane02*.

$$I_a : I(a,\ sendOKforLanding(plane01))$$

Hvilket betyder at agent, a, har som intention at sende landingstilladelse til flyet *plane01*.

Eksemplet udvides nu med yderligere en agent, b, som tænkes placeret i en anden lufthavn. Hvis agent a (igennem en kommunikationskanal) har fortalt agent b om sine intentioner, kan skrives:

$$B_b : I(a,\ sendOKforLanding(plane01))$$

Hvilket betyder at der i agent b's tro komponent haves oplysningen at agent a har "sendOKforLanding(plane01)" som intention.

2.4.6 Logik for BDI-agentens komponenter

Ofte bruges et modal logik system i agentens komponenter. I [7] og [9] anvendes KD45 (også kaldet svag-S5).

Tro

$$\mathbf{K}\quad B: \quad B(\varphi \to \psi) \to (B(\varphi) \to B(\psi)) \tag{2.24}$$
$$\mathbf{D}\quad B: \quad B(\varphi) \to \neg B(\neg\varphi) \tag{2.25}$$
$$\mathbf{4}\quad B: \quad B(\varphi) \to B(B(\varphi)) \tag{2.26}$$
$$\mathbf{5}\quad B: \quad \neg B(\varphi) \to B(\neg B(\varphi)) \tag{2.27}$$

Mål

$$\mathbf{K}\quad D: \quad D(\varphi \to \psi) \to (D(\varphi) \to D(\psi)) \tag{2.28}$$
$$\mathbf{D}\quad D: \quad D(\varphi) \to \neg D(\neg\varphi) \tag{2.29}$$

Intention

$$\mathbf{K}\quad I: \quad I(\varphi \to \psi) \to (I(\varphi) \to I(\psi)) \tag{2.30}$$
$$\mathbf{D}\quad I: \quad I(\varphi) \to \neg I(\neg\phi) \tag{2.31}$$

2.4.7 Relationer mellem BDI-agentens komponenter

I princippet kan man designe de relationer mellem agentens komponenter, som findes passende til agentens formål. De fleste artikler holder sig dog til de tre typer BDI-agenter, kaldet **stærk realisme**, **realisme** og **svag realisme**.

Stærk realisme

Stærk realisme er når *intention* er den delmængde af *mål*, som igen er en del mængde af *tro*. Det vil sige:

$$I(\alpha) \rightarrow D(\alpha) \tag{2.32}$$
$$D(\alpha) \rightarrow B(\alpha) \tag{2.33}$$

Det vil sige at hvis agenten ikke tror et udsagn, så vil den hverken have dette udsagn som mål eller som intention. Dette kan nemmere ses hvis 2.32 og 2.33 omskrives med reglen $p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$:

$$\neg B(\alpha) \rightarrow \neg D(\alpha) \tag{2.34}$$
$$\neg D(\alpha) \rightarrow \neg I(\alpha) \tag{2.35}$$

Realisme

Realisme er når agentens *tro* er en delmængde af agentens *mål*, som igen er en delmængde af agentens *intention*:

$$B(\alpha) \rightarrow D(\alpha) \tag{2.36}$$
$$D(\alpha) \rightarrow I(\alpha) \tag{2.37}$$

Det vi sige at hvis en agent tror det, så er det også agentens mål og intention.

Svag realisme

Svag realisme er en kombination af stærk realisme og realisme. Agenten har ikke et udsagn som mål, hvis det negerede udsagn er agentens tro, agenten har ikke et udsagn som intention, hvis det negerede udsagn er agentens mål, og agenten har ikke et udsagn som intention, hvis det negerede udsagn er agentens tro. Dette udtrykkes i:

$$B(\neg\alpha) \rightarrow \neg D(\alpha) \tag{2.38}$$
$$D(\neg\alpha) \rightarrow \neg I(\alpha) \tag{2.39}$$
$$B(\neg\alpha) \rightarrow \neg I(\alpha) \tag{2.40}$$

2.5 Implementering af BDI-agenter

Mens der er mange artikler, der beskæftiger sig med agent-systemer, er der forholdsvis få, der omhandler implementering af agenter. I [9] beskrives i nogen grad hvordan BDI-agenter gøres mere implementerings venlige, men der er stadig langt til en egentlig implementering. [10] beskriver en systematisk modulopbygning af agenter, hvor beskrivelsen af en agent arkitektur opdels i:

- Komponenter : De enheder eller moduler agenten er opbygget af.
- Logik : Til hver komponent er tilknyttet et logiksystem.
- Teorier : For hver komponent findes en mængde af regler, beskrevet i det tilhørende logiksystem.
- Broregler : Regler som relaterer formler mellem de enkelte komponenter.

Typisk vil BDI-agenter have tro-, mål- og intention-komponenter, men afhængig af agentens formål kan diverse komponenter være nødvendige. Logikken som tilknyttes en komponent kan være DATALOG som beskrevet i afsnit 2.1.1, første-ordens-prædikatlogik eller noget helt tredie. *Teorien* er de formler, som udgør komponentens funktionalitet. Broregler sammenknytter komponenterne. En broregel har formen:

$$\frac{c_1 : \psi, \quad c_2 : \varphi}{c_3 : \theta} \tag{2.41}$$

hvor c_i angiver en komponent. c_1, c_2 og c_3 behøver ikke alle at være forskellige. Broreglen tilføjer θ til komponent c_3, hvis ψ og φ gælder i henholdsvis c_1 og c_2. Der findes to varianter af broreglen:

$$\frac{c_1 > \psi, \quad c_2 : \varphi}{c_3 : \theta} \tag{2.42}$$

som angiver at ψ bliver fjernet fra c_1 når reglen anvendes, og

$$\frac{c_1 : \psi, \quad c_2 : \varphi}{c_3 : \theta}[t] \tag{2.43}$$

som angiver at θ først tilføjes c_3 t tidsenheder efter betingelserne er opfyldt.

Alle komponenter og broregler implementeres seperat så de eksekveres parallelt. Det betyder at broreglerne kontinuerligt checker om deres betingelser er opfyldt.

2.6 Agent arkitektur for fysiske robotter

Flere steder i litteraturen er det konkluderet, at en agent, baseret på et logiske ræsonnering- og repræsentationssystem, ikke er velegnet til at "styre" en fysisk agent. Løsningen er at benytte en hybrid arkitektur, hvor agenten opdeles i en reaktiv og en deliberativ del. Dette forslåes bl.a. i [5] og [13].

Den reaktive del står for kontakt til robottens sensorer og aktorer, og løser de simple situationer, der ikke kræver et større overblik. Den deliberative del står for den overordnede planlægning. Ofte fungerer denne model ved at den deliberative del udformer en plan, som den reaktive udfører. Hvis der opstår problemer undervejs, prøver den reaktive del først selv at løse problemet, og hvis den ikke kan klare problemet, overtager den deliberative del.

Kapitel 3

Robotplatformen

I dette kapitel beskrives den robotplatform, som der eksperimenteres med i dette projekt, og som agent-arkitekturen skal designes til og implementeres på. Kapitlet består af to dele; først i afsnit 3.1 beskrives robotplatformen, og i afsnit 3.2 beskreves et softwaremodul, som beregner robottens position, kalibrerer sensorer m.m.

Robotten er udviklet og bygget af *Institut for automation* på DTU, mens software modulet er udviklet i forbindelse med dette projekt. Robotterne har fået navnet **smr** (**s**mall **m**obile **r**obots), og vil i det følgende blive kaldt smr-robotterne.

3.1 "Small mobile robots"

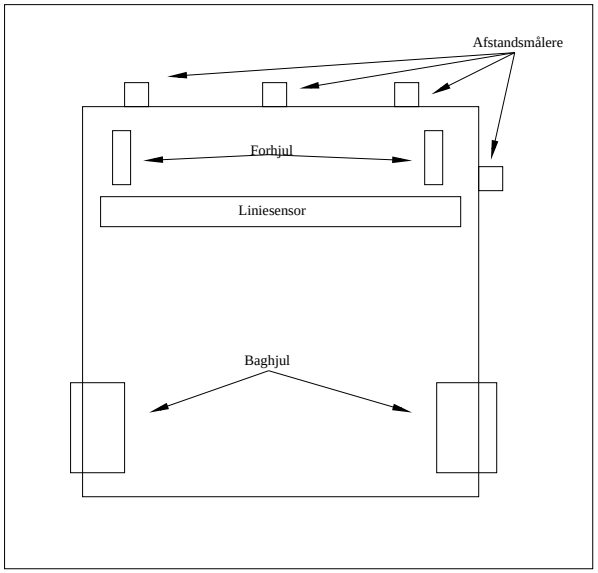
Her under beskrives smr-robotterne og det tilhørende software-interface. Dette projekt har ikke haft nogen indflydelse på robotternes design.

3.1.1 Mekanikken og Elektronikken

Robotterne er opbygget af en stålramme, med fire hjul, to motorer, to hjulsensorer, fire afstandsmålere, og en liniesensor. I bilag B kan ses billeder af smr-robotterne og figur 3.1 viser en skitse af dimensionerne på en smr.

De to motorer er via et gear forbundet til hver sit baghjul. Forhjulene er blot "støttehjul". Robotten styres således udelukkende af baghjulene (styringsprincippet kendes fra kampvogne). På akslen til hvert baghjul sidder en hjulsensor, som måler hjulets bevægelse. Tre af afstandsmålerne sidder foran på robotten, og den sidste sidder på højre side. Se figur 3.1. Liniesensoren, som gør robotten i stand til at følge en linie i gulvet sidder naturligvis under robotten.

Elektronikken består, udover sensorer og motorer, af standard PC hardware, med et trådløst netkort. Desuden er robotten udstyret med et batteri. Drifttiden af batteriet er ikke kendt præcist, men forsøg har vist at robotten kan køre længe nok til (ikke for lange) forsøg og demonstrationer. På IAU gættes på en drifttid på op til en time.



Figur 3.1: Skitse over en smr-robot

```
struct smr
  unsigned int read_flags;
  unsigned int wait_flags;
  struct {
    uint16_t encoder;
    int8_t speed;
  }
  left, right;
  uint8_t ir[SMR_IR_N];
  uint8_t ls[SMR_LS_N];

struct smr *smr_connect(char *hostname, int port);
void smr_disconnect(struct smr *);
int smr_read(struct smr *);
int smr_write(struct smr *);
```

Figur 3.2: Interfacet til smr driveren

3.1.2 Softwaren

Robotterne er udstyret med standard PC-hardware, og kan derfor bruge mange forskellige operativsystemer. IAU har valgt at benytte Linux på robotterne. Aflæsning af sensorer og kommandoer til motorerne sker via en driver (*demon*), hvis nærmere virkemåde ikke er beskrevet. Interfacet til denne driver er beskrevet herunder.

3.1.3 Interface til smr

Interfacet til robotterne er vist lidt simplificeret på figur 3.2 (se det fulde interface i filen `smr.h` i appendix C). Interfacet består af en datastruktur og fire funktioner. De to funktioner `smr_connect(...)` og `smr_disconnect(...)` henholdsvis opretter og afbryder en forbindelse til smr driveren. Mens `smr_read(...)` og `smr_write(...)` henholdsvis skriver til aktorer og læser fra sensorer.

Datastrukturen indeholder data fra sensorer og (nye) kommandoer til aktorerne. De enkelte dele af strukturen er beskrevet herunder:

`read_flags` : Aflæsning af hvilke sensorer der er blevet aflæst
`wait_flags` : Angiver hvilke sensorer der ønskes aflæst
`left.encoder` : Aflæsning af bevægelse af venstre hjul i 2000-del omgang
`right.encoder` : Aflæsning af bevægelse af højre hjul i 2000-del omgang
`left.speed` : Angiver ny fart på venstre hjul (-128 til 127)
`left.speed` : Angiver ny fart på højre hjul (-128 til 127)
`ir[SMR_IR_N]` : Aflæsning af afstandsmålere. Værdi mellem 0 og 255 for hver måler
`ls[SMR_LS_N]` : Aflæsning af liniesensor. Værdi mellem 0 og 255 for hver måler.

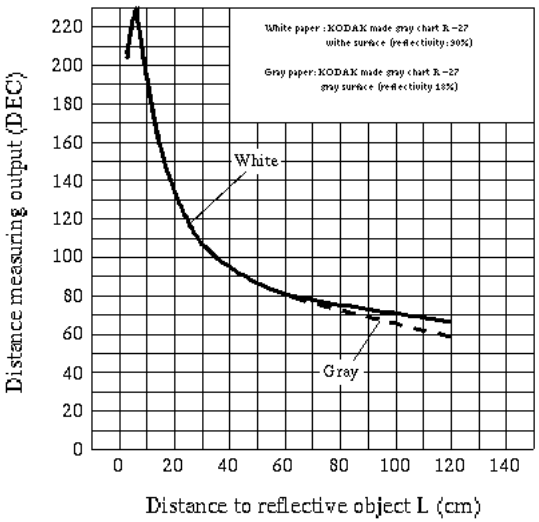
`SMR_IR_N` og `SMR_LS_N` angiver henholdsvis antallet af afstandsmålere og opløsningen på liniesensoren. (Den anvendte smr har fire afstandsmålere og otte punkter på liniesensoren).

3.1.4 Sensorer

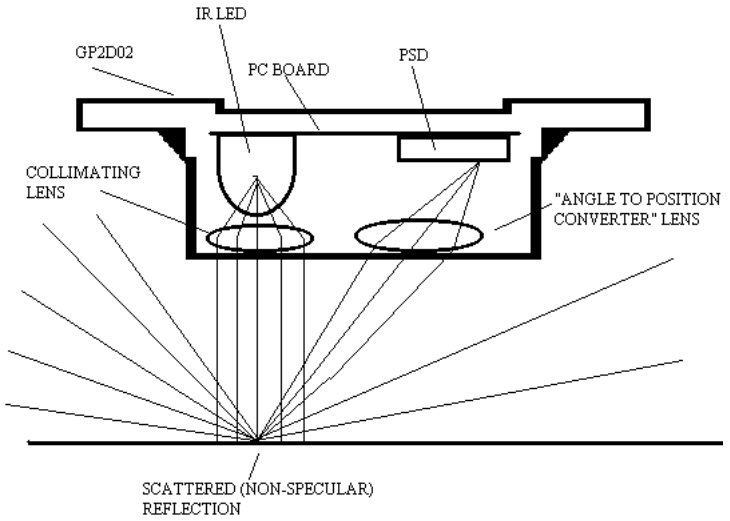
Smr'en har tre forskellige sensorer: liniesensor, hjulsensor og afstandsmålere. Liniesensoren er ikke videre interessant for dette projekt og bliver derfor ikke beskrevet nærmere. Hjulsensorerne tæller hjulenes bevæglese i 1/2000 omgange og bruges til positionsbestemmelse. Hvilket er beskrevet i afsnit 3.2.1. Afstandsmålerne er af typen *Sharp GP2D02*. Data om denne komponent er fra [2], hvorfra målinger og figurer i dette afsnit er lånt.

Sensoren er designet til at måle afstande fra 10 cm til 80 cm. Den fungerer ved at udsende infrarødt lys og måle vinklen på reflektionen. Se figur 3.4. Udlæsningen fra sensoren er en værdi mellem 0 og 255 (8-bit udlæsning), som er proportional med vinklen på det reflekterede lys. Det betyder at den aflæste værdi fra sensoren skal omregnes til en afstand. Figur 3.3 viser fabriksdata over sammenhængen mellem udlæsning fra sensoren og den faktiske afstand. Bemærk "knækket" i toppen af kurven, der gør at udlæsninger over 200 ikke kan omregnes til en entydig afstand.

I [2] er omregningen mellem udlæsning fra sensoren og den målte afstand (med simpel trigometri) bestemt til:



Figur 3.3: Fabriks-datablad for afstandsmåler



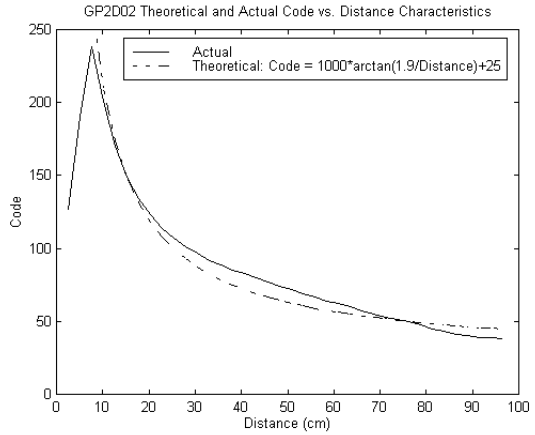
Figur 3.4: Skitse over Sharp GP2D02 afstands måler. Figuren er fra [2]

$$output = K_1 \times \arctan(K_2/Distance) + K_3 \tag{3.1}$$

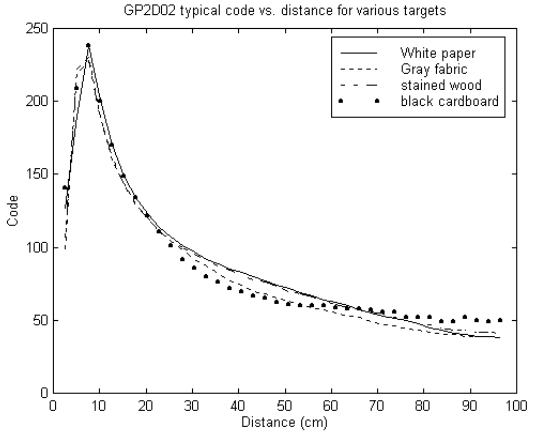
Hvor K_1 er skalering til intervallet $[0-255]$, K_2 er afstanden mellem sensorens sender og modtager og K_3 er et offset. K_1 og K_2 angives til at være tilnæmelsesvis ens for alle afstandsmålere af denne type, mens K_3 varierer en del.

