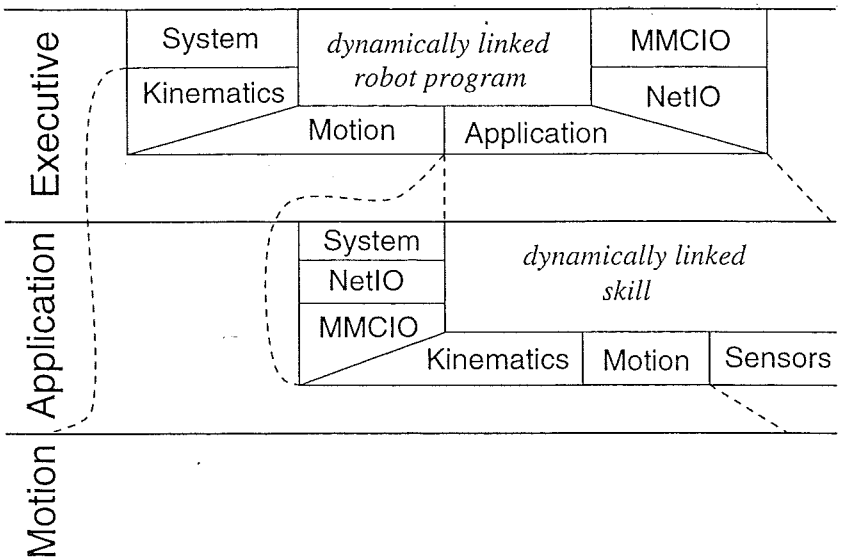
ISSN 0280-5316
ISRN LUTFD2/TRFT- -5557- -SE

Ett öppet system för programmering och styrning av robotar

Jan Andersson



Institutionen för Reglerteknik
Lunds Tekniska Högskola
Maj 1996

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 1996

Document Number
ISRN LUTFD2/TFRT--5557--SE

Author(s)
Jan Andersson

Supervisor
Klas Nilsson, Rolf Johansson

Sponsoring organisation
NUTEK - Swedish National Board for Industrial and Technical Development

Title and subtitle
Ett öppet system för programmering och styrning av robotar
(An open system for robot programming and control)

Abstract

Industrial robots are used/configured/programmed by several different types of users/programmers. The control system should preferably expose a uniform view for each programming situation. Such views were defined by the Open Robot Control (ORC) architecture, developed within the department in an earlier project.

This project refines and evaluates some robot programming principles. First, earlier prototype implementations are merged together to a more complete system. Secondly, the low-level control is verified for an available ABB IRB-6 robot. Third, the system is improved to allow robot programming languages and interpretors to be exchanged at run-time. It is also possible to introduce new control loops based on external sensors. The goal is to let the robot perform contour following, programmed in a user friendly way utilizing the developed features.

Key words
Industrial robots, Robot programming, Robot control, Open systems, Embedded systems

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
Swedish

Number of pages
48

Recipient's notes

Security classification

Förord

Denna rapport beskriver ett examenarbete som utförts vid institutionen för reglerteknik vid Lunds Tekniska Högskola. Arbetet har inneburit vidareutveckling och utprovning av realtidsprogramvaran till det experimentella robotsystem som finns i robotlabbet vid institutionen.

En förutsättning för detta examensarbetes genomförande var den tidigare utvecklade programvaran som fanns tillgänglig. Mjukvarustrukturen som använts heter *Open Robot Control* och är utvecklad av Klas Nilsson. Programvaran som sköter dynamisk bindning[1] av kod är utvecklad av Klas Nilsson, Anders Blomdell och Olof Laurin. Utan dessa personers förberedande arbete hade jag inte uppnått de resultat som redovisas i denna rapport.

Innan jag började mitt examensarbete på institutionen för reglerteknik hade jag aldrig funderat över dynamiskt bunden kod. Under min tid här har jag blivit väl insatt i problematiken med detta, vilka tillämpningar och vilken potential system med möjlighet att dynamiskt binda kod har. Det har varit väldigt intressant och lärorikt.

Jag vill här passa på att framföra ett tack till min handledare Klas Nilsson för hans uthållighet och aktiva vägledning av arbetet trots att han själv hade sin egen doktorsavhandling att skriva. Jag vill även tacka Jonas Lindholm som tillsammans med mig inom ramen för ett projekt i kursen realtidssystem implementerat ett exempel på sensorbaserad höjdreglering av roboten. Dessutom vill jag tacka Bart Hendriks, som arbetat med sitt eget examensarbetet i robotlabbet, för all hjälp och alla tips. Till sist vill jag även tacka Olof Laurin som hjälpt mig att korrekturläsa rapporten.