I [2] er konstanterne for en konkret afstandsmåler bestemt til $K_1 = 1000$, $K_2 = 1.9$ og $K_3 = 25$. Ligning 3.1 er afbilledet med disse værdier i figur 3.5, sammen med faktiske målinger fra den afstandsmåler, hvor til konstanterne er fundet. Som det kan ses er der en del forskel mellem de teoretiske og de praktiske værdier. Specielt ved en udlæsning på omkring 75, hvor der er ca. 10 cm forskel.

Sensoren er desuden afhængig af hvilket materiale den måler afstanden til. På figur 3.6 er vist sammenligning af målinger mod fire forskellige materialer. Igen ses det, at omkring en udlæsning på 75 kan den faktiske afstand svinge med op til 10 cm, afhængigt af materialet.



Figur 3.5: Sammenligning af målinger fra sensor og ligning 3.1. Figur lånt fra [2]



Figur 3.6: Sammenligning af målinger på fire forskellige materialer. Figur lånt fra [2]

3.1.5 Aktorer

Robottens aktorer er de to motorer. Hvis de to motorer kører lige hurtigt samme vej (og robottens hjul er præcis lige store, og gearingen er ens) vil robotten køre lige ud - frem eller tilbage. Kører de to motorer ikke lige hurtigt vil robotten køre i en cirkelformet kurve, og hvis motorerne kører hver sin vej, vil robotten dreje om et punkt mellem de to styrende hjul.

Kommandoen til en motor er en værdi mellem -128 og 127, som angiver hjulets fart - fortegnet angiver rotationsretningen. Der er i robotten en regulerings mekanisme som regulerer farten på hjulet, således at farten på det enkelte hjul er tilnærmelsesvis konstant. Denne mekanisme fungerer dog ikke således at de to hjul kører lige hurtigt. Det vil sige, at hvis de har fået samme kommando, kører de ikke nødvendigvis præcis lige hurtigt.

3.2 Software modul til SMR

Data fra afstandsmåler i cm

Regulering/styring af motorer, moveto(x,y,alpha) funktion

Positionsbestemmelse [1] (Terrestrisk Navigation)

Det ovenstående Interface til smr-robotten er ikke velegnet til en agentarkitektur. Derfor bygges et "modul" oven på smr-interfacet, som tilbyder et nyt interface på et højere niveau. Om sådan et modul er det nederste lag i en agentarkitektur, eller om det er det øverste lag af robotten kan diskuteres. Men her betragtes det som en del af robotten.

Modulet skal indeholde tre ting:

- Løbende positionsbestemmelse for robotten.
- Støjreduktion og lignær udlæsning af afstandsmålere og observationer omregnet til punkter.
- Funktion som kan udføre "bevægelseskommandoer"som `MoveTo( x, y )` og `TurnTo(` α `)`

Algoritmer for disse tre funktionaliteter er beskrevet i de følgende afsnit.

3.2.1 Positionsbestemmelse

Det er vigtigt for en robot af denne type at kende sin position. Den eneste måde smr-roboten kan bestemme sin position er ved *Terrestrisk Navigation*[1] Terrestrisk positionsbestemmelse kan udføres med odometriberegninger, som beskrevet i [1]. (I [5] beskrives positionsbestemmelse ud fra observationer. Dette er dog ikke praktisk muligt på smr-robotterne på grund af de forholdsvis ringe sensorer).

Odometriske beregninger bygger på, at man kender sin startposition og diskretiserer robottens bevægelse. Den nuværende position beregnes ud fra den foregående samt viden om, hvor meget hjulene har bevæget sig. Beregningernes præcision afhænger meget af, hvor ofte denne beregning foretages. På smr-robotterne kan hjul-sensorerne aflæses hvert 10. millisekund, hvilket gerne skulle overholdes for at få en så præcis positionsberegning som muligt.

Denne form for positionsbestemmelse har sine begrænsninger. For det første akkumuleres unøjagtigheden af beregningen, og for det andet må robottens hjul ikke skride i forhold til underlaget.

Da robotten kun kan bevæge sig på et fladt plan angives positionen som (x, y, θ), hvor (x, y) angiver positionen af robottens centrum (hvor centrum er defineret som punktet midt mellem de to styrende hjul), og θ angiver robottens orientering. Vinklen θ defineres på samme måde vinkler på "enhedscirklen"fra trigometri. Det vil sige $\theta = 0$ når robotten kører parallelt med koordinatsystemets x-akse og $\theta = 90$ når robotten kører parallelt med y-aksen. θ øges således, når robotten drejer mod venstre.

Odometriske beregninger

Forholdet mellem antallet af "skridt" målt på hjul-sensoren og hvor langt hjulet har kørt er:

$$c_m = \frac{\pi D_n}{C_e} \tag{3.2}$$

hvor D_n er hjulets diameter og C_e er opløsningen på hjul-sensoren.

[1] Oversat fra det engelske udtryk Dead-reckoning : "The calculation of a ship's position from the log and compas, when observations cannot be taken."

Hvis robottens forrige position er (x_i, y_i, θ_i) og højre og venstre hjul-sensorer har målt henholdsvis N_H og N_V skridt siden denne position, da har højre og venstre hjul bevæget sig:

$$\Delta U_{H,i} = c_m N_{H,i} \tag{3.3}$$
$$\Delta U_{V,i} = c_m N_{V,i} \tag{3.4}$$

Robottens centrum har da flyttet sig:

$$\Delta U_i = \frac{(U_{H,i} + U_{V,i})}{2} \tag{3.5}$$

Ændringen i robottens orientering er:

$$\Delta\theta_i = \frac{U_{H,i} - U_{V,i}}{b} \tag{3.6}$$

hvor b er afstanden mellem de to styrende hjul. Både 3.5 og 3.6 antager at $U_{H,i}$ og $U_{V,i}$ er "små". Robottens orientering bliver nu:

$$\theta_{i+1} = \theta_i + \Delta\theta_i \tag{3.7}$$

og robottens nye centrum bliver:

$$x_{i+1} = x_i + \Delta U_i \cos\theta_{i+1} \tag{3.8}$$
$$y_{i+1} = y_i + \Delta U_i \sin\theta_{i+1} \tag{3.9}$$

I det ovenstående er robottens centrum beregnet som positionen midt i mellem robottens styrende hjul. Robottens virkelige centrum beregnes som:

$$x_c = x + d\cos\theta \tag{3.10}$$
$$y_c = y + d\sin\theta \tag{3.11}$$

hvor (x, y) er punktet mellem hjulene, (x_c, y_c) er robottens centrum og d er afstanden fra bagakslen til robottens centrum.

Fejl ved de odometriske beregningerne

I [1] inddeles fejlene ved odometri i to kategorier: systematiske og ikke-systematiske. Ikke-systematiske fejl, er fejl der opstår når et eller flere af de styrende hjul glider i forhold til underlaget. Hvilket kan forekomme af flere grunde f.eks. for kraftig (de)acceleration, påkørsel af objekt, ujævnt underlag eller påvirkning af ydre kræfter. Systematiske fejl, er fejl som skyldes upræcise "robot-data" som afstanden mellem de styrende hjul b og størrelsen på hjulene D. Begrænset opløsning på hjul-sensorerne og begrænset *sampling rate* er også systematiske fejl.

Robotten har ikke nogen mulighed for at opdage ikke-systematiske fejl. Derfor er det væsentligt at denne type fejl ikke forekommer. Begrænsninger på hjul-sensorerne og *sampling rate* kan ikke ændres uden at ændre på konstruktionen af robotten. Fejl der skyldes upræcise værdier af D og b kan rettes. Det gøres ved at indføre tre fejlparametre, som så kan kalibreres:

Hvis de to hjul ikke er præcis lige store vil robotten køre i en bue, mens odometrien beregner, at robotten kører ligeud. Derfor defineres en fejlparameter E_{d1} som forholdet mellem de to hjuls størrelser:

$$E_{d1} = \frac{D_H}{D_V} \tag{3.12}$$

Hvis afstanden mellem de styrende hjul, b ikke er præcis bliver robottens orientering beregnet forkert:

$$E_b = \frac{b_{faktisk}}{b_{nominelt}} \tag{3.13}$$

Endelig hvis den gennemsnitlige hjulstørrelse er forskellig, vil robotten køre længere eller kortere end beregnet:

$$E_{d2} = \frac{D_{faktisk}}{D_{nominelt}} \tag{3.14}$$

I [1] foreslås en systematisk måde til at kalibrere E_{d1} og E_b. Metoden går ud på at lade robotten køre i en firkant, først den ene vej og så den anden vej. Ud fra forskellen mellem start- og slutpositionen for robotten kan så beregnes en bedre værdi for fejlparametrene. Ved gentagne forsøg bliver

fejlparametrene mere og mere præcise. Forsøg med smr-robotten har dog vist, at denne metode er for tung og besværlig i forhold til den præcision, der kan opnås. Derfor benyttes en mere ad-hoc metode til kalibreringen.

3.2.2 Afstandsmålerne og observationer

For at robottens afstandsmålere skal kunne bruges skal de kalibreres, og udlæsningen skal omregnes til en lignær skala.

Kalibrering

Udlæsningen fra afstandsmålerne omregnes til lignær skale (i cm) ved at omskrive ligning 3.1 til:

$$D(output) = K_2 \;/\; \tan\left(\frac{output - K_3}{K_1}\right), \quad \text{for} \quad output > K_3 \tag{3.15}$$

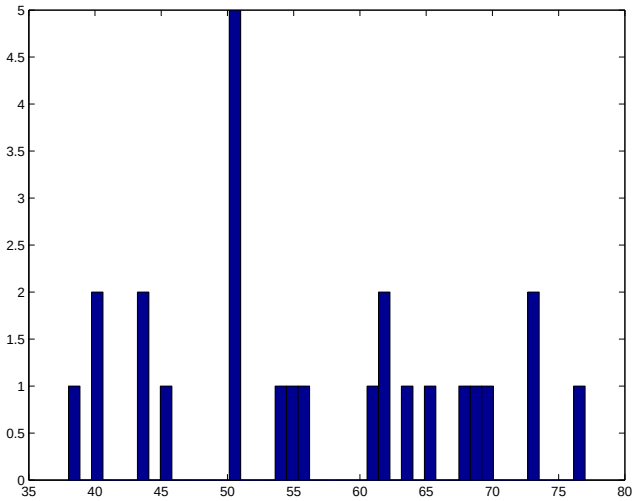
For konstanterne K_1 og K_2 bruges værdierne foreslået i [2]. Til at bestemme K_3 foretages n målinger mod et kalibreringsobjekt, med forskellig afstand til robotten. Hvis den i'te måling er lavet med afstanden d_i til kalibreringsobjektet, og resultatet af 3.15 er $D(\text{output})_i$, vælges K_3 således at

$$\sum_{i=1}^{n} (D(output)_i - d_i)^2 \tag{3.16}$$

minimeres. Typisk kunne kalibreringen fortages med $n = 3$ ved afstandene 10, 30 og 50 cm til et objekt. Hvilket vil fordele usikkerheden nogenlunde over hele afstandsmålerens interval. Men man kan også vælge at foretage kalibreringen med nogle afstande som ligger tæt. F.eks 38, 40 og 42 cm, hvilket vil gøre, at kalibreringen bliver mest præcis ved afstande omkring 40 cm, og tilgengæld mere upræcis i resten af intervallet.

Støj på afstandsmålerne

En støjfri afstandsmåler vil måle den samme afstand til et objekt, ved gentagne målinger, såfremt afstanden til objektet ikke ændres. Men i praktis



Figur 3.7: Fordeling af målingerne fra en afstandsmåler på en smr-robot med mere end tre meter til et objekt.

er dette ikke tilfældet, og variationen af målingerne er faktisk rimelig stor. Dette kan dog løses ved at tage gennemsnittet af flere målinger. Test har vist, at ved at bruge 100 målinger bliver resultatet rimelig stabilt, så længe der er et objekt indefor afstandsmålerens rækkevide. 100 målinger tager godt 1 sek. på smr-robotterne.

Det egentlige problem ved denne støj er, når der ikke er et objekt indefor afstandsmålerens rækkevidde (angivet til ca. 80 cm). Når dette er tilfældet burde udlæsningen af afstandsmåleren være ca. det samme som K_3. (Se ligning 3.15). På figur 3.7 er vist fordelingen af 25 målinger[2] fra en afstandsmåler fra en smr-robot, hvortil der er fundet at $K_3 = 39$ og hvor nærmeste objekt er mere end 3 meter væk. Som det ses er de fleste målinger noget over K_3, og de højeste af disse målinger svarer til en afstand på omkring 50 cm.

For at undgå disse fantom-observationer kan der indlægges et filter, som ignorer udlæsninger under en vis grænse. En høj grænse giver flere fantom-observationer, mens en lav grænse forkorter målerens rækkevidde. Hvis der for måleren, der er benyttet til målingerne på figur 3.7, lægges en grænse

[2] Hvor hver måling består af gennemsnittet af 100 aflæsninger af afstandsmåleren

på 80, begrænses målerens rækkevidde til ca. 45 cm. Målinger længere væk bliver ignoreret.

3.2.3 Reguleringsalgoritmer til robottens motorer

Til at styre robotten er to funktioner nødvendige:

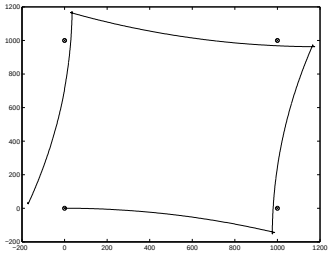
- MoveTo(x, y): Kører robotten i en lige linie fra sin nuværende position til positionen (x, y).
- TurnTo(θ): Drejer robotten så dens orientering bliver θ.

Disse to funktioner er beskrevet herunder. Som beskrevet i afsnit 3.2.1 kan for kraftig acceleration give fejl i robottens positionsberegninger. Det har dog vist sig, at hvis farten på motorerne ikke overstiger 20 (se interface på figur 3.2), er specelt hensyn til accelerationen ikke nødvendig, så længe de odometriske beregninger fortsættes indtil robotten står stille.

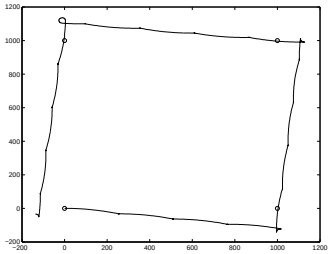
Robotten kan dog køre væsentligt hurtigere. Den maksimale fart på motorerne er 127. Hvis dette ønskes udnyttet kræves nogle mere robuste reguleringsalgoritmer end dem der er beskrevet herunder. En fart på 20 er dog ikke specielt langsomt i forhold til størrelsen af robotternes forsøgsområde.

For at demonstrere de problemer, der er ved at styre robotten, vises her nogle plots over robottens rute, ved forskellige reguleringsalgoritmer. Plottene viser fire punkter i en firkant, som udgør en rute for robotten. Ruten starter i punktet (0,0) og går herfra mod uret igennem punkterne (1000,0), (1000, 1000) og (0, 1000) tilbage til (0.0).

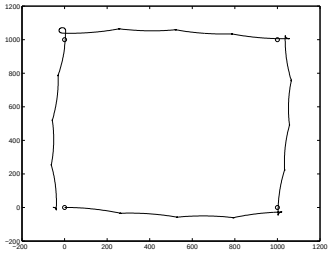
Først hvis det ene hjul har større periferi-fart end det andet, vil robotten køre i en bue. En tur rundt i en firkant kommer til at se ud som på figur 3.8, hvis venstre hjul kører 10% hurtigere end højre. Dette kunne løses ved at rette kursen med "TurnTo" kommandoen med faste intervaller. På figur 3.9 er dette vist hvor kursen rettes til begyndelskursen for hver 25 cm. Dette kan gøres bedre ved at beregne en ny kurs ved hver justering i stedet for blot at justere til begyndelskursen. Dette ses på figur 3.10. Endelig hvis "TurnTo" funktionen laver for kraftig overstyring, bliver resultatet som på figur 3.11.



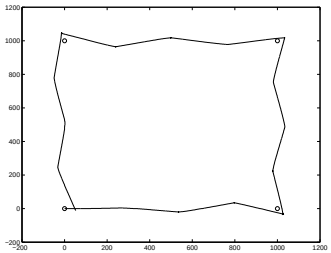
Figur 3.8: Plot af smr-robots rute. Venstre hjul kører 10% hurtigere end højre. Den lige strækning køres uden korrektion af kurs. Sidelængde 1000mm



Figur 3.9: Plot af smr-robots rute. Venstre hjul kører 30% hurtigere end højre. De lige strækninger korrigeres til begyndelses kursen for hver 25cm. Sidelængde 1000mm



Figur 3.10: Plot af smr-robots rute. Venstre hjul kører 30% hurtigere end højre. De lige strækninger korrigeres til en (ny) kurs mod målet for hver 25cm. Sidelængde 1000mm



Figur 3.11: Plot af smr-robots rute, hvor "TurnTo" funktionen overstyrer kraftigt. De lige strækninger korrigeres til kurs mod målet for hver 25cm. Sidelængde 1.000mm

TurnTo funktionen

Robottens nuværende orientering kaldes θ_1 og den ønskede orientering kaldes θ_2. Afhængig af θ_1 og θ_2 skal robotten enten dreje højre eller venstre om, for at dreje den korteste vej. Eksempelvis hvis $\theta_1 = 10°$ og $\theta_2 = 0°$, kan enten drejes 10° mod højre, eller 350° mod venstre. Hvor den første mulighed naturligvis skal vælges.

Hvis forskellen mellem de to orienteringer, $\Delta V = \theta_2 - \theta_1$ normeres til intervallet [0;360[, er den korteste rotation mod højre hvis $\Delta V > 180$, og ellers mod venstre.

Funktionen `Update(speed_left, speed_right)` bruges til at kommunikere med robotten igennem interfacet vist på figur 3.2. Funktionen sætter farten på henholdsvis venstre og højre hjul til `speed_left` og `speed_right`, og beregner robottens nye position.

For at robotten skal dreje til højre skal venstre hjul køre hurtigere end højre, derfor vil update(SPEED, -SPEED), sætte robotten til at rotere omkring sig selv mod højre med farten `SPEED`. `Update(0, 0)` stopper robottens bevægelse.

Algoritmen, der roterer robotten højre om mod orienteringen `v`, kan nu beskrives i sproget C:

```
void TurnTo(float v)
  {
     float dV = v - theta;
     while (dV <= 0.0)
       {
         theta = update(SPEED, -SPEED);
         dV = v - odo->theta;
       }
     update(0, 0);
  }
```

Hvor `theta` er robottens aktuelle orientering, som beregnes af `update` funktionen. Funktionen antager, at alle vinkler (v, theta og dV) er normaliseret til intervallet [0;360[. Det vil sige $v = v \ mod \ 360$.

Den fulde funktion, som normaliserer vinklerne og bestemmer om der skal drejes til højre eller venstre, kan ses i bilag C.

Problemet med denne algoritme er, at selv om motorerne stoppes, er robotten i bevægelse, og der vil gå lidt tid før den stopper. Det har dog vist sig i praksis at funktionen kan positionere robotten med få graders præcision, når blot `SPEED` ikke er for høj. Forsøg har vist at en passende værdi for SPEED er mellem 5 og 20, hvilket svarer nogenlunde til en 180 graders rotation på ca. 5 sekunder.

Hvis man ønsker en mere præcis algoritme kan man lade `SPEED` være en funktion af `dV`. F.eks. ved at halvere `SPEED`, når `dV` er under 10 grader.

MoveTo funktionen

Første skridt mod en "MoveTo(x, y)"funktion er en "move(t)"funktion, som kører t meter lige ud. Dette kunne umiddelbart gøres med et kald til funktionen `update(SPEED, SPEED)`. Men dette er ikke nok, da det kun får robotten til at køre tilnærmelsesvis lige ud. Problemet er, at de to motorer (til henholdsvis højre og venstre hjul) ikke kører præcis lige hurtigt selvom de sættes til samme hastighed. Desuden kan selv en lille forskel i størrelsen af robottens hjul betyde, at robotten ikke kører lige. Det er altså ikke nok at sikre, at hjulenes centrum kører lige hurtigt, de skal køre lige hurtigt ved hjulenes periferi. Under odometriberegningerne beregnes hvor langt henholdsvis højre og venstre hjul har kørt (3.3 og 3.4). Dette kan bruges til at sikre, at de kører lige langt.

Igen bruges funktionen `update(speed_left, speed_right)` til at angive hastigheden på hjulene og til at udføre de odometriske beregninger. Man kunne vælge at reducere hastigheden på det hjul, som har kørt for langt. Men i stedet er valgt at holde farten på venstre hjul konstant, og så regulere farten på højre hjul. Højre hjuls fart bestemmes som farten på venstre hjul plus et reguleringsbidrag. Reguleringsbidraget er proportionalt med forskellen mellem hvor langt de to hjul har kørt:

$$speed_{right} = speed_{left} + speed_{left}(\Delta_{left} - \Delta_{right})K \tag{3.17}$$

K er en reguleringskonstant, som afgør hvor "følsom"reguleringen er. En værdi på $K = 10$ har vist sig passende. Δ_{left} og Δ_{right} angiver hvor langt de to hjul har kørt.


```
void move(double distance, int speed)
 {
    double olddist = odo->dist;
    double start_dist_L = odo->dist_L;
    double start_dist_R = odo->dist_R;

    double dl, dr;
    double speedR, K = 10;

    while((odo->dist - olddist) <= fabs(distance))
      {
        dl = odo->dist_L - start_dist_L;
        dr = odo->dist_R - start_dist_R;
        speedR = speed + speed*(dl-dr)*K;
        if (speedR > 2*speed) speedR = 2*speed;
        if (speedR < 0.5*speed) speedR = 0.5*speed;
        update( speed, speedR );
      }
  }
```

Figur 3.12: "move" funktionen

Som sikkerhed for at algoritmen ikke "over reagerer" begrænses $speed_{right}$ til:

$$2speed_{left} \geq speed_{right} \geq 0.5speed_{left}$$

Med "move (t)" og "TurnTo(θ)" funktionerne er MovtTo(x, y) funktionen rimelig simpel. Først drejes mod punktet, og så køres afstanden til punktet.

Robottens udgangsposition betegnes (x_1, y_1, θ_1), og den ønskede position betegnes (x_2, y_2). Robottens slutorientering θ_2 er ikke angivet, men vil ca. være retningen mellem de to punkter.

Retningen, der skal køres i, beregnes som:

$$\theta = \arctan\left(\frac{\Delta y}{\Delta x}\right) \quad \text{for} \quad \Delta x \neq 0$$

hvor $\Delta x = x_2 - x_1$ og $\Delta y = y_2 - y_1$. Hvis $\Delta x = 0$ er θ 90 eller 270, afhængigt af fortegnet på Δy.

Distancen der skal køres bestemmes af:

$$distance = \sqrt{\Delta x^2 + \Delta y^2}$$

Istedet for at korrigere kursen ved et fast interval, vælges løbende at checke hvor meget kursen afviger fra begyndelskursen. Bliver afvigelsen for stor, stoppes robotten, og kursen justeres med TurnTo-funktionen. Afvigelsen skal være større end præcisionen på TurnTo-funktionen, for at en fornuftig justering af kursen er mulig.

Kapitel 4

Analyse og diskussion af scenarier for SMR

I dette afsnit ses på muligheder og begrænsninger af forsøg med smr robotterne. Forsøgene er begrænsede dels af robotternes konstruktion af mekanik og elektronik og dels af de fysiske rammer og udstyr i IAU's smr lokale.

Først gennemgås hvilke muligheder der er med smr-robotterne, og herefter opstilles nogle scenarier, som kunne tænkes løst med en agent-arkitektur.

4.1 Muligheder med smr platformen

For at få indblik i hvad der kan lade sig gøre med robotterne, og hvilke problemer der opstår, er der lavet en række forsøg med robotterne. Forsøgene er udført med det software modul, som er beskrevet i 3.2. Dette er derfor ikke forsøg med *agenter*, men med den robot som agent-arkitekturen skal designes til.

Forsøgene i sig selv er ikke videre interssante, derfor beskrives her kun de konklusioner førsøgene har medført:

Forsøgsområdet: Til forsøg kan kun benyttes forsøgsområdet i IAU's smr-lokale, da robotterne er afhængige af det trådløse netværk, som kun findes i dette lokale. Forsøgsområdet er ca. seks gange syv meter. Der er mulighed

for at opstille bander langs en del af områdets kant, men ikke hele vejen rundt.

Antal robotter: Der er i alt bygget 8 robotter. To til tre er dog den praktisk grænse for hvor mange robotter der er til rådighed til projektet. Desuden er forsøgsområdets størrelse ikke egnet til mere end tre robotter (da de ellers vil køre meget tæt).

Kommunikation: Alle robotterne er forbundet med trådløst netværk, der er derfor gode mulighed for at lave kommunikation mellem robotterne.

Positionsbestemmelse: Robotterne er ikke udstyret med GPS eller ligende positionsbestemmelsesudstyr. Positionsbestemmelse ud fra observationer er ikke praktisk muligt med robottens begrænsede sensorer. Den eneste mulighed for positionsbestemmelse er derfor Terrestrisk navigation, som beskrevet i afsnit 3.2.1. De odometriske beregninger giver dog en akkumuleret unøjagtighed.

Opdatering af position: Efter som robotternes positionsbestemmelse akkumulerer fejl, ville det være interessant med en mulighed for at opdatere robotternes position. Det ville være specielt interessant, hvis robotterne kunne bestemme deres position i forhold til hinanden f.eks. ved brug af deres afstandsmålere. Dette har dog vist sig at være meget svært i praksis, dels på grund af unøjagtighed på afstandsmålerne og dels på grund af robotternes ujævne overflade.

En anden mulighed er at robotterne opdaterer deres position ved banderne langs forsøgsområdets kant. Robottens position (x,y) kan bestemmes med rimelig nøjagtghed (omkring 3-4 cm), hvis afstandsmålerne er kalibreret meget præcist. Men robottens orientering (θ) kan kun bestemmes med en nøjagtighed på ca. $\pm 15°$. Det betyder, at positionen kan forbedres, hvis den akkumulerede fejl er blevet stor, men den kan ikke opdateres helt præcist.

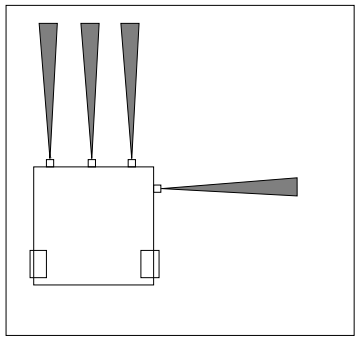
En sidste mulighed er at benytte robottens liniesensor, til at opsøge "kendte positioner" og herved opdatere sin position. Det er dog valgt ikke at benytte liniesensoren i dette projekt, da det vil kræve en del "robot-teknisk" software udvikling.

Objekter i robottens verden: Alle robottens afstandsmålere sidder i samme højde. Det er derfor nødvendigt at antage, at alle objekter i robottens verden har mindst denne højde, for at robotterne kan "se" dem. Desuden må det af samme grund antages, at objekterne er lige brede i alle højder.

Detektion af andre objekter: Robotterne har mulighed for at detektere andre objekter i verden med deres afstandsmålere. Dette er dog forbundet med nogen problemer. For det første ved robotten ikke, hvad den ser. Den kan blot registrere, at der "er noget", men om det er en bande ved forsøgsområdets kant, en anden robot eller noget tredje kan ikke skelnes. For det andet er der en rimelig stor usikerhed på afstanden til dette objekt, da afstandsmålerne er kalibreret til at måle mod et bestemt materiale. (Se evt. figur 3.6).

Fantom objekter: Selv om der bruges et støjfilter (se afsnit 3.2.2) på robottens sensorer, vil disse stadig kunne observere "fantom objekter". Det skyldes, at der ikke er tid til at foretage gentagende målinger, når robotten kører, og der observeres noget foran robotten. Det er i disse situationer nødvendigt at stoppe robotten, for at kontrollere målingerne.

Manipulering af objekter: Den eneste mulighed robotterne har for at manipulere verden er ved at skubbe ting. Dette kræver dog særlig opmærksomhed, da robotterne ikke må skubbe til hinanden eller til banderne. Desuden vil agentens afstandsmålere blive blokeret, mens et objekt skubbes, så robotten ikke kan se om den skulle påkøre noget. Det er også svært for robotten at skubbe f.eks en papkasse så nøjagtigt, at kassen ikke bliver skubbet til side i stedet for fremad.



Figur 4.1: Omtrentlig skitse over robottens synsfelt.

Kollision: Hvis en robot skubber et objekt er den altså "blind" og kan derfor ikke opdage/forhindre en kollision. To robotter, som skubber hvert sit objekt og som er på kollisionskurs, vil derfor støde sammen. Hvis en robot, som skubber et objekt, kører mod en robot, som ikke skubber noget, vil den ene robot kunne se kollisionsfaren, men først når robotterne har en afstand på mindst 50 cm. Dette vil muligvis give robotten tid nok til at

flytte sig, men hvis robotten har sine afstandsmålere rettet væk fra "faren", vil en kollision igen kunne opstå. Desuden kan robotterne kollidere, hvis de bakker mod hinanden, eller hvis de kører mod hinanden i en spids vinkel, hvor afstandsmålerne ikke kan se den anden. Robottens sensorer alene er altså ikke nok til at undgå kollisioner, heller ikke selv om robotterne ikke skubber på objekter.

Beregningskraft: Robotterne er udstyret med en "standard" PC-processer, så beregninger kan udføres med samme hastighed som på andre PC'ere. Der er dog den begrænsning, at sålænge robotten kører, skal hjul-sensorerne aflæses hvert 10. millisekund, og aflæsningen tager omkring 2-5 millisekunder. Den resterende tid kan bruges til andre beregninger. Da operativsystemet på robotterne ikke er et real–tids system, kan det ikke af operativsystemet garanteres at processen, der har kontakt til robottens sensorer, har adgang til processoren på de rigtige tidspunkter. Dette løses ved at give den kritiske proces højere prioritet i operativsystemets *scheduler* i forhold til øvrige brugerprocesser. Uden at gå i nærmere detaljer omkring prioritering af processer, betyder dette, at mens robotten kører, kan der kun laves "mindre", eller "lettere" beregninger. Mens eventuelle tunge beregninger må udføres, mens robotten står stille.

4.2 Anvendelsesmuligheder

SMR robotterne er udviklede til undervisning og forskning, og det er svært et finde en anden praktisk anvendelse af dem. De kan dog godt bruges til at eksperimentere med nogle løsninger, som kunne anvendes i mere realistiske situationer. For at få en ide om hvilke opgaver der kunne være interessante at løse med autonome robotter beskrives her to anvendelsesmuligheder for autonome robotter:

- **Rengøringsrobotter**: Robotter der kan støvsuge eller vaske gulv.
- **Lagerhalsrobotter**: Robotter som skal flytte rundt på paller med varer i en lagerhal.

4.2.1 Rengøringsrobotter

Man kan forestille sig mange steder, hvor automatisk rengøring ville være nyttigt. Lige fra private hjem til kontorer og fabrikshaller. For at sådan en

robot skal være anvendelig skal den kunne fungere i et dynamisk miljø, hvor mennesker, møbler, maskiner og sågar andre robotter bevæger sig rundt på uforudsigelige måder.

I et privat hjem er én robot formentlig tilstrækkelig, mens flere robotter med fordel kunne deles om arbejdet i f.eks. en stor fabrikshal. Den nemme løsning ville være at give hver robot et fast del af fabrikken at rengøre. En mere interessant og fleksibel løsning ville dog være, hvis robotterne fik en samlet ordre, som de så selv fordelte mellem sig. Dette ville også betyde, at hvis nogle af robotterne er defekte eller til service vil opgaven stadig blive løst. En ordre til sådan et system af robotter kunne være som: "Rengør hal A og B nu", eller det kunne være: "Gulvet skal vaskes med x minutters mellemrum".

Det optimale for et rengøringssystem vil være, at de overhovedet ikke kræver indblanding fra mennesker. Dette vil betyde, at robotterne får en ordre, som de udfører så effektivt som muligt, og selv opdager ændringer i miljøet, og selv finder ud af at køre til en "docking-station", når deres batterier skal lades op, have skiftet vand eller lignende. Det kræver også, at robotterne besidder en stor fleksibilitet, så uforudsete ændringer i miljø eller ordre ikke er en hindring. Der vil dog være situationer, som ikke kan klares. F.eks. hvis en rengøringsrobot bliver lukket inde. Systemet bør også kunne håndtere til- og fragang af robotter, således at hvis f.eks. fabrikken anskaffer flere robotter, skal disse kunne sættes i drift uden at systemet stoppes eller omkonfigureres.

De fleste har erkendt, at det ikke er muligt at lave en robot, som ikke laver fejl. Men hvor hyppigt kan fejl accepteres? Det kommer naturligvis an på konsekvensen af fejlen. For en støvsugerrobot vil fejlen typisk være, at den "sidder fast" i et hjørne og skal hjælpes i gang igen. Dette ville ikke være katastrofalt, men en husmor ville formentlig hurtigt kassere robotten, hvis det sker for tit. En stor tung industristøvsugerrobot kan derimod forårsage langt større skader på både mennesker og materiel. For en sådan robot er det derfor essentielt, at den er konstrueret til ikke at støde ind i ting.

4.2.2 Lagerhalsrobotter.

Med en lagerhal tænkes på en hal til opbevaring af forskellige varer. Typisk ankommer varer i lastbiler med mange paller med samme type vare, og afhentes af lastbiler, som aftager mange forskellige varer. Lagerrobotternes

opgave er så at hente paller i lastbilerne som leverer varer, og anbringe dem passende steder i lagerhallen, og at hente varer fra lagerhallen til de lastbiler, som skal bruge dem.

Som for rengøringsrobotterne kræver dette stor fleksibilitet af robotterne, så de kan løse uforudsete opgaver. Men specielt gælder at robotterne skal dele arbejdet meget præcist mellem sig, således at lastbilerne bliver pakket med præcis de varer som ønskes. Desuden skal robotterne kunne finde frem til de rigtige hylder, undgå at kollidere og undgå at stå i vejen for hinanden.

4.2.3 Kombination af flere typer af robotter

En spændende kombination er en lagerhal med både lagerhalsrobotter og rengøringsrobotter. De to typer af robotter kunne virke uden kendskab til hinandens eksistens, da de skal kunne fungere i et miljø med ukendt dynamik. Men specielt interessant ville det være, hvis de kunne kommunikere og udveksle information som f.eks. intention. De to typer robotter kan ikke direkte hjælpe hinanden, men de kan samarbejde om ikke at være i vejen for hinanden. Hvis for eksempel en rengøringsrobot vil rengøre et område, hvor en lagerhalsrobot vil hente en vare. Hvem skal så have lov til at køre i området først? Lagerhalsrobotten er sikkert under størst tidspres hvis en lastbil venter på den, mens rengøringsrobotten evt. i mellemtiden kan rengøre et andet område. Derfor kan robotterne blive enige om at lagerhalsrobotten får adgang først. Men hvis dette bliver gentaget mange gange i samme område, vil området blive mere og mere beskidt, og på et tidspunkt må robotterne indse at rengøringsrobotten bliver nødt til at komme til. Løsningen på dette må indebære, at robotterne har en eller anden form for prioritering af deres mål/opgaver. For eksempel kan rengørings robottens prioritering af at rengøre et område være en funktion af tiden siden sidst området er rengjort.

4.3 Antagelser for scenarierne

I det følgende afsnit beskrives nogle scenarier for smr-robotterne. Scenarierne er beskrevet i en rækkefølge med stigende kompleksitet, og sigter mod et scenarie som simulerer en lagerhal med rengørings- og lagerhalrobotter. Men først beskrives de antagelser, som er nødvendige for at eksperimentere med robotterne. Antagelserne stammer fra beskrivelsen i afsnit 4.1.