1 Dynamisk bindning är en metod för att under drift binda exekverbar kod till ett exekverande system.

Innehåll

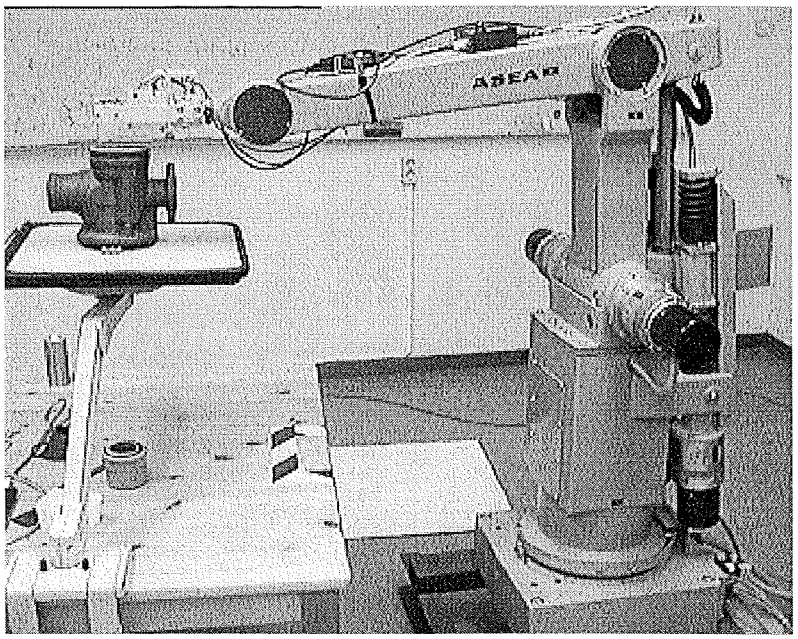
1. Inledning

Vid institutionen för reglerteknik, LTH bedrivs forskning inom realtidsystem och robotstyrning. Målsättningen är att utveckla flexiblare robotsystem. För att åstadkomma detta är en skiktad mjukvarustruktur lämplig. En av idéerna är att göra det möjligt att byta ut delar av mjukvaran under drift för att på så vis anpassa roboten för olika tillämpningar. En annan idé är att använda sensorbaserad rörelsestyrning för att öka robotens förmåga att anpassa sig till variationer i dess omgivning.

Vid institutionen har utvecklats en mjukvarustruktur, Open Robot Control arkitekturen [7], som i fortsättningen kommer att förkortas ORC. ORC är uppbyggd av olika skikt. Mellan varje skikt finns ett väldefinierat gränssnitt som gör det enkelt att byta ut ett skikt utan att övriga koden behöver ändras. ORC är speciellt utvecklad för robot- och reglertillämpningar.

Vid institutionen har även utvecklats en metod för att under drift binda exekverbar kod till ett exekverande system. Detta kommer i fortsättningen benämnas dynamisk bindning. Dynamisk bindning gör det möjligt att ändra systemets färdigheter[1] (eng: skill) utan att behöva starta om hela systemet; det räcker med att dynamiskt binda den nya koden till det exekverande systemet.

För sensorbaserad rörelsestyrning utnyttjas i detta fall en givare monterad på robotens gripdon. För att minimera tidsfördröjning och därmed fasförskjutning som uppkommer då givarsignalen återkopplas in i reglerloopen, så krävs en tät koppling mellan de olika delarna av realtidsprogramvaran.



Figur 1.1 Roboten ASEA IRB6/2 som använts.

Syftet med detta examensarbetet har varit att sammanställa, tillämpa och vidareutveckla resultatet av flera tidigare vitt skilda projekt, samt att utveckla

[1] En färdighet hos systemet är något som systemet kan utföra. Exempelvis en robot med ett gripdon, samt lämplig programvara för att styra arm och fingrar, har färdigheten att kunna gripa saker.

sensorbaserad robotstyrning, utveckla ett gränssnitt som använder dynamisk bindning och implementera ett grafiskt användargränssnitt. Eftersom Modula-2 ger ett visst stöd för realtidstillämpningar, och på grund av att befintlig programvara till största delen var skriven i Modula-2, valde jag att huvudsakligen använda detta programspråk.

Denna rapport har följande uppläggning:

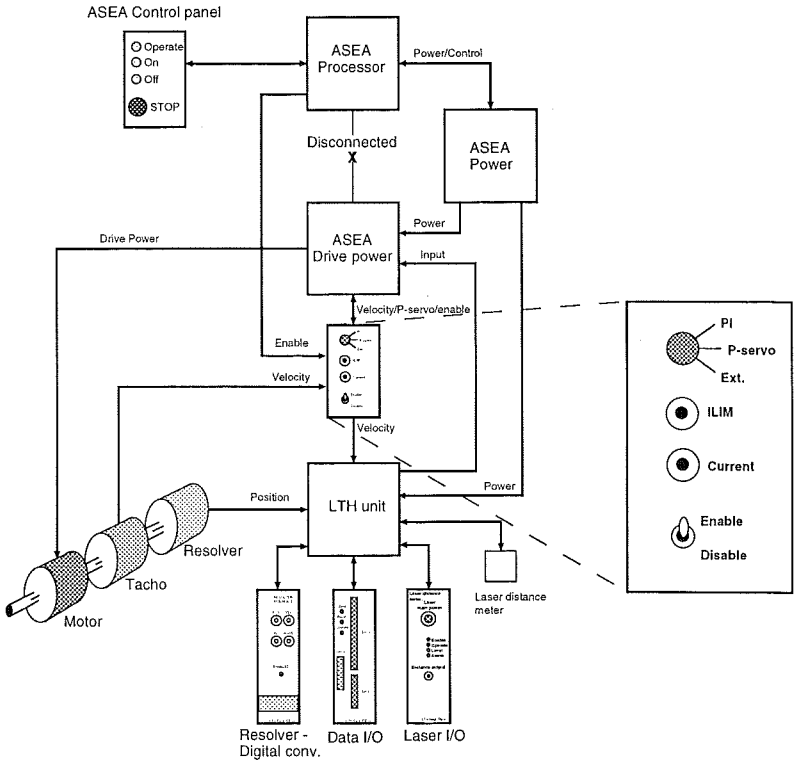
- Kapitel 2, **Bakgrund**, beskriver systemet i stort.
- Kapitel 3, **Sammanställning av befintliga system**, beskriver hur de olika delarna av systemet som sammanfogats fungerar.
- Kapitel 4, **Kontrollpanelen**, beskriver vad kontrollpanelen innehåller och hur den fungerar.
- Kapitel 5, **Dynamisk bindning**, beskriver gränssnittet som används för dynamisk medlänkning av exekverbar kod av såväl robotprogram som färdigheter hos systemet.
- Kapitel 6, **Sensorbaserade rörelser**, beskriver hur sensorbaserade rörelser kan användas samt vilka möjligheter och begränsningar som finns.
- Kapitel 7, **Exempel**, beskriver hur de delar av systemet jag implementerat kan användas.
- Kapitel 8, **Möjliga vidareutvecklingar**, beskriver de vidareutvecklingar som kan göras av de delar av systemet som jag utvecklat.
- Kapitel 9, **Sammanfattning och kommentarer**, beskriver förutom sammanfattning mina erfarenheter och slutsatser från arbetet.
- Appendix A, **Objektdiagram**, beskriver notationen för de objektdiagram som används i rapporten.
- Appendix B, **Matlabs grafiska användargränssnitt**, beskriver vilka möjligheter som finns i Matlabs grafiska användargränssnitt.