Agentens verden er et plant firkantet område. Områdets størrelse og robottens startposition er kendt af agenten.

I agentens verden findes kun to typer af objekter: andre agenter og kasser. En kasse kan både simulere en lagerhals-palle, som skal skubbes rundt og en forhindring, som robotten skal køre uden om.

4.4 Scenario 1: Én agent i en "statisk" verden.

Det første scenario undersøger, hvad der kræves for at robotten kan køre rundt i den begrænsede verden.

- **Verden**: En agent, ingen kasser.
- **Opgave**: Udvælg et tilfældigt punkt (x,y) og kør til dette. Udvælg så et nyt punkt og kør til dette osv.
- **Krav til robot:** Robotten (hardware/mekanik) fungerer, og kan kontrolleres.

Som tidligere beskrevet er en hybridarkitektur bestående af en deliberativ del og en reaktive del formodentligt et godt valg. Dette scenario kræver ikke de store "overvejelser", men scenariet løses alligevel med denne hybrid model for at kunne sammenligne med senere scenarier/løsninger. Den deliberative dels opgave er her at udvælge et tilfældigt punkt og så sende en "moveto(x,y)" kommando til den reaktive del.

4.4.1 Krav til den reaktive del

Den reaktive del skal kunne udføre en "moveto" kommando og samtidig løbende beregne robottens position. (Det er oplagt at placere odometriberegningerne i den reaktive del, da det kræver tæt kontakt til robottens sensorer.) Algoritmer til odometriberegninger og til en movoto-funktion er beskrevet i kapitel 3.

4.4.2 Krav til den deliberative del

Den deliberative del skal kunne udvælge et punkt inden for den afgrænsede verden. Så længe agenten ved hvilke punkter, der tilhører den afgrænsede verden, skulle dette være simpelt nok.

4.4.3 Hvad kan gå galt?

Robotten kan køre uden for sit afgrænsede område. Hvis den reaktive del altid styrer robotten i en lige linie mellem punkterne, og punkterne altid vælges inde for området, kan dette kun opstå, hvis agentens odometriske beregninger er blevet så upræcise, at agenten tror den kører i det afgrænsede område, mens den faktisk er udenfor. Det betyder, at når robotten ikke kan opdatere sin position, kan den kun køre et begrænset stykke tid før positions-bestemmelsen er så upræcis, at robotten kører uden for området.

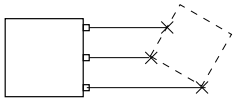
4.5 Scenario 2: Én agent i "statisk" verden med ukendte forhindringer

Scenario 1 udvides nu med nogle kasser. Kasserne simulerer forhindringer, som agenten skal køre uden om. Agenten kender hverken antallet af kasser eller deres placeringen. Dette scenario undersøger agenternes mulighed for at opbygge et "verdensbillede" og planlægge en rute ud fra dette.

Dette stiller noget større krav til agenten:

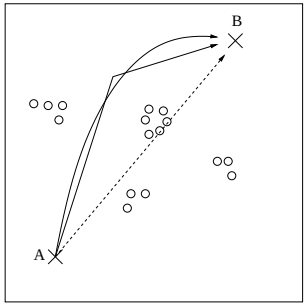
- Observationer fra afstandsmålerne skal gemmes (eller huskes) på en passende måde.
- Observationer fra afstandsmålerne skal omregnes til placering af kassen.
- Robotten kan ikke "bare" køre i en lige linie mod det ønskede punkt, men må lægge en rute uden om kendte forhindringer.
- En planlagt rute kan være blokeret af en kasse, som ikke tidligere er observeret, og en rute skal derfor kunne ændres undervejs.

Agenten behøver en form for kort, hvorpå observationer kan "indtegnes". Desuden behøver den en funktion, som kan finde en rute fra robottens nuværende position til den ønskede position. Endelig er der behov for en funktion, som kan sammenkæde observationer fra afstandsmålerne med kasser og deres placering. Bemærk at en observation fra afstandsmålerne blot er et punkt (x,y). Når "noget" observeres, skal der afgøres om observationen tilhører en allerede kendt kasse, eller om der er tale om en ny kasse. Desuden skal en mængde af observationer kunne omregnes til en kasse med et gæt på dens placering. Se skitse på figur 4.2



Figur 4.2: Skitse af robot der ud fra tre punkter gætter på placeringen af en kasse.

Disse "nye" funktioner kunne i princippet tilhøre både den reaktive og den deliberative del af agenten. Men i første omgang vælges at placere disse funktioner i et selvstændigt modul. Dette modul kaldes for *Map*. Map-modulet skal altså modtage information om observationer fra den reaktive del på formen *observation(x,y)*. Desuden skal den deliberative del kunne hente en rute mellem to punkter. Ruteplanlægningen kan være en graf søgealgoritme, simuleret udglødning eller andet. På figur 4.3 er vist et eksempel på et verdenskort, som det kunne se ud for agenten. Agenten ønsker at køre fra A til B. Den direkte vej er blokeret så en alternativ rute må findes. På figuren er vist to mulige. En af rette linier og en med "bløde" kurver. Sidstnævnte vil kræve en mere avanceret styring af robotten end de algoritmer beskrevet i 3.2.



Figur 4.3: Skitse ef et verdensbillede som agenten kunne have opbygget det. Observationer er markeret med en cirkel. Agenten ønsker at køre fra punkt A til B.

Efter som den reaktive del kører i lige linier mellem punkter må en rute være en liste af punkter, som der kan køres direkte i mellem, og hvor det første og det sidste punkt er henholdsvis rutens start- og slutpunkt.

Hvis den reaktive del må stoppe pga. en blokeret rute, er der to muligheder for at forsætte. Den reaktive del kan selv forsøge at køre uden om for eksempel ved anvendelse af en myrealgoritme, hvor agenten ved hjælp af sine sensorer følger kanten af forhindringerne, indtil der er frit mod målet. Denne algoritme er beskrevet i [5]. Eller den deliberative del kan hente en ny rute i Map-modulet, som tager hensyn til de nye observationer.

Dette stiller nye krav til både den deliberative og til den reaktive del af agenten.

4.5.1 Yderligere krav til den deliberative del

Den deliberative del udvælger stadig et tilfældigt punkt, men istedet for at sende en moveto kommando til den reaktive del, hentes en rute i Map-modulet fra robottens aktuelle position til det udvalgte punkt. Ruten sendes nu stykke for stykke til den reaktive del.

4.5.2 Yderligere krav til den reaktive del.

Den reaktive del skal stadig kunne udføre kommandoen "moveto(x,y)". Men skal samtidig, med passende små intervaller, checke sine afstandsmålere. Hvis noget observeres omregnes dette til et observations punkt, som sendes til den deliberative del. Hvis det observerede blokerer robottens vej, skal den naturligvis stoppe, og den deliberative del må finde en ny rute.

4.5.3 Hvad kan gå galt?

Igen er robottens upræcise positions bestemmelse et problem. De "objekter" robotten observerer kan naturligvis ikke stedbestemmes bedre end robottens egen position. Usikkerheden på disse objekter er en sum af usikkerheden på robottens position og usikkerheden på afstandsmålerne. Dette betyder, at når en rute planlægges, skal ruten gå i passende stor afstand til observerede objekter.

4.6 Scenario 3: Flere agenter i en "statisk" verden.

Scenario 1 udvides nu til at indeholde flere agenter. Det vil sige verden indeholder to eller flere agenter, som kører rundt imellem hinanden. Der er ingen andre forhindringer eller objekter.

Den største udfordring ved denne udvidelse er naturligvis at undgå kollision mellem robotterne. Som beskrevet i afsnit 4.1 kan robotterne ofte "se" hinanden, sålænge de kører forlæns og ikke skubber på en kasse, som blokerer deres udsyn. Men der eksisterer altså også situationer, hvor afstandsmålerne ikke er tilstrækkelige.

En mulig løsning er at agenterne reserverer et område, hvor de øvrige agenter, så accepterer ikke at køre. Dette vil dog udelukke samarbejdsopgaver, som kræver at agenterne er tæt på hinanden. En anden mulighed er at agenterne kører "almindeligt" så længe der ikke er nogen anden agent i nærheden og kører med "særlig opmærksomhed", når en anden agent er tæt på. "Særlig opmærksomhed" kunne være, at agenten kører meget langsomt og ofte drejer fra side til side for at få så meget information som muligt. Eller det kunne betyde, at alle agenter i området står stille, mens en af gangen udfører sin opgave eller forlader det "kritiske område".

Hvor tæt robotterne kan køre på hinanden afgøres af robotternes positionssikkerhed. Hvis to robotter begge forventer at have en positions unøjagtighed på op til 20 cm, skal afstanden mellem dem være mindst 2×20 cm plus en passende margen.

Den første løsning vil betyde, at agenterne ofte skal forhandle om "rettighederne" til forskellige områder. Mens den anden løsning vil betyde, at alle agenterne hele tiden skal vide, eller have et godt gæt på, hvor de andre er.

Der findes flere beskrevne løsninger til hvordan agenter kan forhandle om ressourcer som refereret til i afsnit 2.2. Omvendt så er det rimelig simpelt at få alle til at vide, hvor de andre er. Hvis hver agent med jævne mellemrum, udsender et signal indeholdende sin egen formodede position, og disse signaler opfanges af det før omtalte Map-modul, så vil informationen være til rådighed for alle agenterne.

Dette vil give en del kommunikation, specielt hvis man forestiller sig et scenario med mange agenter. F.eks. kunne man godt forestille sig en stor lagerhal med 100 robotter. Men dette skal sammenlignes med hvor meget kommunikation en forhandling vil give. Hvis robotterne kører meget tæt (mange robotter på lille areal) vil agenterne ofte skulle forhandle, og hvis der er mere end to agenter kan de komme ud for ikke bare 2-parts forhandlinger, men n-parts forhandlinger. Hvilket er noget mere kompliceret.

Agenter har kun brug for information om agenter lige i nærheden af sig selv. Man kunne forestille sig robotter med mere avancerede sensorer, som kunne scanne for andre robotter inden for f.eks. en meter. Dette kunne simuleres

ved at en agent via en kommunikationsforbindelse kunne forespørge, om der er nogen der befinder sig i et bestem område omkring robotten. Det ville dog i såfald være nødvendigt for agenten meget ofte at stille denne forespørgsel, og derfor kan positions-informationen lige så godt sendes direkte.

4.7 Scenario 4: Rengøringsrobotter

Smr-robotterne kan bruges til at simulere en rengørings robot, ved blot at antage at et område er rengjort, når robotten har kørt over det.

Map-modulet fra de tidligere scenarier bruges til holde styr på hvilke områder, der er rengjort. For at kunne gøre det inddeles agentens "kort" i felter med en størrelse svarende til robotten. Hvor der så holdes styr på, hvilke felter der er vasket.

Hvis agenten ønsker at vaske et bestemt område skal bestemmes en rute, som dækker alle felter i området, som ikke enten er rengjort eller er blokeret af en forhindring. Denne funktion lægges i map-modulet lige som funktionen, der finder en rute mellem to punkter. For at dække et område behøves ikke at beregne en optimal rute, men den bør dog optimeres i en vis grad.

Hvis der er flere agenter, kan de undgå kollision som beskrevet i de tidligere scenarier. Hvis agenterne har hver deres område, er yderligere interaktion unødvendig. Men ellers skal agenterne deles om opgaverne.

Som udgangspunkt antages at agenternes kommando er at rengøre hele "fabrikshallen" (hele forsøgsområdet). Arbejdet kan nu deles på flere måder. Hver agent kan lave en plan over hvilke områder den enkelte agent skal rengøre, og herefter kan agenterne forhandle om, hvilken plan der skal bruges.

Alle agenterne sender med faste intervaller sin position til de andre agenter. Dette betyder, at hver agent kan holde styr på hvilke områder der er rengjort – ikke bare af den selv, men af alle robotterne. Det er altså ikke strengt nødvendigt at opdelingen af opgaven er helt præcis. Men for at udnytte at der er flere agenter er nogen koordinering naturligvis nødvendig. En løsning til dette er beskrevet i kapitel 5.

4.8 Scenario 5: Lagerhalsrobotter

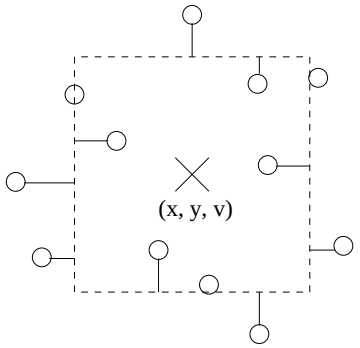
I dette scenario ses på en simulering af lagerhalsrobotter. Kasser bruges til at simulere paller med varer. Opgaverne kan være af typen "Flyt kasse A til position X", og typisk ville der være en række af opgaver. F.eks kunne der være 10 kasser, som alle skulle flyttes til en bestemt position.

Modsat det forrige scenarie er det her strengt nødvendigt at opgaverne bliver delt præcist mellem agenterne, så to agenter ikke vil flytte på samme kasse.

En mulighed er et "først-til-mølle" princip, hvor en agent vælger en af opgaverne, og fortæller de andre hvilken opgave den starter op. De andre skal så naturligvis vælge en anden opgave. Problemet med dette er dog, at eftersom agenter benytter samme algoritmer, og sandsynligvis har samme verdensbillede, vil de vælge den samme opgave samtidig. Dette kunne løses på den måde, at hvis to eller flere vil have den samme opgave, kan hver agent vente et tilfældigt stykke tid og så prøve igen. (Denne teknik anvendes f.eks i mange netværkskommunikationssystemer).

Systemet kunne også indrettes således, at nye opgaver ikke sendes ud til alle agenter men kun til en agent (F.eks tilfældig valgt). Så kunne agenterne udveksle opgaver ved at en agent enten kan tilbyde at modtage en opgave, eller en agent kan forsøge at sende en opgave til en anden. Denne forhandling kan laves forholdsvis simpelt, ved at en agent fortæller alle andre, når den ikke har flere opgaver. Når en agent, der har flere opgaver, "hører" dette, kan den sende opgaver til den ledige. Det betyder, at en robot der melder sig "ledig" sandsynligvis modtager flere opgaver, men dette er ikke noget problem, da agenten blot starter på en af dem og "gemmer" de andre.

Dette scenario kræver desuden en mekanisme, der nogenlunde sikkert kan skubbe kasser fra et punkt til at andet. Hvilket ikke er nogen triviel opgave. Der er to større problemer. Det første er, at agenten ikke kan bestemme kassens position og orientering præcist. Det betyder, at en agent nemt kommer til at skubbe en kasse i en skæv vinkel, hvilket betyder, at kassen ikke skubbes i samme retning som agenten kører. Det andet problem er, at agenten skal have plads til at køre hele vejen rundt om kassen for at kunne skubbe den i alle retninger. Hvis f.eks en kasse bliver skubbet ud mod en af banderne, der omgiver agentens verden, kan robotten ikke hive kassen væk igen. (Dette problem ville givetvis ikke eksistere for en virkelig lagerhalsrobot, da den formentlig er udstyret med en gaffeltruck lignende ordning, som holder kasserne fast på robbotten.)



Figur 4.4: De 12 cirkler angiver observationer. Den stiplede firkant er et gæt på placeringen af en kasse.

En måde at bestemme en kasses position og orientering er ved at udvælge en mænge af observationer, som menes at stamme fra samme kasse. Dette kunne gøres ved at vælge alle punkter inde for at bestemt område. Hvis kassens centrum er (x, y) og dens orientering er v, kan afstanden fra en observation til kassen bestemmes. På figur 4.4 er dette skitseret. Stregerne mellem observationerne og kassen angiver den korteste afstand. Hvis afstanden fra observation i til kassen betegnes r_i kan vælges at tro at kassens placering er bestemt af de x, y og v som minimerer:

$$\sum_{i=1}^{n} r_i^2$$

x, y og v kan bestemmes med diverse optimerings algoritmer fra numerisk-matematik.

Kapitel 5

Agent-arkitekturen

Som udgangspunkt er valgt at designe en arkitektur til scenariet "rengøringsrobotter" beskrevet i afsnit 4.7. Arkitekturen skal dog opbygges så fleksibelt som muligt, så den "nemt" kan udvides til også at kunne anvendes på lagerhalsrobotter.

Resten af dette kapitel beskriver reelt to forskellige ting:

1. En *agent-arkitektur-model*, som beskriver det *framework* af logik, komponenter, kommunikation osv., som er til rådighed for den endelige arkitektur.
2. En *arkitektur* designet udfra med ud fra *arkitektur-model*.

I det følgende beskrives først *arkitektur-model* i korte træk, hvorefter den egentlige arkitektur beskrives, hvor denne beskrivelse tjener som eksempel, der uddyber beskrivelsen af *arkitektur-model*.

5.1 Arkitektur-modellen

Som beskrevet i afsnit 2.5 giver [10] en god beskrivelse af en mulig arkitektur-model. Det har dog været nødvendigt at bruge en anden model, som er nemmere at implementere, men som er inspireret af [10].

Den valgte model består at en samling af komponenter, som er forbundet af en *kommunikationsbus*. En *kommunikationsbus* er en kommunikationsforbindelse som sender beskeder rundt imellem komponenterne. En besked

fra en komponent bliver sendt ud til alle andre komponenter. Der kan være et vilkårligt antal komponenter i arkitekturen.

Hvilke komponenter der benyttes, og hvordan arbejdesfordelingen er mellem dem afhænger helt af anvendelsen af agenten.

De enkelte komponenter designes og implementeres helt individuelt. De skal blot kunne kommunikere med modellens kommunikationsbus. En komponent har ikke noget direkte kendskab til hvilke øvrige komponenter der findes, eller hvordan de er designet. Men det ligger implicit i designet af komponenten.

Der er typisk brug for komponenter, som er baseret på logiske repræsentation og ræsonneringer, og for komponenter som er baseret på mere "sædvanlige" programmeringsteknikker. For de første typer af komponenter er lavet et logiksystem, som er beskrevet i de to følgende afsnit.

5.1.1 Logiksystemet

Komponenterne har mulighed for at benytte et logiksystem som er baseret på DATALOG[1] (beskrevet i afsnit 2.1.1).

Da det ikke har været muligt at implementere SLD-algoritmen, eller have lavet en form for sammenspil med et prolog system (som beskrevet i afsnit 2.1.5, er logiksystemet her anvendt noget simplificeret i forhold til DATALOG.

I DATALOG består en videnbase af en mængde af regler og atomiske udsagn. Men i dette system deles det op i to dele, som her kaldes henholdsvis for *faktabase* og *regelbase*. En *faktabase* er en mængde af atomiske udsagn på grundform. Mens en *regelbase* er en mængde af regler, som kan indeholde både konstanter og variabler. Sammensatte termer (funktorer) kan ikke benyttes.

Til systemet haves en funktion $E(r, F)$, som for en given regel r og en Faktabase F, beregner en liste af atomiske udsagn, som kan udledes af reglen udfra udsagn F. Det vil sige, at hvis der haves en Faktabase F:

$$F = \{p(a,c), p(b,c), q(a,d), q(b,e)\}$$

[1] Efter som komponenterne implementeres individuelt kan de opbygges af diverse logiksystemer, når blot der haves en implementering af det.

og en regel r:

$$r = r(X,Y) \leftarrow p(X,c) \wedge q(X,Y)$$

vil gælde at:

$$E(r,F) = \{r(a,d), r(c,e)\}$$

Nøglefelter

Faktabasen benyttes som database, mens *regelbasen* benyttes til dels at generere ny "data" og dels til at opdatere *faktabasen*. Hvis reglen r bruges til at udvide faktabasen F fås en ny faktabase F':

$$\begin{aligned} F' &= F + E(r,F) \\ &= \{p(a,c), p(b,c), q(a,d), q(b,e)\} + \{r(a,d), r(b,e)\} \\ &= \{p(a,c), p(b,c), q(a,d), q(b,e), r(a,d), r(b,e)\} \end{aligned}$$

Dette er godt nok sålænge *faktabasen* udvides, men ikke når man vil opdatere viden. Udsagnet $p(a,c)$ fra ovenstående eksempel kunne i en konkret fortolkning angive at agenten a arbejder på opgaven c. Hvis en regel så udleder udsagnet $p(a,d)$ (agenten a arbejder nu på opgave d), ønskes ikke at faktabasen indeholder både $p(a,c)$ og $p(a,d)$, men kun $p(a,d)$. Når det antages, at agenten kun kan arbejde på en opgave ad gangen.

Dette løses ved at prædikater kan have et *nøglefelt*, som det kendes fra sædvanlige databaser. Hvis et prædiktat i en faktabase har et nøglefelt, kan flere atomiske udsagn bestående af dette prædikat ikke indgå i faktabasen, hvis deres nøgleværdi er ens. Det vil sige, at hvis ovenstående prædikat p's første argument (term) er et nøglefelt kan $p(a,c)$ og $p(a,d)$ ikke indgå i samme faktabase, mens $p(a,c)$ og $p(b,d)$ godt kan.

For overskuelighedens skyld skrives, i denne rapport, prædikater, som har et nøglefelt med en streg over. F.eks. $\overline{p}(a,c)$. Såfremt et prædikat har et nøglefelt, er det altid det første argument, som er nøgleværdien.

Hvis man har en faktabase F:

$$F = \{\overline{p}(a,c), q(a,e), \overline{p}(b,c)\}$$

hvor p har et nøglefelt og q ikke har, og denne opdateres med udsagene: $\overline{p}(a,d)$ og $q(a,f)$ fås F':

$$\begin{aligned} F' &= \{\overline{p}(a,c), q(a,e), \overline{p}(b,c)\} + \overline{p}(a,d) + q(a,f) \\ &= \{q(a,e), \overline{p}(b,c), \overline{p}(a,d), q(a,f)\} \end{aligned}$$

Bemærk at $\overline{p}(a,d)$ har erstattet $\overline{p}(a,c)$, og $q(a,f)$ tilføjet.

5.1.2 Håndtering af logiske regler og beskeder

Komponenter som benytter ovenstående logiksystem har typisk én faktabase og én regelbase. Faktabasen indeholder den information komponenten har. Eksempelvis har mål-komponenten en faktabase med information om hvilke mål agenten har, prioteringen af dem osv, samt en regelbase der indeholder en række regler, som beskriver komponentens funktionalitet.

Når en besked m_{ind} modtages, adderes beskedens indhold til den øvrige information komponenten har. Herefter evalueres alle regler i regelbasen og resultatet adderes til faktabasen. Når dette er gjort fjernes beskeden igen fra faktabasen. Dette kan beskrives i et pseudosprog som:

$$\begin{array}{l} F = F + m_{ind} \\ \textit{forall} \quad r \in R \\ \qquad F = F + E(F,r) \\ F = F - m_{ind} \end{array}$$

hvor F er komponentens faktabase, R er regelbasen og r er en regel fra R.

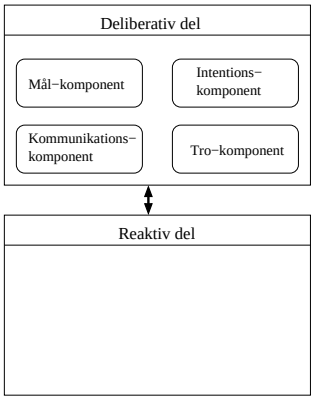
Når en besked er modtaget og faktabasen er opdateret med ovenstående algoritme, checkes om den eventuelt nye informaiton giver anledning til afsende en besked. Til dette har komponenten en liste af beskeder (svarende til beskederne listet under "besked ud" på figure 5.3) sammen med betingelser for at afsende dem. Som notation bruges:

$$p_1, \ldots, p_n \longmapsto m_{ud} \tag{5.1}$$

der angiver at beskeden m_{ud} sendes, hvis de atomiske udsagn p_1 til p_n er opfyldt i faktabasen.

5.2 Den valgte arkitektur

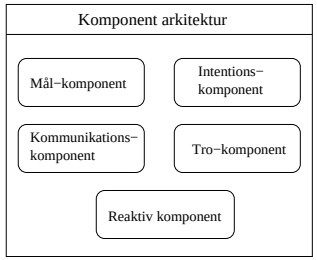
Den valgte arkitektur bygger på den flere gange nævnte hybrid-arkitektur med en deliberativ del og en reaktiv del. Den deliberative del opbygges som en BDI-agent med de tre komponenter tro, mål og intention. Desuden tilføjes en kommunikationskomponent til kommunikation/forhandling med andre agenter. En sådan arkitektur er skitseret på figur 5.1.



Figur 5.1: En hybrid-arkitektur

Men efter som den reaktive del ikke består af flere komponenter, og der i hele arkitekturen ikke er *mange* komponenter, fås en mere overskuelig arkitektur-model ved at betragte den reaktive del som en komponent på lige fod med de andre. Dette giver en "1-lags" model, som udelukkende består af komponenter. Komponent-arkitekturen er skitseret i figur 5.2.

Overordnet set fungerer agenten ved: 1) Mål-komponenten udvælger et mål som skal udføres. 2) Intentions-komponenten står for udførelsen af dette



Figur 5.2: En komponent-arkitektur

mål ved at sende "simple" kommandoer til den reaktive del. 3) Den reaktive del udfører disse kommandoer, og sender undervejs observeret information til tro-komponenten. Alle komponenterne kan hente oplysninger i tro-komponenten.

5.2.1 Kommunikation mellem komponenter

Der benyttes ikke broregler i arkitekturen. Istedet for kommunikerer komponenterne via beskeder.

Det er naturligvis ikke alle beskeder, der er interessante for den enkelte komponent. Derfor har komponenterne et filter, som bestemmer hvilke beskeder komponenten skal reagere på. En besked har form som et atomisk udsagn, hvor prædikatetes navn angiver beskedens betydning, og prædikatets termer er argumenter[2] til beskeden. Eksempelvis betyder beskeden:

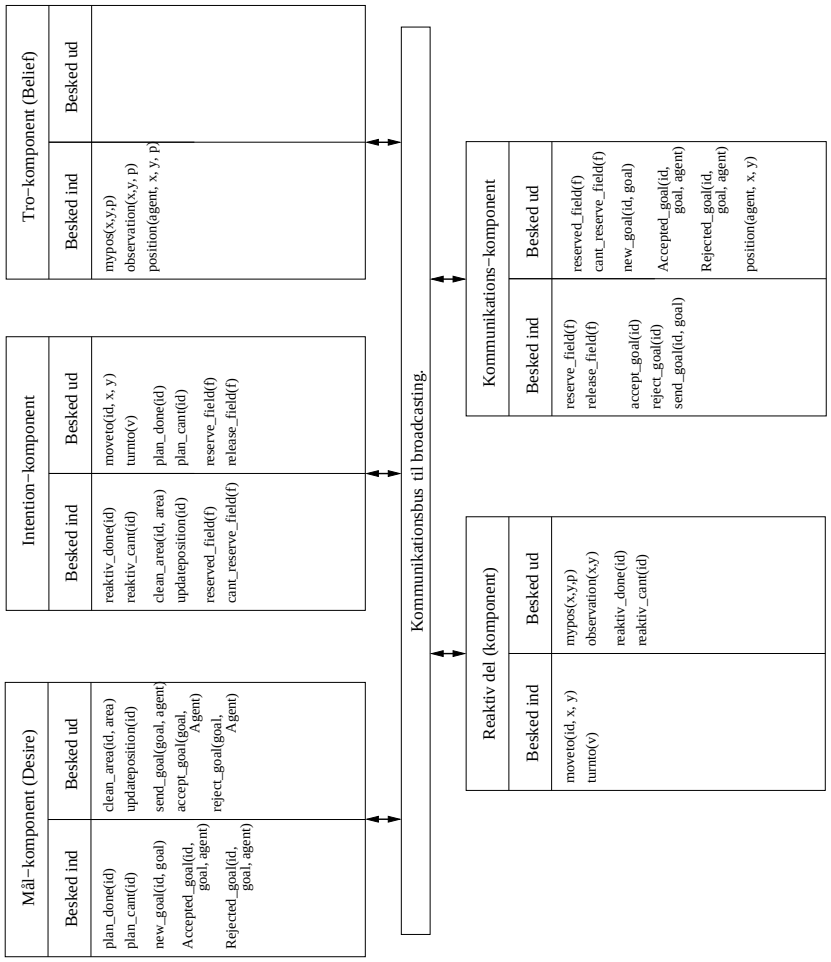
$$plan\_done(id)$$

at planen *id* er udført.

På figur 5.3 er for hver komponent angivet hvilke beskeder, der kommer igennem filteret. De er listet under "Besked ind". På samme figur er under "Besked ud", angivet hvilke beskeder komponenten har mulighed for at sende.

Kommunikationen med beskeder burde være den eneste kommunikation mellem komponenterne. Men for at gøre implementeringen nemmere har det

[2]Ikke diskutionsmæssige argumenter, men som argumenter til en funktion



Figur 5.3: Skitse over de fem komponenter, forbundet med en kommunikationsbus. For hver komponent er angivet hvilke beskeder, der bliver sendt og modtages.

været nødvendigt med en undtagelse. Undtagelsen er en ekstra kommunikation mellem tro-komponenten og de øvrige komponenter. I tro-komponenten findes funktioner til f.eks. ruteplanlægning, og disse funktioner kan kaldes direkte fra de andre komponenter. Præcis hvilke funktioner de drejer sig om er beskrevet i afsnittet om tro-komponenten (5.3). Desuden er al informationen i tro-komponenten tilgængelig for de andre komponenter. Det betyder, at en komponent kan have et udtryk som:

$$p(\ldots) \leftarrow B : q(\ldots) \tag{5.2}$$

hvor $q(\ldots)$ bliver evalueret i tro-komponenten.

5.3 Tro-komponenten – opbygning af kvalitativ database

Tro-komponentens opgave er dels at lagre den information, som agenten opsamler, og dels at have et sæt af algoritmer, som de øvrige komponenter kan benytte.

Informationen skal dog ikke *bare* akkumuleres og stilles til rådighed. Informationen skal bearbejdes til en *kvalitativ database*, hvor unødvendige detaljer er fjernet. Eksempelvis er det i nogle situationer mere væsenligt, om det er muligt at komme til en bestemt position, frem for viden om hvor der er observeret forhindringer.

For at diskretisere verden opdeles denne i tern, som på et skakbræt. Felterne har en størrelse af nogenlunde samme størrelse som robotten selv. For hvert felt gemmes information om, hvorvidt det er ledigt eller om det er blokeret, og hvorvidt feltet er rengjort.

Som det kan ses på figur 5.3 er der ingen udgående beskeder fra tro-komponenten. Det er fordi den er designet som en "passiv" komponent. Den samler information, som den får gennem kommunikationsbussen, men foretager sig derud over ikke noget aktivt. Komponenten har nogle funktioner, som kan benyttes af de øvrige komponenter, men dette foregår, som tidligere skrevet, **ikke** igennem kommunikationssbusen. De tilgængelige funktioner er:

- *GetPath(to)*: Beregner rute fra aktuel position til positionen (feltet) *to*
- *GetCleaningPath(area)*: Beregner en rute som dækker hele området *area*
- Eval(p): Evaluere prædikatet p.

De prædikater som kan evalueres af tro-komponenter er de, som i beskrivelsen af mål-komponenten starter med 'B:', eksempelvis *B:done_ desire(Id)*. Først ses på hvilken information det er at tro-komponenten skal gemme. Som udgangspunkt ses på de beskeder komponenten er intresseret i:

- mypos(x,y,p): Egen position er (x, y)
- position(agent,x,y): Agenten *agent* er på positionen (x, y)
- observation(x,y,p): Observation fortaget på positionen (x, y)

For alle tre beskeder angiver parameteren p hvilken præcision (x, y) er angivet med. Denne information kan gemmes med to tilsvarende prædikater:

- $\overline{\text{position}}(Agent, X, Y, P)$
- observation(X, Y, P)

hvor beskeden *mypos(x, y, p)* gemmes som $\overline{\text{position}}(me, x, y, p)$.

Tro-komponenten er dog ikke opbygget med logiksystemet beskrevet i afsnit 5.1.1. Så informationen gemmes i to tabeller. Position på agenter gemmes i en liste som:

Agent	Position	Felt
string	(x,y)	index
...	...	...

hvor *position* er den præcise position, og *felt* er det felt hvor denne position ligger i.

Observationer gemmes i en liste som:

Position	Felt	TTL
(x, y)	index	tæller
...	...	...

hvor *Position* og *Felt* er som position og tilhørende felt. *TTL* (*Time to live*) er en tæller som sættes til 100 på det tidspunkt observationen er foretaget, og reduceres med fast tidsinterval. Når *TTL* er 0 fjernes observationen.

5.3.1 Observationer

En observation kan være fra en forhindring placeret i agentens verden, som ikke kan flyttes, det kan være en anden agent, eller observationen kan skyldes en "sensor-fejl". I første tilfælde er observationen reel og skal gemmes, således at agenten ikke forsøger at køre den vej igen. Stammer observationen fra en anden agent, skal den ikke gemmes, da observationen kun er midlertidig. Endelig hvis observationen er en fejl, bør den ignoreres.

Når en observations besked modtages checkes om denne ligger, inden for en passende margen, tæt på en anden agent. Hvis dette er tilfældet slettes observationen. Hvis der ikke er en agent tæt på gemmes observationen som en reel observation. Det betyder at "fantom-observationer" også vil blive gemt. Hvilket naturligvis er et stort problem hvis der er for mange. En mulig løsning, som anbefales i [5], er at give en observation en begrænset levetid. Det vil sige en observation forsvinder efter en vist tidsrum. Dette har dog den ulempe at reel indformation også bliver slettet.

5.3.2 Et "landkort"

Informationen fra de to ovenstående tabeller indtegnes på "landkort". Efter som agenten inddeler "verden" i felter, består kortet af en liste af felter. For hvert felt angives om der er observationer og andre agenter placeret på dette felt. (Et grafisk billede af dette kan ses på figur 7.2 og 7.3 i næste kapitel).

5.4 Mål-komponenten – agentens motivation

Mål-komponentens opgave er at udvælge et mål som intentions-komponenten skal arbejde mod.

Mål-komponenten er opbygget omkring logiksystemet beskrevet i afsnit 5.1.1. Agenten er designet til at kunne håndtere to forskellige mål:

- $cleanArea(f_1, f_2, Priority)$
- $updatePosition(Priority)$

Det første mål angiver, at området der har felt f_1 som øverste venstre hjørne og f_2 som nederste højre hjørne skal rengøres. Andet mål betyder, at robotten skal opdatere sin position. (Opdatering af position er ikke

medtaget i implementeringen af agenten, men er medtaget i designet for at demonstrere at agenten kan have flere forskellige mål).

Til hvert mål knyttes et unikt id nummer, en prioritet og en statusangivelse. Dette udtrykkes med fire prædikater:

- $\overline{desire}(Id, Type)$, hvor Type er *cleanArea*(f_1, f_2) eller *updatePosition*
- $\overline{priority}(Id, Priority)$, hvor *Priority* er 1 (lavest prioritet) , 2, 3, 4 eller 5 (højeste prioritet)
- $\overline{status}(Id, Status)$, hvor *Status* kan være *none*, *waiting_ reply*, *done*, *working_ on*.

Hvad de enkelte statusindikeringer, betyder forklareres i det følgende. I det ovenstående er *cleanArea*(f_1, f_2) en konstant, og *ikke* en funktor. Konstanten kunne altså også kaldes: *cleanArea_f1_f2)*.