2. Bakgrund

I detta kapitel ges en översikt över hur systemet fungerar. Översikten är uppdelad i en hårdvarudel och en mjukvarudel. Målet med detta kapitel är att läsaren skall få inblick i hur systemet är uppbyggt för att sedan bättre kunna förstå de följande kapitlen.

2.1 Hårdvara

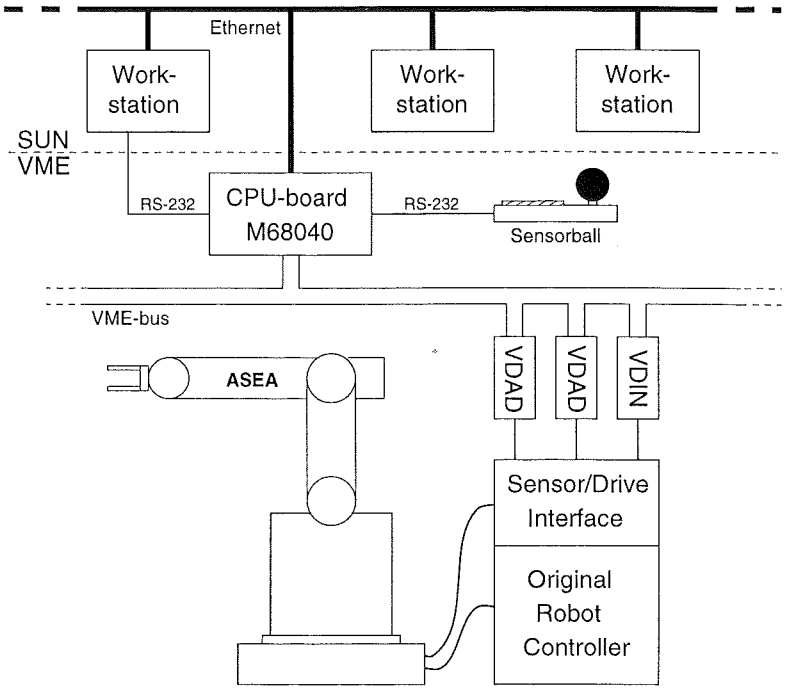
En industrirobot av typ ASEA IRB-6/2 har modifierats för laborationsanvändning. Av det ursprungliga styrsystemet utnyttjas endast kraftelektroniken och nödstoppsanordningen. Styrsignalen från den ursprungliga datorn är bortkopplad vilket öppnat systemet och gjort gör det möjligt för användaren att själv implementera hela reglersystemet. Den ursprungliga datorn har behållits endast p.g.a. att den tillhandahåller vissa säkerhetssystem. Robotens ursprungliga givare (resolvrar för mätning av motorvinklarna) används men ny mätelektronik (Resolver Digital conv. i figur 2.2) har tillfogats. Systemet är även utrustad med en lasergivare för avståndsmätning. Se figur 2.1. För mer utförlig information om den modifierade roboten hänvisar jag till *Reconfiguring an ASEA IRB-6 Robot System for Control Experiments* [1].



Figur 2.1 Den modifierade hårdvaran för ASEA robot IRB-6/2.

Datordelen av den experimentella plattform består av två fysiska nivåer. Överst finns värddator, en SUN arbetsstation. På den sker all filhantering och kompilering. Här finns även användargränssnitt med kontrollpanel och plottning. På nivån under finns ett VME-system med en M68040 processor. Mellan

dessa system finns två kommunikationskanaler. Dels en seriell förbindelse (RS-232), dels en förbindelse via ethernet. Genom den seriella förbindelsen skickas enklare kommandon (för start av systemet, etc.) och utskrifter (för felsökning). Denna typ av kommunikation används också via en annan port för överföring av data från styrboll (sexdimensionell styrspak). Via ethernet skickas all övrig information. Det kan vara programkod, plotvärden, signaler från kontrollpanelen etc. Figur 2.2 ger en översikt av hårdvaran i systemet.



Figur 2.2 Experimentplattformen som använts.