For at angive forholdet mellem prioriteringer bruges prædikaterne *dec*(P_1, P_2) , *inc*(P_1, P_2) og *greather_ than*(P_1, P_2).

inc og *dec* angiver henholdvis en forøgelse og en reduktion af prioriteten, og *greather_ than*(P_1, P_2) angiver P_1 er størren end P_2. Disse udsagn instantieres til:

- *inc: inc(1,2), inc(2, 3), inc(3, 4), inc(4, 5), inc(5, 5)*
- *dec: dec(5,4), dec(4, 3), dec(3, 2), dec(2, 1), dec(1, 1)*
- *greather_ than*: *greather_ than*$(5, 4)$, ..., *greather_ than*$(2, 1)$

Udover ovenstående prædikater, benyttes:

- *max_ priority(Id, S)* : *Id* er det mål med højest prioritet som har *Status*
- *small_ desire(Id)* : *Id* er et "lille"mål (eller atomisk mål) som ikke kan opdeles i undermål.
- *working_ in(Id)* : agenten arbejder med at løse målet *Id*.
- *next_ id(Id* : Id på det næste "nye" mål.

De ovenstående udsagn er ikke bestemt af definitte regler, men beregnes af en dedikeret funktion.

Endelig benytte nogle udsagn som evalueres i agentes tro-komponent:

- *B:done_ desire(Type)* : Målet *Type* er opfyldt.
- *B:notdone_ desire(Type)* : Målet *Type* er ikke opfyldt.
- *B:agent(Agent)* : *Agent* er en agent.
- *B:agent_ max_ desire(Agent, Prio)* : Tro-komponentens antagelse om hvilken prioritet det højst prioriterede mål er for agenten *Agent*.

5.4.1 Funktionalitet og regler

I dette afsnit beskrives komponentens funktionalitet, samt de regler, til komponentens faktabase, der udtrykker det.

Komponenten har tre muligheder for at opfylde et mål:

- Intentions-komponenten kan sættes til at arbejde mod målet (lave en plan som vil opfylde målet, og forsøge at udføre den). Når intentions-komponenten arbejder mod et mål, sættes status for målet til *working_ on)*
- Opdele målet til flere delmål.
- Forsøge at få en anden agent til at overtage målet. Når et mål er sendt til en anden agent, og der stadig ikke er kommet svar på, hvor vidt den anden agent er villig til at overtage målet, sættes status til *waiting_ reply*.

Når et mål er opfyldt sættes status til *done*. Når et mål er overtaget af en anden agent, eller er opdelt til undermål, betragtes målet som opfyldt af mål-komponenten (men ikke af tro-komponenten). Som udgangspunkt antages at intentions-komponenten kun kan arbejde med en plan af gangen, og dermed kun mod et mål af gangen. Desuden arbejdes kun på mål som ikke kan opdeles yderligere.

working_ on(none) angiver at intentions-komponenten mangler et mål at arbejde imod. Er dette tilfældet udvælges et nyt mål:

$$status(Id,working\_on) \leftarrow working\_on(none), status(Id,none), max\_priotet(Id,none), small\_desire(Id)$$

I næste afsnit beskrives hvordan dette bliver kommunikeret til intentions-komponenten via en besked.

Et mål kan være opfyldt ved et "tilfælde", det vil sige uden at agenten har haft intention om det. Derfor checkes i tro-komponenten om målet er opfyldt:

$$status(Id,\ done) \leftarrow desire(Id,\ Type), B{:}done\_desire(Type)$$

Når komponenten modtager en besked bliver indholdet af denne besked tilføjet til komponentens faktabase som beskrevet i afsnit 5.1.2. Eksempelvis kunne intentions-komponenten have sendt beskeden *done(Id)*. Dette angives i denne beskrivelse som *I:done(Id)* som angiver, at det er en oplysning, som stammer fra intentions-komponenten. Det tilføjede "*I:*" benyttes ikke i implementeringen.

Når intentions-komponenten har udført en opgave, sender den beskeden done(Id). Så fremt tro-komponenten er enig heri betragtes målet som opfyldt:

$$status(Id,\ done) \leftarrow I{:}done(Id), B{:}done\_desire(Id)$$

Er tro-komponenten derimod ikke enig haves et dilemma, enten tager tro-komponent eller intentions-komponenten fejl. Det vælges at antage at intentions-komponenten har et problem med at opfylde målet. Målet betragtes derfor som ikke opfyldt. Når et mål ikke kan opfyldes reduceres dets prioritet. Det betyder, at hvorvidt agenten skal forsøge igen, eller udføre at andet mål afhænger af prioriteringerne mellem målene.

Status ændres til *none*:

$$status(Id,\ none) \leftarrow I{:}done(Id), B{:}notdone\_desire(Id)$$

og prioteringen reduceres:

$$\begin{aligned} priority(Id,NewPriority) \leftarrow\ & I{:}done(Id), \\ & B{:}notdone\_desire(Id), priority(Id,Priority) \\ & dec(Priority,NewPriority) \end{aligned}$$

Hvis intentions-komponenten ikke kan udføre en opgave sender den *cant(Id)*. Igen, som for ovenstående situation betragtes målet som ikke opnået, og prioriteringen reduceres.

$$status(Id,none) \leftarrow I{:}cant(Id), B{:}notdone\_desire(Id)$$

$$\begin{aligned} priority(Id,NewPriotet) \leftarrow\ & I : cant(Id) B : notdone\_desire(Id), \\ & priority(Id,Priority), dec(Priority,NewPriority) \end{aligned}$$

Et mål sendes til en anden agent, hvis det antages at den anden agents højeste prioritet er lavere end dette mål:

$$\begin{aligned} send(Agent,Id) \leftarrow\ & priority(Id,Prio), status(Id,\ none \\ & B{:}agent(Agent), small\_desire(Id) \\ & B{:}agent\_max\_desire(Agent,Prio\_other), \\ & greather\_than(Prio,Prio\_other) \end{aligned}$$

Status ændres til *wait_reply*:

$$status(Id,wait\_reply) \leftarrow send(Agent,Id)$$

Hvis en anden agent accepterer at overtage et mål, modtages *accepted(Id,Agent)* fra kommunikations-komponenten, og målet betragtes som opfyldt:

$$status(Id,done) \leftarrow C{:}accepted(Id,Agent)$$

Ved denne besked opdateres agentens tro på den anden agents højeste prioritet. Dette foregår i tro-komponenten, men er angivet her for overblikkets skyld:

$$\begin{aligned} B{:}agent\_max\_desire(Agent,Prio) \leftarrow\ & C{:}accepted(Id,Agent) \\ & priority(Id,Prio) \end{aligned}$$

Hvis en agent afviser at overtage et mål, betragtes målet som uopnået:

$$status(Id,none) \leftarrow C{:}rejected(Id,Agent)$$

og igen kan antagelsen om den anden agents maksimale prioritet ændres:

$$B{:}agent\_max\_desire(Agent, NewPrio) \leftarrow C{:}rejected(Id,Agent), priority(Id, P_1), inc(P_1, P2) \; inc(P_1, P2)$$

Når en anden agent forspørger om komponenten vil overtage et mål, sammenlignes prioriteringen af målet, med prioriteringen af eget højst prioriterede mål.

$$new\_desire(NewId, Type, Prio) \leftarrow C{:}incomming(Type, Prio, Agent) \; max\_priority(Id,none), \; priority(Id,MyMaxPrio) \; greather\_than(Prio,MyMaxPrio), \; next\_id(NewId)$$

Et nyt mål udtrykkes med de tidligere beskrevene prædikater:

$$desire(NewId,Type) \leftarrow new\_desire(NewId,Type,Prio)$$

$$status(NewId,none) \leftarrow new\_desire(NewId,Type,Prio)$$

$$priority(NewId,Prio) \leftarrow new\_desire(NewId,Type,Prio)$$

Hvis målet accepteres, sendes dette til den forespørgende agent:

$$send\_accept(Type,Agent) \leftarrow C{:}incomming(Type,Prio,Agent) \; new\_desire(NewId,Type,Prio)$$

Kan målet ikke overtaget, afvises det overfor den forespørgende agent:

$$send\_reject(Type,Agent) \leftarrow C{:}incomming(Type,Prio,Agent), \; max\_priority(Id,none), \; priority(Id,MyMaxPrio), \; greather\_than(Prio,MyMaxPrio), \; next\_id(NewId)$$

Opdeling af mål

Agenten håndterer kun mål som ikke kan opdeles yderligere (*small_desire(Id)*, så i stedet for at udtrykke denne opdeling i form af regler, er lavet en funktion som opdeler alle mål til "atomisk mål". Denne funktion må naturligvis afhænge af målet. Et "rengøringsmål" opdeles ved at dele området i to dele. Det vælges at betragte et "rengøringsmål" som atomisk, når området fylder under en 10. del af agentens verden.

5.4.2 Afsendelse af beskeder

Komponenten sender tre typer af beskeder: Accept eller afvisning af et mål. Ved afsendelse af mål til en anden agent og ved nyt mål til intentions-komponenten.

Afsendelse af mål til anden agent:

$$send(To,\ Id), desire(Id,\ Type) \longmapsto send\_goal(To,\ Type) \qquad (5.3)$$

Ny opgave til intentions-komponenten:

$$working\_on(Id), desire(Id,\ Type) \longmapsto working\_on(Type) \qquad (5.4)$$

Hvis et mål "tilfældigt" er opfyldt skal intentions-komponenten stoppe den aktuelle opgave:

$$working\_on(none) \longmapsto working\_on(none) \tag{5.5}$$

Når et mål fra en anden agent enten accepteres eller afvises sende:

$$send\_accept(Type,Agent) \longmapsto send\_accept(Type,Agent) \tag{5.6}$$

$$send\_reject(Type,Agent) \longmapsto send\_reject(Type,Agent) \tag{5.7}$$

5.5 Den reaktive komponent

Den reaktive komponents opgave er at styre og kontrollere robottens sensorer og aktorer. Den får kommandoer fra intentions-komponenten, som den udfører uden nærmere overvejelser.

Den reaktive komponent kan udføre nedenstående kommandoer. For hver kommando angives et unikt idenfikaitons nummer (id):

- `moveto(`id, x, y, θ`)`: Kører robotten i en lige linie til punktet x,y, og herefter drejer til orienteringen θ. Undervejs checkes for objekter der blokerer vejen.
- `calibrate_pos(`id, θ`, `δ`)`: Kaliberer agentes position. I retningen θ findes et objekt med absolut position δ. (δ angiver x eller y afhængig af θ

Når en kommando er udført sender den reaktive komponent svaret `reaktiv_don(id)`, som angiver at kommandoen med id er udført. Hvis en kommando ikke kan udføres sendes svaret `reaktiv_cant(id)`.

Id nummeret på kommandoerne giver kommunikationsmæssig mulighed for at den reaktive komponent kan udføre flere kommandoer samtidig (eller have en kommandokø).

Sensorer

Robottens afstandsmålere checkes med fast tidsinterval. Hvis "noget" observeres af en af afstandsmålerne standses robotten med det samme, da

objektet som er observeret, med stor sandsynlighed blokerer robottens vej. Når robotten er standset foretages gentagne aflæsninger af afstandsmålerne for at kontrollere rigtigheden af den første måling. Hvis det viser sig, at der ikke var noget observeret, fortsætter robotten ad sin rute. Ellers sendes en observationsbesked: `observation(`x, y, ψ`)`, der angiver at et object er observeret på koordinat (x, y). ψ angiver robottens forventede præcision af denne observation position. (som afhænger meget af præcision af robottens egen position).

Den reaktive komponent beregner desuden løbende robottens position. Algoritmer for dette, samt for den øvrige brug af robottens sensorer og aktorer er beskrevet i kapitel 3.

5.6 Intentions-komponenten – udførelse af planer

Intentions-komponenten modtager besked fra mål-komponenten om hvilken opgave der skal løses. I denne agent er det eneste mål at rengøre et område. En rute der dækker området hentes i tro-komponenten. Ruten sendes til den reaktive komponenten i form af "moveto" kommandoer.

De er flere ting der kan gå galt undervejs:

1. Den reaktive komponent støder på en forhindring. (Ruten er blokeret)
2. Tro-komponenten kan ikke finde en rute
3. Der kan ikke reserveres de nødvendige felter

Problemet er at komponenten ikke ved om et problem er opstået fordi opgaven reelt er uløselig eller om det blot skyldes "forkert håndtering" af opgaven.

Hvis ruten er spæret, hentes blot en ny rute fra tro-komponenten. Dette hændre ikke udførslen af opgaven sålænge der findes en alternativ rute.

Hvis Tro-komponenten ikke kan finde en rute opgives opgaven med det samme. At der ikke findes en rute betyder dog ikke at opgaven er uløselig, da ruten kan være spæret af en anden agent, som senere flytter sig. Det er derfor vigtigt at mål-komponenten ikke opgiver et mål, blot fordi intentions-komponenten ikke kan opfylde målet.

Hvis der ikke kan reserveres de felter, som er nødvendige, prøver komponenten med en nye rute, som muligvis indeholder felter, som kan reserveres.

Da dette er en oplagt dead-lock situation, sættes et maksimalt antal gange dette kan forsøges før opgaven betragtes som mislykkes. Hvormange gange agenten skal forsøg, må bestemmes ved ekspermentation.

5.7 Kommunikationskomponenten

Agentens kommunikations-komponent er ydderst simpel, i det at den ikke indeholder nogen form for forhandling. Det betyder at den reelt blot vidre sender beskeder melle de øvrige komponenter og andre agenter.

Kommunikations-komponenten er derfor til en vis grad overflødig, men tjener til at agenten nemt (implementeringsmæssigt) kan udvides med et forhandlingssystem.

Kommunikations-komponenten har dog en mindre opgave, når en anden agent vil reservere at felt er det kommunikatione-komponenten der acceptere eller afviser dette. Dette gøres på baggrund af hvorvidt feltet er anvendt af agent agent eller ej. Dette er dog en yderest simpel måde at håndtere problemet på. En mere passende løsning ville være at lade intentionskomponenten håndtere hvilke felter der kan reserveres, idet denne komponent har information om hvikle felter der skal bruges i nærmeste fremtid.

5.8 En planlægningskomponent

Arkitekturen indeholder ikke en egentlig planlægningskomponent. Det betyder at agenten kun kan udføre standard planer, og derfor ikke er videre fleksibel. Arkitekturen, opbygget med moduler, giver dog gode mulighed for at udvide med en planlægnings-komponent. Dette ville kun kræve mindre ændringer i intentions-komponenten, som skulle hente planer fra planlægnings-komponenten fremfor i tro-komponenten.

En planlægningskomponent (eller en planlægningsmekanisme i en af de eksisterne komponenter) vil være nødvendig hvis agenten skulle udføre mere kompliceret opgaver, som for eksempelvis lagerhaldsrobotten.

Kapitel 6

Implementering af arkitekturen

Der er i dette kapitel forsøgt at udvælge de meste interessante dele af implementeringen. Flere steder i det følgende er indsat "bidder" af koden, hvilke dog oftest er forkortet. Tre punktummer (`...`) benyttes til at angive, at der er fjernet noget af koden. Den fulde kode kan ses i bilag C. På side 113 er et index over filer og klasser.

6.1 Oversigt

Der er to mulighed for at implementere en komponent-arkitektur. Den ene er at implementere hver komponent i hver sin tråd, eller komponenterne kan implementeres som selvstændige programmer. Den sidste løsning er valgt. Der er dog lavet en "short-cut" i det et komponenterne er tro, mål og intention er implementeret i samme program. Dette samlede program fungere ved at når en besked til en ad de tre komponenter modtages, eksekveres den eller de komponenter, som skal bruge beskeden. Denne samlede implementering findes i filen `Deliberativ_v1.cc`. De tro komponenter er implementeret i de tre klasser `TBelief`, `TDesire` og `TIntention`.

6.2 Diverse "værktøjer"

I dette afsnit beskrives kort nogle "værktøjs" klasser, som anvendes i implementeringen.

6.2.1 Lister som *dynamisk array*

I hele implementeringen bruges dynamiske arrays til at implementere lister og mængder. Til dette bruges en *template* som hedder `Dynamic_array`. Denne datastruktur kan indsætte nye elementer i konstant tid $O(1)$, og finde elementer i listen i lignær tid $O(n)$. Såfremt listen indeholder n elementer. For eksempel en liste, *knowledge*, af prædikater:

```
Dynamic_array<TPredicates *> knowledge;
```

For at tilføje elementer benyttes `knowledge.add( p )`, og det i'te element hentes fra listen med `knowledge[i]`.

6.2.2 Konfigurering af systemet

Der er en del parametre i agenterne, som det er hensigtsmæssigt at kunne eksperimentere med uden at skulle oversætte programmet igen. Disse parametre beskrives i en konfigurations fil. I bilag A kan ses den fil der er blevet benyttet under forsøgene med agenterne. Filen læses linie for linie. En linie består af en "ligning" på formen:

$$variabel = værdi$$

Hvis en linie enten er tom eller er en "kommentar-linie", som starter men tegnet '%' bliver linien ignoreret.

Til at læse filen bruges klassen `TConfig`. Klassen initialiseres med navnet på konfigurationsfilen. Til at hente værdier fra klassen er brugt C++ operatoren `[ ]` som overloades. Typisk bruges et globalt objekt af `TConfig` klassen, så de beskrevne parametre hele tiden er tilgængelige. Eksempel på brugen af `TConfig`:

```
TConfig c("config.txt");

// later...
int speed = c["movespeed"].AsInt();
```

Eksemplet giver variablen `speed` værdien som er tildelt til `movespeed` i konfigurationsfilen.

6.2.3 Grafik på X-terminaler

For at kunne følge med i hvad der sker "inden i" agenterne er det nødvendigt med nogle "monitor-systemer". En del af agenternes aktivitet kan vises som tekst i et almindeligt terminal vindue. Men for at vise f.eks. agentens opbyggede kort, er grafisk illustration praktisk talt nødvendigt.