2.2 Mjukvara

Istället för den ursprungliga del av hårdvara som kopplats bort vid modifieringen av robotsystemet finns en stor mängd mjukvara implementerad för att styra roboten. Även mjukvaran kan delas upp i olika nivåer. Överst finns värddatorn (arbetsstation) som kör Matlab. I Matlab finns implementerat olika script för att ta emot och sända information till målsystemet. I Matlab finns även en kontrollpanel och parametergränssnitt som även dessa arbetar mot målsystemet. Från kontrollpanelen är det möjligt att styra roboten och via parametergränssnittet är det möjligt att ändra vissa parametrar hos systemet.

Till målsystemet (M68040) finns en realtidskärna utvecklad på institutionen [5]. Den tillåter samplingshastigheter upp till 1 kHz. Målsystemet (VME-systemet) är i sig uppbyggt i flera nivåer. Det finns olika processer som arbetar mot de olika scripten, kontrollpanel och parametergränssnitt på värddatorn. Det finns även en process för att generera trajektorier[1] (eng: trajectory) som

[1]Trajektorian talar om längs vilken bana roboten ska röra sig och var den ska befinna sig vid varje tidpunkt.

aktiveras när en ny rörelse ska utföras. Trajektorierna skickas sedan till en process som sköter regleringen av roboten.

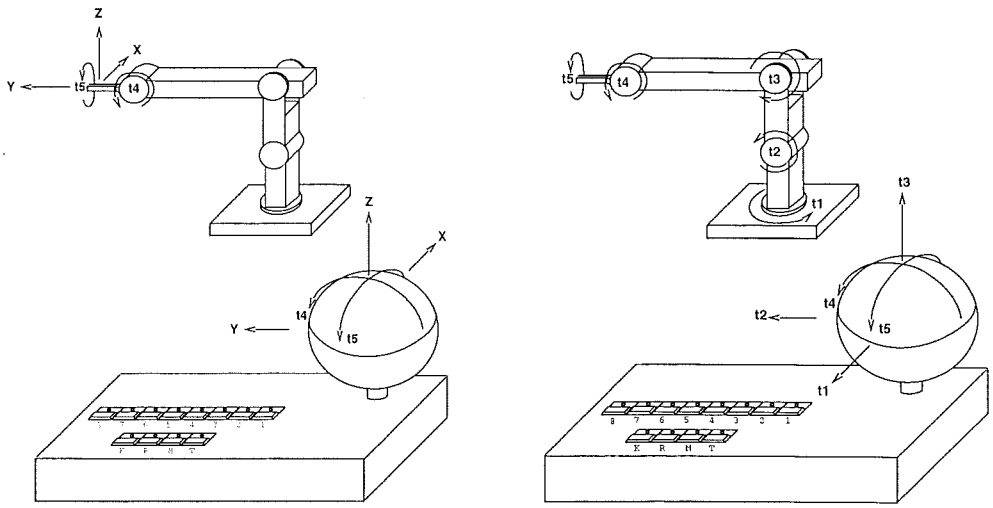
Ur reglerteknisk synvinkel ses processen som sköter regleringen av roboten som den väsentliga delen av systemet, men programvarutekniskt utgör den en liten del av styrsystemets programvara. Däremot tar processen som genererar trajektorier och de processer som sköter kommunikationen med värddatorn betydligt större utrymme. Samtidigt som processen som sköter regleringen av roboten körs måste processen för generering av trajektorier och de processer som sköter kommunikationen med värddatorn få processortid att utföra sina uppgifter. Detta är ett realtidsproblem som måste hanteras så att alla processer blir tillgodosedda för att systemet ska fungera. Programkoden till målsystemet är skriven i Modula-2. För mer information om robotmjukvaran hänvisar jag till *Implementering av experimentellt robotstyrsystem* [2] samt *Matlab som "compute server" för inbyggda system - beräkningar av robotrörelser* [11]. Dessa rapporter beskriver tidigare versioner vilka legat till grund för mina utvidgningar av systemet.

3. Sammanställning av befintliga system

I tidigare examensarbeten har robotsystemet modifierats för olika tillämpningar som robotstyrning med styrboll [2], förberäkning av trajektorier i Matlab [11] och dynamisk bindning av exekverbar kod [3, 9, 7, 8]. Dessa olika tillämpningar har dock inte tidigare integrerats till ett system. Jag tog därför som min första uppgift att sammanfoga de olika delarna till en fungerande helhet. Test och utprovning av programvaran har varit tidskrävande. Att beskriva alla problem som jag stött på är inte meningsfullt. Jag har i detta kapitel istället valt att beskriva de delar som jag sammanfogat.

3.1 Styrning med styrboll

Vid styrning med styrboll kan robotens 5 olika leder (eng: joint) enkelt styras. När styrning ska göras från styrbollen överlåts kontrollen helt och hållet till styrbollen och kan inte fås tillbaka förrän styrbollen lämnar tillbaka kontrollen till datorn. Kontrollen lämnas tillbaka till datorn genom att trycka på knappen **Mode** på styrbollen (knappen 8 i figur 3.1).



Figur 3.1 Koppling mellan styrboll och robot. Till vänster då kartesiska koordinater används och till höger då ledvinklar används [2].

I figur 3.1 visas hur roboten rör sig när styrbollen används. Styrbollen kan användas i två olika koordinatsystem, kartesiska koordinater eller ledvinkelkoordinater.

3.2 Förberäkning av trajektorier

Matlab kan användas som "compute server" för att förberäkna trajektorier som sedan ger referensvärden till robotens servoreglering. Modulen **PreComputeTraj** sköter kommunikationen med "compute servern" i Matlab. I syste-

met finns en enkel trajektoriegenerator, men för att beräkna mer avancerade trajektorier bör trajektorierna förberäknas för att inte det ska bli tidsfördröjning mellan regulatorn och trajektoriegeneratorn eftersom de delar på samma processorkapacitet. Följande funktioner finns implementerade hos "compute servern" i Matlab [11]:

- linjär rörelse med viapunkter i ledkoordinater
- linjär rörelse med viapunkter i kartesiska koordinater
- cirkulär rörelse i kartesiska koordinater
- linjärinterpolation
- sluten lösning av invers kinematik för IRB-6
- animering av robotrörelser m.h.a. 3D-funktioner

3.3 Dynamiskt bunden kod

På institutionen har utvecklats en metod för att dynamiskt binda kod till systemet. Det vill säga att det under exekvering är möjligt att länka med ytterligare exekverbar kod. Detta kan användas för att t.ex. ladda ner instruktioner till roboten eller ladda ner koden till en interpretator. Man skulle också kunna tänka sig att under drift byta ut robotens regulator mot en annan.

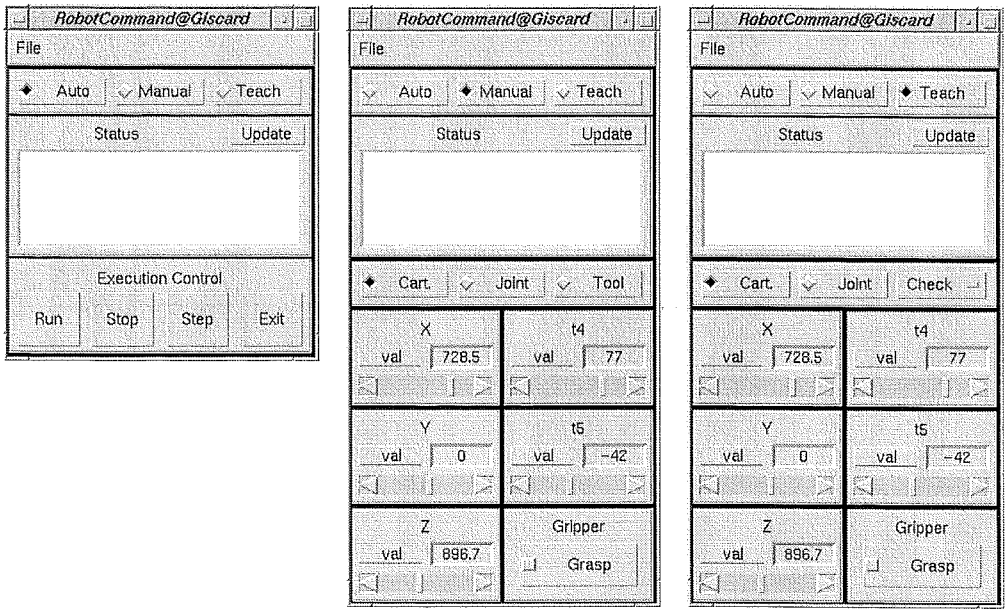
Modulen **DynamicLinker** innehåller den länkare som gör det möjlig att dynamiskt binda exekverbar kod. För mer information om dynamisk bindning hänvisar jag till [3, 9, 7, 8].

4. Kontrollpanel

Det finns tre alternativa helt olika sätt att styra roboten. Det kan ske genom att ett program med robotinstruktioner exekveras sekventiellt. Det kan även ske genom att manuellt ställa in robotens position med hjälp av kontrollpanelen. Det tredje sättet är att överlåta kontrollen till en styrboll varifrån man enkelt kan manövrera roboten. Jag beskriver i detta kapitel hur jag byggt upp och konstruerat en kontrollpanel varifrån man kan styra roboten på dessa tre sätt.

4.1 Funktionsbeskrivning

Kontrollpanelen har tre olika moder (driftstillstånd, eng: mode), en för varje sätt att styra roboten. Genom att välja mod så förändras stora delar av kontrollpanelen, till utseende eller funktion. De delar som är karakteristiska för en viss mod finns bara synliga när denna mod är aktiv, vilket kraftigt minskar komplexiteten hos kontrollpanelens utseende. De tre olika moderna har jag valt att kalla **auto**, **manual** och **teach**. Kontrollpanelens utseende när de olika moderna är aktiva visas av figur 4.1. Till vänster i figur 4.1 visas moden **auto**, i mitten moden **manual** och till höger visas hur moden **teach** ser ut. Anvädargränssnittet är uppdelat i block för att man lättare ska se vilka grafiska objekt som hör ihop. De två översta blocken ligger statiskt synliga. Det översta innehåller tre knappar för att välja mod. Det näst översta innehåller ett textfönster där korta meddelanden från systemet kan skrivas ut.



Figur 4.1 Kontrollpanelens utseende i de tre moderna. Kontrollpanelen till höger i figuren visar endast robotens position som kommenderats med hjälp av styrbollen (figur 3.1). Kontrollpanelen i mitten tillåter användaren att ändra robotens position genom att editera de visade numeriska värdena, alternativt genom att ändra rullningslisten.

Exekveringskontroll av robotprogram

Moden **Auto** innehåller funktioner för exekveringskontroll av robotprogram. Till vänster i figur 4.1 visas hur kontrollpanelen ser ut då moden **Auto** är aktiv. I exekveringskontrollen ingår fyra olika funktioner. Genom en knapptryckning är det möjligt att starta, stoppa, avsluta/omstarta eller stega exekveringen av robotprogrammet. Nedan beskrivs de fyra funktionerna.