Der findes flere forskellige programmer/biblioteker, som kan lave grafik til X-windows[1]-systemet. Her er dog valgt at skrive en klasse direkte til X via `Xlib` biblioteket, som følger med alle X installationer. `Xlib` har den fordel, at det er nemt at lave simpel grafik, mens det ikke er egnet til brugergrænseflader med knapper og lignende.

Klassen hedder `TViewport` og den fulde kode kan ses i bilag C. Hver instans af `TViewport` tegner et nyt vindue. Klassen initialiseres med:

```
TViewport(  uint win_width, uint win_height,
            uint view_width, uint view_height,
            uint border_size, TText title);
```

hvor `win_width` og `win_height` angiver vinduets størrelse i pixels. `view_width` og `view_height` angiver størrelsen på det koordinatsystem, der "tegnes i". `border_size` angiver en ramme inde i vinduet, hvor der ikke kan tegnes og `title` angiver navnet på vinduet.

`TViewport` har en række metoder til at tegne i vinduet bl.a.:

```
void DrawLine(float x1, float y1, float x2, float y2,
              TText color)
```

[1] X-windows, eller blot X, er det grafiske vinduessytem der benyttes på de fleste Linux systemer


```
void DrawString(float x, float y, char *str)
void DrawCircle(float x, float y, float r)
```

Disse funktioner har tilsvarende funktioner i `Xlib`, koordinaterne skal blot omregnes for at passe til det valgte koordinatsystem. Punktet (0,0) vælges placeret i nederste venstre hjørne, men i X-windows ligger dette punkt i øverste venstre hjørne. Alle y-koordinater "vendes" defor:

$$y = \texttt{view\_height} - y$$

Herefter omregnes koordinaterne til skærm-koordinater (angivet i pixels):

$$y_s = (\frac{\texttt{w\_height} - 2 \times \texttt{border}}{\texttt{v\_height}})y + \texttt{border}$$

$$x_s = (\frac{\texttt{w\_width} - 2 \times \texttt{border}}{\texttt{v\_width}})x + \texttt{border}$$

Endelig kan alt i vinduet slettes med metoden: `void Clear()`

Vinduet bliver fjernet fra X-serveren når `TViewport`'s dekonstruktor kaldes.

6.3 Kommunikation mellem moduler

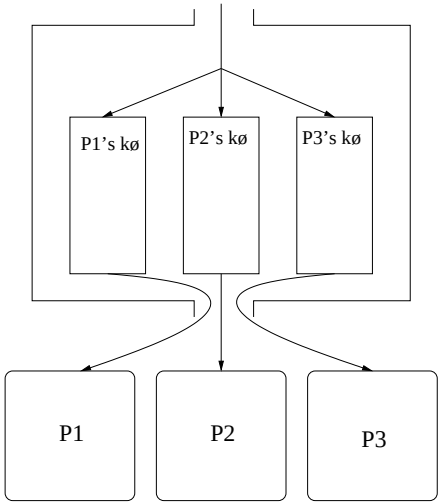
Kommunikation mellem de enkelte program-moduler kan ske på flere måder. Her er valgt at bruge *sockets*. En *socket* er en to-vejs kommunikationsforbindelse, der kan oprettes mellem to kørende programmer. Programmerne behøver ikke at køre på samme computer sålænge IP-nummeret (netværks adressen) på computerne er kendt. Som beskrevet i kapitel 5 skal kommunikationen fungere som en *bus*, hvor hver besked sendes videre til alle program moduler. Det vil derfor være smart med et kommunikationssystem, som ikke skal konfigureres til hvormange eller hvilke program moduler der indgår i arkitekturen.

Løsningen, der her er valgt, er en client/server lignende model, hvor program-modulerne er klienter, og der laves en kommunikationsserver, som fordeler beskederne.

Kommunikationsserveren udvikles som et selvstændigt program, der enten kan køre direkte på smr-robotten eller på en anden computer i samme netværk.

Når kommunikationsserveren er aktiv kan "klienterne" (de program-moduler der udgør agenten) oprette en forbindelse til serveren. Når serveren modtager en ny forbindelse oprettes en "besked-kø", som tilknyttes til denne forbindelse. En klient kan sende en forspørgsel om at få en besked fra serveren. På sådan en forespørgsel fjerner serveren den ældste besked fra køen tilknyttet en forespørgende klient og sender beskeden til denne. Hvis der ingen beskeder er, sendes en standard besked, som angiver, at der ikke er nogen beskeder.

Alle beskeder som kommunikationsserveren modtager bliver kopieret ud til alle aktive besked-køer.



Figur 6.1: Skitse over kommunikationsserver hvortil tre program moduler (P1, P2 og P3) er forbundet

Dette system er skitseret på figur 6.1, hvor tre program-moduler P1, P2 og P3 er forbundet til kommunikationsserveren, som har oprettet en kø til hver.

Det at kommunikationsserveren har en kø til hvert program modul gør, at program modulerne ikke selv behøver at have en kø, og ikke hele tiden behøver at være klar til at modtage beskeder.

6.3.1 Kommunikationsserver

Kommunikationsserveren er implementeret i filen `Comcenter.cc`. Beskeder og beskedkø og listen af beskedkøer er implementeret i klasserne `TMessage`, `TOne_message_queue` og `TMessage_queues`.

6.3.2 Kommunikations klient

Klienterne (de enkelte komponenter) bruger klassen `TComClient` som opretter forbindelse til serveren, og tilbyder de to funktioner:

```
int SendMsg(TMessage *, int timeout);
TMessage *GetNextMsg(int timeout);
```

Som henholdsvis sender og modtager en besked. Som sikkerhed mod diverse problemer med overbelastet server eller ligende benyttes en *time-out*. (Agent arkitekturen er dog ikke designet til at handle efter dette).

6.4 Logiksystemet

De fem vigtigste klasser i implementeringen af logiksystemet, beskrevet i afsnit 5.1.1, er `TPredicate`, `TRule`, `TFactBase`, `TRuleBase` `TGroundSolutions`. Beskrivelsen af implementeringen tager udgangs puntk i beskrivelse af disse klasser.

6.4.1 `TPredicate` klassen

`TPredicate` bruges til at repræsentere et prædikat.

```
class TPredicate {
  public:
    /// predicate name
    string name;

    /// list of arguments
    Dynamic_array<TTerm *> arguments;

    // eval function
    typedef void* (*TFuncPtr)(void*, void*);
    TFuncPtr function_eval;
...
```

`name` angiver navnet på prædikatet og `arguments` er en liste af argumenter (termer). `function_eval` er en pointer til en funktion af typen `void f(void, void)`. `function_eval` benyttes kun når prædikatet er en del af en regels krop (se afsnit 2.1.1), og afgør hvordan prædikatet skal evalueres. Hvis `function_eval` **ikke** peger på en funktion (= `NULL`) er prædikatet et "almindeligt" prædikat, som kan evalueres med en bevisprocedure. Hvis der peges på en funktion er det denne funktion som evaluerer prædikatet.

Denne mulighed for at sammenknytte prædikater og C++ funktioner gør, at der kan benyttes diverse special prædikater. Eksemplevis kan haves et prædikat:

$$max\_pritet(X, Y)$$

Som angiver at X er større end Y.

Dette beskrives yderligere under `Eval` og `EvalF` funktionerne.

6.4.2 `TRule`, `TFactBase` og `TRuleBase` klasserne

`TRule` repræsenterer en regel som ligning 2.1:


```
class TRule {
  public:
    TPredicate *head;
    Dynamic_array<TPredicate *> body;
...
```

Hvor `head` er reglens hoved og `body` er en liste af prædikater, som udgør kroppen.

`TRuleBase` indeholder en liste af regler:

```
class TRuleBase {
 public:
   Dynamic_array<TRule *> rules;

   void AddFunctionEval(string pat, TFuncPtr EvalFunc);
  ...
```

Ud over listen af regler indeholder klassen funktionen `AddFunctionEval`. Funktionen finder alle prædikater, som indgår i kroppen på en af reglerne i `rules` og som har et navn der starter med `pat`. Alle disse prædikater sættes til at pege på funktionen `EvalFunc` som deres evauleringsfunktion.

Dette benyttes til at en mængde af prædikater kan tildeles den samme evalueringsfunktion. Eksemplesvis kunne `pat` være `'B:'` for at sætte alle prædikater der starter med B: til at blive evalueret af en funktion i Tro-komponenten.

`TFactBase` indeholder en liste, `facts`, af atomiske udsagn på grundform:

```
class TFactBase {
   private:
     Dynamic_array<TPredicate *> facts;
     Dynamic_array<string > keyed_predicates;

   public:
     void Add_key_field_predicate(string n);
     int IsKeyedPredicate(TPredicate *p);
     void AddFact(TPredicate *pred);
     TGroundSolutions *Eval(TPredicate *pred);
     TGroundSolutions *EvalF(TPredicate *p,
                             TGroundSolutions *sol);
     TPredicate **EvalRule(TRule *rule, int &count);
...
```

For at holde styr på hvilke prædikater der har nøglefelter haves en liste, `keyed_predicates`, som indeholder de prædikater, som har et nøglefelt.

Funktionen `AddFact` tilføjer et nyt udsagn til faktabasen, og kontrollere om det tilføjede udsagn er i `keyed_predicates` listen. Hvis det er tilfældet fjernes det eventuelle "gamle" udsagn.

Funktionerne `Eval`, `EvalF` og `EvalRule` bruges til at evaluere en regl udfra udsagn i faktabasen. Disse funktioner er beskrevet i afsnit 6.4.5.

Alle tre klasser indeholder desuden funktioner til pæn udskrift til skærm eller logfil, og funktioner til at læse fra filer, så f.eks regler kan læses fra en fil.

6.4.3 `TGroundSolutions` klassen

`TGroundSolutions` repræsenterer en liste af løsninger, i form af mulige variable tildelinger. Eksempelvis kunne en `TGroundSolutions` være:

$$(X, Y) = (a, b), (a, c)$$

hvilket betyder, at der findes to løsninger $(X = a, Y = b)$ og $(X = a, Y = c)$ Dette er implementeret ved at have en liste af varable (X, Y i ovenstående tilfælde) og en liste af løsninger, hvor en løsning er en liste af konstanter.


```
class TGSol {
   public:
     Dynamic_array< char * > c;
 ...
 };

class TGroundSolutions {
   public:
     Dynamic_array< char * > vars;
     Dynamic_array < TGSol * > solutions;
...
```

Der findes to specielle tilfælde af `TGroundSolutions`. Den første er løsnings listen:

$$() = () \tag{6.1}$$

Som betyder at løsningen ikke binder nogle variable, og løsningen er derfor altid *sand*. Den anden er løsningslisten:

$$(X) = () \tag{6.2}$$

Som betyder, at der ikke findes en løsning. (Der findes ingen variabel tildeling til X, der gør udtrykket sandt).

6.4.4 `AddSolutions` funktionen

Det viser sig i det følgende, at det er nødvendigt at addere to løsningslister. Det gøres med funktionen `AddSolutions`:

```
TGroundSolutions *AddSolutions(TGroundSolutions *s1,
                               TGroundSolutions *s2)
```

Ved addition af løsninger forstås de løsninger som opfylder begge de to oprindelige løsninger. Hvis en løsning er:

$$(X, Y) = (a, b), (a, c) \tag{6.3}$$

og en anden løsning er

$$(X, Z) = (a, e), (a, f) \tag{6.4}$$

Er den samlede løsning:

$$(X, Y, Z) = (a, b, e), (a, b, f), (a, c, e), (a, c, f) \tag{6.5}$$

adderes nu yderligere en løsning, som begrænser Y til b:

$$(Y) = (b) \tag{6.6}$$

bliver den samlede løsning:

$$(X, Y, Z) = (a, b, e), (a, b, f) \tag{6.7}$$

Efter som (a, c, e) og (a, c, f) ikke er oplyldt af 6.6.

Funktionen fungerer ved først at generere alle kombinationer af løsninger uden at tage hensyn til at samme variable kan indgå i begge ligninger. Det giver for 6.3 og 6.4:

$$(X, Y, X', Z) = (a, b, a, e), (a, b, a, f), (a, c, a, e), (a, c, a, f) \tag{6.8}$$

De to variable X og X' er reelt den samme variable men de stammer fra hver sin løsning. Det betyder, at alle løsninger hvor $X \neq X'$ fjernes fra løsnings listen. Der er dog ingen i dette eksempel. Så X' kan fjernes fra løsningslisten hvorefter løsningen bliver 6.5. Når 6.6 adderes bliver kombinationen af alle løsningerne:

$$(X, Y, Z, Y') = (a, b, e, b), (a, b, f, b), (a, c, e, b), (a, c, f, b) \tag{6.9}$$

Hvor Y' stammer fra 6.6. I det to sidste løsninger er $Y \neq Y'$, så de fjernes:

$$(X, Y, Z, Y') = (a, b, e, b), (a, b, f, b) \tag{6.10}$$

Hvor efter Y' kan fjernes og løsningen bliver 6.7

6.4.5 `Eval` og `EvalF` funktionerne

De tre funktioner `Eval` og `EvalF` tilhører klassen `TFactBase` og bruges til at evaluere prædikater og regler.

`Eval(p)` og `EvalF(p)` beregner en løsnings liste (`TGroundSolutions`) som gør `p` gyldig i faktabasen. Eksempelvis haves faktabasen F:

$$F = \{p(a), p(b)\} \tag{6.11}$$

Resultatet af `Eval( p(X) )` bliver:

$$(X) = (a), (b) \tag{6.12}$$

Da $p(X)$ er opfyldt for $X = a$ og $X = b$. Resultatet af `Eval( p(a) )` er:

$$() = () \tag{6.13}$$

Som betyder er $p(a)$ er et gyldigt udsagn i faktabasen, uden nogen binding af variable. Resultatet af `Eval( q(X) )` bliver:

$$(X) = () \tag{6.14}$$

Da der ikke findes en binding af X som gør $q(X)$ gyldigt.

`EvalF` bruges når et prædikat er tilknyttet en C++ funktion. `EvalF` returnerer blot reslutatet fra den tilknyttede evalueringsfunktion.

`Eval` finder løsningerne ved at søge gennem udsagnene i faktabasen. Først udvælges de udsagn hvor navnet på prædikatet er ens. Herefter sammenlignes udsagnenes termer. Hvis der søges en løsning til prædikatet:

$$p(t_1, t_2, \ldots, t_n) \tag{6.15}$$

og der fra faktabasen er udvalgt udsagnet (Bemærk at alle udsagn fra faktabasen er på grundform og alle termer er derfor konstanter):

$$p(c_1, c_2, \ldots, c_n) \tag{6.16}$$

sammenlignes termerne som:

$$\begin{aligned} t_1 &= c_1 \\ t_2 &= c_2 \\ &\vdots \\ t_n &= c_n \end{aligned}$$

Hvis der for en af ovenstående gælder at $t_i \neq c_i$ og t_i er en kostant er 6.16 ikke en instans af 6.15. Er dette ikke tilfældet er 6.15 gyldigt i faktabasen for den variable tildeling der er bestemt af ovenstående ligninger $t_i = c_1$, hvor t_i er en variable.

6.4.6 `EvalRule` funktionen

`EvalRule` evaluerer en regel i faktabasen. Hvert udsagn i reglens krop evalueres med `Eval` eller `EvalF`, og løsningslisterne fra alle udsagnene adderes med `AddSolutions` funktionen. Den variable tildeling der kommer ud af summen påtrykkes udsagnet i reglens hoved, hvilket giver en række atomiske udsagn, som kan udledes fra faktabasen med reglen.

6.5 Implementering af den reaktive komponent

Den reaktive komponent består stortset kun af de algoritmer som beskevet i kapitel 3. Komponenten er dog speciel på den måde at den er implementeret til at hente sin beskedder fra *standard-in* kanalden. Dette gør at man

kan eksperimenter med robotten, ved at skrive kommandoer direkte i komponenten fra en terminal. Dette er en store fordel når for eksempel robotten skal kalibreres.

For at forbinde den reaktive komponent til kommunikationserveren, benyttes et ekstra program `reaktiv_control`, som starter den reaktive komponent op som et *child-program*. Dette gøres med system kaldene: `fork()` og `execve(...)`. For at kommunikere med den reaktive komponent omdiregeres standard-in og standard-out kanalerne til to nye *fildescriptor*, som `reaktiv_control` kan læse/skrive til og fra. Herefter vidresender `reaktiv_control` blot al kommunikaiton melle kommunikaitonsserveren og den reaktive komponent.

6.5.1 En "stop" kommando

Det har vist sig at være vigtigt med en "stop" kommando til den reaktive komponent, som omgående stopper robotten. Dette giver dog det problem at mens robotten kører, skal odometrien, som tidligere beskrevet, beregnes med meget små intervaller. Derfor kan ikke udmidelbart hentes en komando fra *standard-in*, da dette vil blokere den øvrige aktivitet i robotten, så længe der ikke er en ny kommando. Dette løses ved at ændre *standard-in* til `non-blocking`. Men når den reaktive komponent ikke kører (venter på en ny komando) skal *standard-in* vente på en ny komando. Dette løses ved at have to sæt argumneter til *standard-in* fildescriptoren:

```
stdin_blocking_args = fcntl(0, F_GETFL);
stdin_nonblocking_args = stdin_blocking_args | O_NONBLOCK;
```

standard-in kan nu skifte i mellem `blocking` og `nonblocking` med:

`cntl(0, F_SETFL, stdin_blocking_args);`

eller `cntl(0, F_SETFL, stdin_nonblocking_args);`

Koden til den reaktiv komponent kan ses i filen `TReaktiv.cc`

6.6 Tro-komponenten

Tro-kompoeneten er implementeret i klassen `TBelief`. Informationer om observationer, placering af objekter (andre angeter) og listen over felter,

som verden er opdelt i, gemmes i klasserne: `TObsavation`, `TWObject` og `TField`.