Run: Startar exekveringen av robotprogrammet. Om systemet tidigare blivit stoppat så fortsätter exekveringen där den stannat. Om robotprogrammet redan exekverat klart händer ingenting, annars börjar första instruktionen i robotprogrammet att exekveras.

Stop: Avbryter rörelsen som roboten håller på att exekvera och roboten bromsas snabbt in. När roboten bromsats in kan exekveringen av robotprogrammet återstartas där den avbröts genom något av kommandona **run** eller **step**.

Step: Exekverar nästa robotinstruktion varefter roboten stoppas. Om robotprogrammet tidigare blivit stoppat under exekvering av en rörelseinstruktion så exekveras den avbrutna instruktionen klart varefter robotprogrammet åter stoppas. Om exekveringen ännu inte påbörjats eller om exekveringen blivit stoppad mellan två olika robotinstruktioner så exekveras nästa instruktion varefter robotprogrammet åter stoppas.

Exit: Avslutar exekveringen av robotprogrammet genast och roboten stoppas. Robotprogrammet återstartas men exekveringen startar inte förrän kommandot **run** ges.

Användning av kompilerat språk för robotprogrammering innebär vissa problem vid implementering av denna funktionalitet som normalt kräver interpretering (tolkning). Detta kommenteras sist i detta kapitel.

Manuell styrning av roboten

Moden **Manual** innehåller funktioner för att direkt styra roboten genom att via kontrollpanelen ange ny position. Moden **manual** har tre olika koordinatmoder: **Cartesian**, **Joint** och **Tool**. I samtliga koordinatmoder används ett parametergränssnitt där varje parameter (koordinat) kan ställas in antingen alfanumeriskt genom att skriva in ett numeriskt värde eller grafiskt genom att ändra positionen på en rullningslist (eng: scrollbar). Mittenbilden i figur 4.1 visar kontrollpanelens utseende när moden **manual** är aktiv.

Cartesian: Innebär att roboten styrs genom att man anger de kartesiska koordinater (x, y och z-led) som man önskar att roboten ska flytta sig till. Eftersom roboten har fem leder räcker det inte att ange tre koordinater utan man måste även ange gripdonets läge, dvs vinklarna till de två leder gripdonet har. Gripdonet kan även öppnas och slutas.

Joint: Innebär att roboten styrs genom att man anger vinklarna till robotens samtliga fem leder. Gripdonet kan öppnas och slutas.

Tool: Innebär att robotens gripdon alltid befinner sig i koordinatsystemets origo och att alla rörelser relateras till gripdonets aktuella position och vinkel. Denna funktion finns ännu inte implementerad.

Styrning med styrboll

Moden **Teach** innehåller funktioner för att styra roboten från en styrboll. När moden **Teach** blir aktiv överlåts automatiskt kontrollen till styrbollen. Från styrbollen som är speciellt utvecklad för att manövrera robotar är det sedan enkelt att styra roboten. Kontrollpanelen kan visa robotens koordinater. Det finns tre olika sätt för kontrollpanelen att göra uppdateringen av robotens aktuella positionsvärden. De tre olika sätten har jag valt att kalla **Check**, **Loop** och **Cont**.

Check: Uppdaterar robotens positionsvärden en gång.

Loop: Uppdaterar robotens positionsvärden en gång per sekund tills användaren lämnar moden **teach** från styrbollen. Detta sätt att uppdatera positionsvärdena tar stor del av Matlabs arbetstid varför den kan vara olämplig om Matlab behöver användas av andra tillämpningar samtidigt.

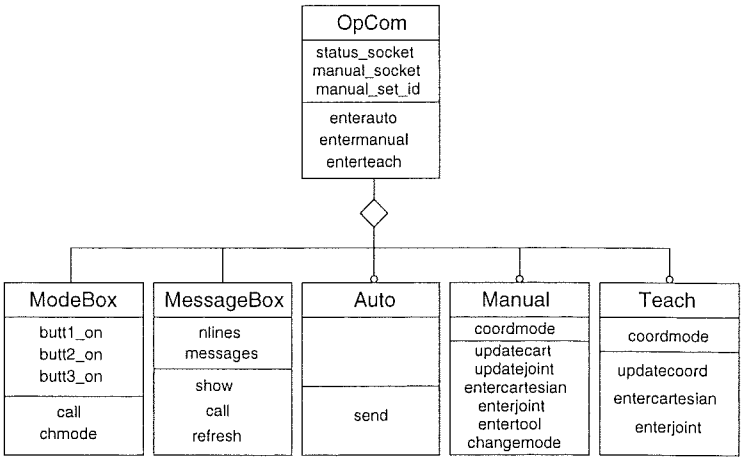
Cont: Uppdaterar robotens positionsvärden kontinuerligt tills användaren lämnar moden teach från styrbollen. Detta sätt att uppdatera positionsvärdena blockerar Matlab så att det inte kan användas av andra tillämpningar samtidigt.

Från styrbollen finns det förutom möjligheter att manövrera roboten bl.a. funktioner för att byta koordinatmod och lämna moden **teach**. När användaren ändrar koordinatmod från styrbollen ändras automatiskt de visade positionsvärdena i kontrollpanelen till rätt koordinatmod vid nästa uppdatering. Möjliga koordinatmoder är **Cartesian** och **Joint**. Till höger i figur 4.1 visas kontrollpanelens utseende när moden **teach** är aktiv.

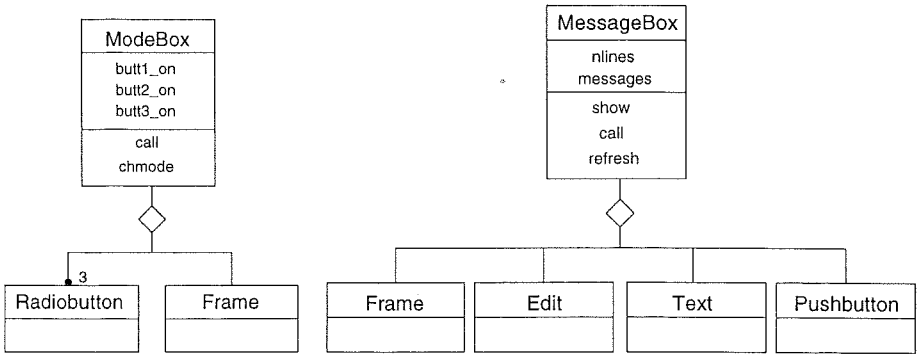
4.2 Objektorientering

Den visuella delen av kontrollpanelen är gjord i Matlab och använder de inbyggda funktionerna som finns i Matlab för att skapa grafiska användargränssnitt, se appendix B. Trots att Matlab saknar egentligt stöd för objektorientering så har jag gjort en objektorienterad uppbyggnad av det grafiska användargränssnittet. För att på ett schematiskt sätt visa den objektorienterade hierarkin har jag använt mig av en teknik som heter OMT (Object Modeling Technique) [10]. Notationen för Objektdiagram enligt OMT finns beskriven i appendix A.

Figur 4.2 visar objektdiagrammet över kontrollpanelen. Blocket **OpCom** består av ett block av typen **ModeBox** och ett av typen **MessageBox**. Det kan dessutom bestå av ett block av typerna **Auto**, **Manual** eller **Teach**. Blocket **ModeBox** används för att välja mod. Eftersom bara en mod får vara aktiv åt gången så används ömsesidig uteslutning för att knapparna inte ska vara markerade aktiva samtidigt. Detta löses genom att callback-rutinen som anropas när knappen trycks ner kollar om det är tillåtet att byta mod. Om så är fallet så avmarkeras den modknappen som tidigare var markerad aktiv innan kontrollpanelen byter mod. Om det däremot inte är tillåtet att byta mod så måste knapparna återställas till ursprungstillståndet eftersom knappen automatiskt tänds när den trycks ner. Objektdiagrammet för **ModeBox** visas till vänster i figur 4.3.



Figur 4.2 Objektdiagram för **Opcom.**



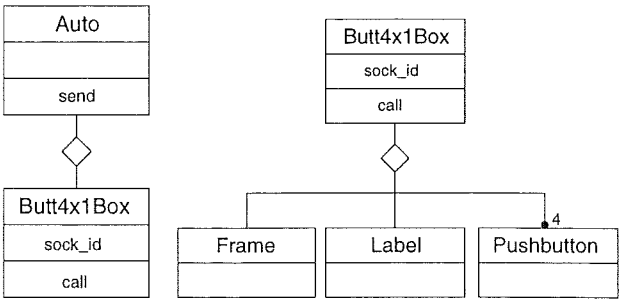
Figur 4.3 Objektdiagram för **ModeBox** och **MessageBox**

Blocket **MessageBox** har en matcomm-förbindelse[1] via ethernet med målsystemet. Om knappen **Update** trycks ner så skickas ett kommando från Matlab via ethernet till målsystemet. Matlab ställer sig sedan och väntar på svar från målsystemet. Kommandot tas emot av processen **Remote** i målsystemet och avkodas. Om det mottagna kommandot är **update** så aktiveras processen **StatusProc**. **StatusProc** skickar upp samtliga buffrade meddelanden till Matlab, som tar emot meddelandena och skriver ut dem i **MessageBox**. Objektdiagrammet för **MessageBox** visas till höger i figur 4.3.

Auto

Blocket **Auto** har också en matcomm-förbindelse via ethernet till målsystemet. Det är samma förbindelse som blocket **MessageBox** använder för att kommunicera med målsystemet. När någon av knapparna **run**, **stop**, **step** eller **exit** trycks ner så skickas ett kommando via ethernet till målsystemet. I målsystemet tas kommandot om hand av processen **Remote** som avkodar kommandot och utför operationen i robotens styrsystem. Det skickas även ett meddelande via en brevlåda till processen **StatusProc** som skickar upp meddelandet till Matlab som skriver ut det i blocket **MessageBox**. Blocket **Auto** består av bara en instans av blocket **Butt4x1Box**. Objektdiagrammen för blocken **Auto** och **Butt4x1Box** finns i figur 4.4.

[1]MatComm är ett programpaket som sköter kommunikationen mellan Matlab och målsystemet via ethernet.