Det mest interssante i tro-komponenten er rute planlægning:

6.6.1 Ruteplanlægning

Ud fra informationer om hvad der er placeret på et felt beregnes en *gennemtrængelighedskonstant*, $k_p{}^{as}$, for hvert felt. Som udgangspunkt sættes alle felter til $k_p{}^{as} = 0$, som angiver at robotten uhindret kan køre i gennem dette felt. Alle felter som indeholder en observation, en anden agent, eller er reserveret sættes til $k_p{}^{as} = \infty$, som angiver, at der ikke kan køres gennem feltet. Hvis en observations unøjagtighed er så stor, at observationen kan ligge i flere felter markeres alle felter med $k_p{}^{as} = \infty$ (dette sker specielt når observationerne ligger tæt på grænsen mellem to felter).

Selv om der bliver taget højde for unøjagtighed for observationer og egen position, er det stadig bedre, at robotten kører så langt uden om andre objekter som muligt. Derfor sættes alle felter som er nabo til et felt, hvor $k_p{}^{as} = \infty$ til $k_p{}^{as} = c$, hvor c evt. kan vælges som kvadratroden af antallet af felter.

Til at bestemme en rute fra robottens aktuelle position til en vilkårlig position benyttes *Dijkstra's* algoritme. *Dijkstra's* algoritme beregner den korteste rute i en graf fra et udangspunkt til alle knuder. Algoritmen er beskrevet i [4]. Søgegrafens knuder består af felterne i agentens verden. Hver knude (felt) er forbundet til fire naboer; nord, syd, øst og vest. Som pris for at køre fra et felt, i, til andet felt, j, bruges:

$$\frac{k_p{}^{os,i} + k_p{}^{os,j}}{2} + 1 \tag{6.17}$$

En rute fra A til B er nu bestemt af en liste af felter, hvor feltet A er det første felt og B er det sidste.

Ruten bestemt med *Dijkstra's* algoritme er den korteste, men den tager ikke højde for at rute kan gå på "skrå". Derfor udglattes funktionen, så der fås en rute med så få kursændringer som muligt. Til dette bruges funktionen `AjustPath`. Funktionen kigger på to punkter i ruten. Hvis der mellem disse to punkter findes en retlinie, og afstanden fra denne linie til nærmeste observation er større end k, ændres ruten til at følge denne linie. Som k

anvendes størrelsen på et felt. Funktione startet med at ses på rutens start og slut punkt, hvis der ikke kan laves en rute optimering her ses på de næste sidste punkter i ruten. Dette fortsættes indtil hele ruten er gennemgået.

6.6.2 Et monitor vindue

For at kunne følge med i hvad der sker inde i agenten er tro-kompoenten udstyret med et grafisk vindue, som tegner agentens indeling af verden i felt. Observationer tegnes som en prik, og agenter som en kasse. I alle felter som er optaget sættet et kryds. Et eksempel på vinduets indhold kan ses i næste kapitel.

Kapitel 7

Demonstration og afprøvning

I dette skulle have været en demonstration af agent-arkitekturen, implementeret på smr-robotterne. Men dette har ikke været muligt på grund af tekniske proglemer med robotterne hos IAU. Kun software modulet, beskrevet i 3.2, er blevet afprøvet.

Det ville så være oplagt at demonstrer arkitekturen med en passende simulator. Dette har bare ikke tidsmæssigt været muligt efter som hele implementeringen er lavet direkte til robotterne.

Dette kapitel indeholder derfor kun små demonstrationer af enkelte del af implementeringen.

7.1 Logiksystemet

Logiksystemet er beskrevet med eksempler i afsnit 5.1.1. Her er vist et mindre eksempel hvor alt *output* stammer fra implementeringen af logiksystemet.

Som udgangspunkt haves en faktabase:

```
FactBase:
```



```
p(a, b, a)
p(c, d, a)
p(a, c, b)
q(a, b, c)
```

og en regel

```
r(X, Y, Z) <- p(a, Y, X), q(X, Y, Z)
```

Når reglen evalueres, evalueres de to udsagn i korppen først hver for sig:

```
Query: p(a, Y, X)
Solutions:
(Y, X) = (b, a) | (c, b)

Query: q(X, Y, Z)
Solutions:
(X, Y, Z) = (a, b, t)
```

Herefter adderes de to løsninger:

```
sum solutions:
(Y, X, X, Y, Z) = (b, a, a, b, t)
```

Denne resluterne variabeltildeling er den som gør reglens krop gyldig. Når denne variabeltildeling påtrykke reglens hoved føs:

```
r(a, b, t)
```

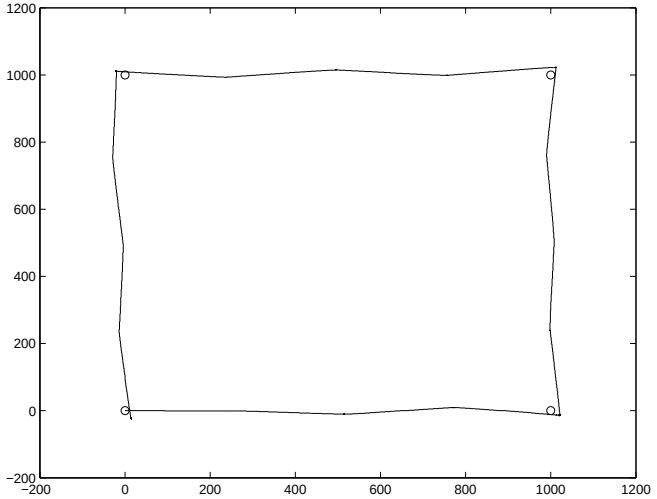
Som så kan adderes til faktabasen:

```
FactBase:
p(a, b, a)
p(c, d, a)
p(a, c, b)
q(a, b, c)
r(a, b, t)
```

7.2 Reaktive komponent og robot algoritmerne

Den rekative komponent indeholder det software modul som er udvilket til at styre robotten. Modulet er afprøvet og det virker. Når robotten sættes

til en køre rundt i en firkant, fås et rute plot som på figur 7.1. Figuren viser robottens beregnet position, og ikke den faktiske position. Det kan dog ses på figure at robotten køre nogenlunde lige ud og drejer rimelig præcist.

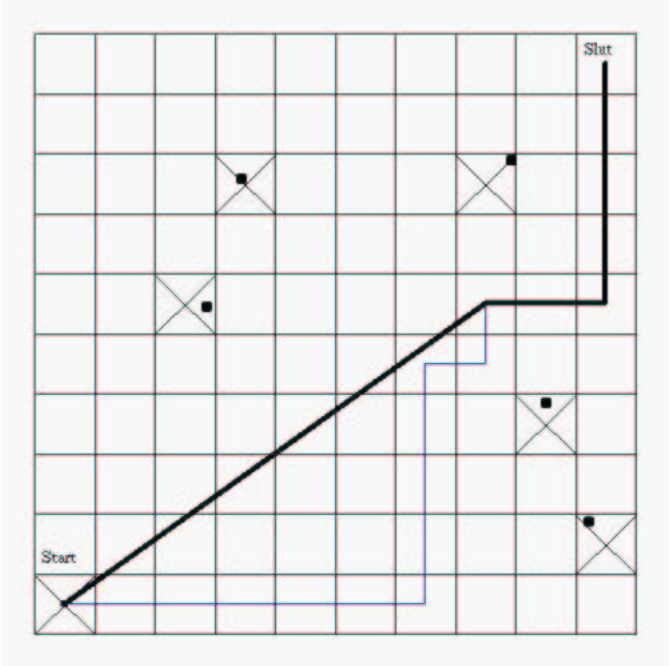


Figur 7.1: Path trace for en smr-robot. Plottet viser **beregnet** position, ikke robottens faktiske position. 1.000mm

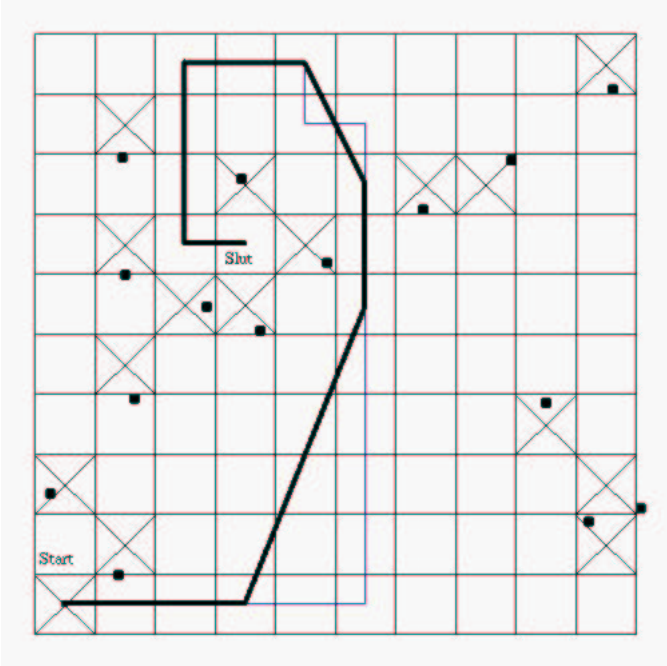
Hvad der ikke kan ses på figuren er hvor gode de odometriske beregninger er. For at få en ide om det, blev robotten sat til at gennem kører ruten (firkanten) 10 gange. Det vil sigen en tur på 40 meter. Efter de to omgange var forskellen på robotens faktiske position og den beregnede position omkring 20 cm. Det betyder en forkskel på omkring 0.5 cm per kørt meter. Det kan dog kun bruges som en "grov regel" da fejlen også afhænger meget af hvormeget der drejes.

7.2.1 Rute planlægning

Tro-komponenten har en monitor vindue, der viser hvilke observationer den har fortaget, og hvilke rute den vælger. På figur 7.2 og 7.3 vises en rute fundet med algoritmen beskrevet i afsnit 5.3. Den tynde streg angiver ruten fundet med graf-søgningsalgoritmen og den tykke streg angiver den endelige rute efter udglatning. "start" og "slut" mærkningerne er sat på "manuelt".



Figur 7.2: Roboten har fundet en rute fra punktet "start" til punktet "slut".



Figur 7.3: Roboten har fundet en rute fra punktet "start" til punktet "slut".

Kapitel 8

Konklusion

Efter som afprøvning af agent-arkitekturen ikke har været mulig er det heller ikke muligt at diskutere arkitekturen ud fra hvordan den fungerer i praksis.

En afprøvning af agenten ville formentlig tydelig have vist en række punkter, hvor agenten er designet uhensigtsmæssigt. Uden afprøvning er det dog alligevel muligt at en pege, på række punkter som vil kunne forbedres.

Reservationen af felter er meget simpel. En mere fornuftig løsning ville være at lade agenterne lave en reel forhandling om felterne baseret på, hvor vigtigt det er for den enkelte agent at bruge disse felter.

Den generelle kommunikation mellem agenterne kunne også være lavet mere kompliceret, så agenterne evt. kunne planlægge i fællesskab, og forhandle om opgaverne.

8.1 Anvendelse til realistiske opgaver

Som det tidligere i rapporten er konkluderet, er det meget begrænset hvad forsøgsrobotten kan bruges til af reelle opgaver. Men hvis agent-arkitekturen blev benyttet på en robot, som faktisk kunne vaske gulv, ville det så virke?

Det ville ihvertfald kræve, at forhindringer observeres med stor sikkerhed. Desuden ville det være nødvendigt med et positionsbestemmelsessystem, som enten ikke akkumulerer fejl, eller som det er muligt at opdatere.

Spørgsmålet er om agent-arkitekturer generalt er anvendelig på robotter, eller om traditionelle *robot-arkitekturer* er bedre. Der er naturligvis ikke noget entydigt svar. Ved robot-tekniske ting som styring af motorer, optegning af kort og ruteplanlægning vinder man ikke udmiddelbart nogen fordel ved at bruge en agent-arkitektur. Men ligeså snart robotter skal samarbejde på en ikke fast planlagt måde, vil der være stor fordel ved anvendelse af agent-teknologi. Det virker derfor som om, at ved en agent/robot løsning bør så meget som muligt løses med velkendte robot-teknikker, så agent-arkitekturen bliver så overordnet som mulig.

8.2 Arkitektur-modellen

Tilgengæld for den manglende afprøvning er lagt mere vægt på implementer et logiksystem til agenterne. Det er lykkedes med en data-dreven bevisalgoritme.

En oplagt mulighed for at udvide logiksystemet er at implementere en sammenkædning til prolog. Dette vil dog kræve at platformen for implementeringen kan afvikle et prologsystem.

Der er fundet en fornuftig agent-arkitektur-model baseret på komponenter. Modellen giver vide muligheder for implementation af agenter, med nem adgang til kommunikation mellem komponenterne. Modellen kunne udvides til et system, hvor alle agentens komponenter specificeres med en passende syntaks, som beskriver regler og beskeder for hver komponent. Dette ville give en mulighed for at lave forsøg med agent-design uden at skulle programmere komponenterne. Der er ikke langt fra den aktuelle implementering til sådan et system.

I arkitekturen der er beskrevet i denne rapport, er det kun målkomponenten der er implementeret med logiksystem. Men både intentions- og kommunikation- komponenterne kunne også være designet med dette system. Tro-komponenten kunne ligeledes være opbygget omkring logiksystemet, men der er dog en del funktionalitet som nemmere beskrives med "sædvanlige" programmeringssprog. reaktive

Litteratur

[1] J. Borenstein, H. R. Everett, and L. Feng. Where am i? - sensors and methods for mobil robot positioning, 1996.

[2] Sean H. Breheny. The sharp gp2d02 ir distance measureing sensor.

[3] Marco Colombetti. Different ways to have somthing in common. In *Third International Conference: Flexible Query Answering Systems*. Springer, May 1998.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT Press, 1990.

[5] Gregory Dudek and Michaael Jenkin. *Computional Principles of Mobile Robotics*. Cambridge University Press, 2000.

[6] Jørgen Fischer Nilsson. Data logic - a gentle introduction to... Department of Information - Technology, Technical University of Denmark, 1998.

[7] Simon Parsons, Carkes Sierra, and Nick Jennings. Agents that reason and negotiate by arguing. In *Journal of Logic and Computation*, volume 8. Oxford University Press, June 1998. Special Issue: Computational and Logical Aspects af Multiagent Systems.

[8] David Poole, Alan Mackworth, and Randy Goeble. *Computional Intelligence - a logical approach*. Oxford University Press, 1998.

[9] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[10] J. Sabater, C. Sierra, S. Parsons, and N. Jennings. Engineering executable agents using multi-context systems, 1999.

[11] Yoav Shoham. *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann Publishers, Inc., Kap. 8.

[12] Munindar P. Singh. Agent communication languages: Rethinking the principles. *Computer*, 0018-9162:40–47, 1998.

[13] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Bilag A

Konfigurationsfilen

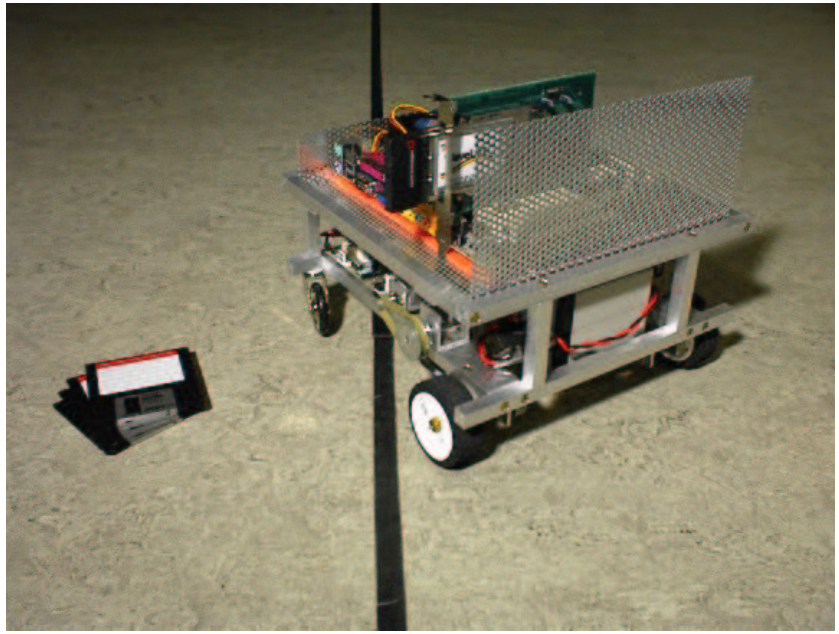
```
% Address of the communication server
comserver = 130.225.76.177
%comserver = 130.225.77.25
%comserver =  192.38.66.14

%Comunaction server max connections
max_sockets = 20

%Communication port
port = 10956

% size of monitor window in pixels largest edge
win_size = 500

% The size of the world in mm.
world_x = 10000
world_y = 10000

% resulution of the "grid"
grid_rows = 15
grid_cols = 15

% The speed of the robot
```



```
movespeed = 20
turnspeed = 5

% IR-sensor callibrating
sensor1_k1 = 1000
sensor1_k2 = 1.9
sensor1_k3 = 25

sensor2_k1 = 1000
sensor2_k2 = 1.9
sensor2_k3 = 25

sensor3_k1 = 1000
sensor3_k2 = 1.9
sensor3_k3 = 25

sensor4_k1 = 1000
sensor4_k2 = 1.9
sensor4_k3 = 25


% The name and position of this agent (in mm)
myname = sim_smr
start_pos_x = 500
start_pos_y = 500
start_pos_theta = 0

% How often..(in millisec.)

% ..send robot position to comcenter
send_pos_interval = 1000

% ..check IR sensors
IR_check_interval = 1005

% ..write robot position to file
Pathtrace_interval = 1009

% End of config file...
```

Bilag B

Billed af smr-robot



Figur B.1: Billed af en smr-robot

Bilag C

Kildekode

Bilag C er vedlagt rapporten i en seperat bind.