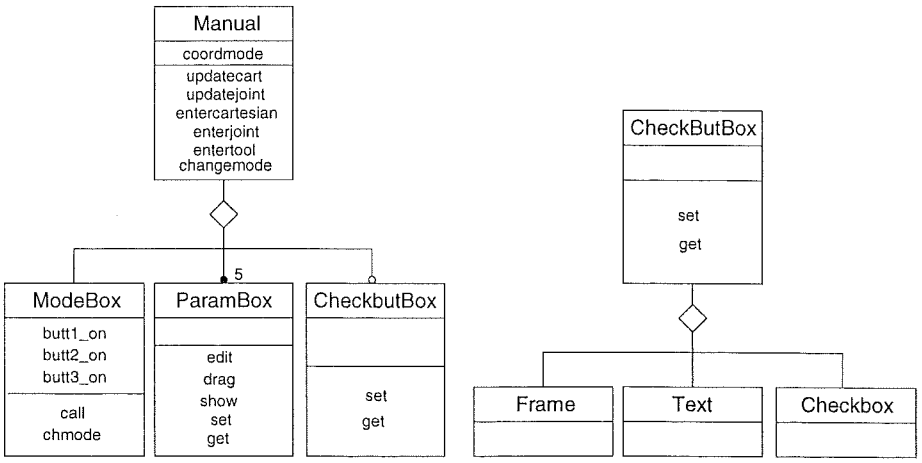


Figur 4.4 Objektdiagram för **Auto** och **Butt4x1Box**

Blocket **Butt4x1Box** är ett generellt block och består av en ram, en titel och fyra tryckknappar. Blockets titel och knapparnas titlar anges när blocket skapas. Knapparnas callback-rutiner kan ändras under drift med en operation **call**.

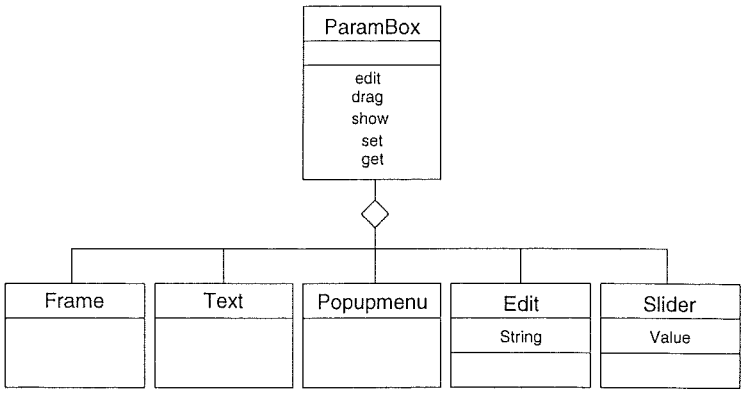
Manual

Blocket **Manual** består av följande block: en **ModeBox**, fem **ParamBox**ar och en **CheckButBox**. Från **Manual** kan roboten styras genom att ändra värdet på parametrarna i **ParamBox**arna. Det finns 2 olika koordinatmoder och varje koordinatmod har 5 positionsparametrar vardera, dessutom kan gripdonet öppnas och slutas. Detta blir totalt 11 olika positionsparametrar. För att överföra de nya parametervärdena till målsystemet används ett fördefinierat parametergränssnitt som kommunicerar via matcomm. När ett parametervärde ändras i någon av **ParamBox**arna så anropar en callback-rutin matcomm och parametervärdet överförs via ethernet till målsystemet. I målsystemet anropas en annan callback-rutin som tar hand om de nya parametervärdena och ger order till roboten att flytta sig till den nya positionen.



Figur 4.5 Objektdiagram för **Manual** och **CheckButBox**

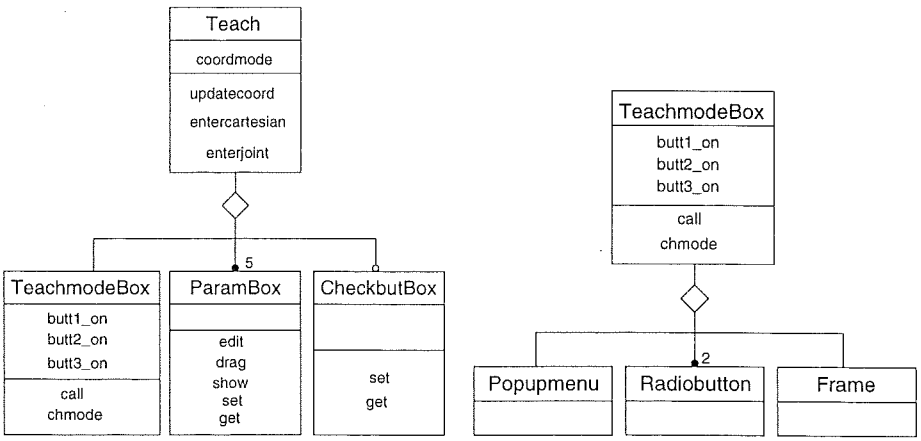
Blocket **ModeBox** används för att välja koordinatmod. Till vänster i figur 4.3 visas objektdiagrammet för **ModeBox**. Blocket **CheckButBox** används för att öppna och stänga gripdonet. Figur 4.5 visar objektdiagrammen för blocken **Manual** och **CheckButBox**. Blocket **ParamBox** fanns tillgängligt och är ett generellt block som kan användas för att ändra värdet hos en parameter. Figur 4.6 visar objektdiagrammet för **ParamBox**.



Figur 4.6 Objektdiagram för **ParamBox**

Teach

Blocket **Teach** består av följande block: en **TeachModeBox**, fem **ParamBoxar** och en **CheckButBox**. **Teach** används för att visa robotens position då roboten manövreras med styrspak. Det som visas i kontrollpanelen är således bara information om robotens position. Blocket **TeachModeBox** används för att visa vilket koordinatsystem som används vid manövreringen av roboten. Den innehåller också en knapp för val av uppdateringsmetod för positionsinformationen. När uppdatering ska ske skickar Matlab en begäran via ethernet till målsystemet att uppdatera parametrarna i parametergränssnittet som beskrevs i föregående stycke om **Manual**. Figur 4.7 visar objektdiagrammet för **Teach** och **TeachModeBox**. Objektdiagrammen för **CheckButBox** finns till höger i figur 4.5 och för **ParamBox** i figur 4.6.

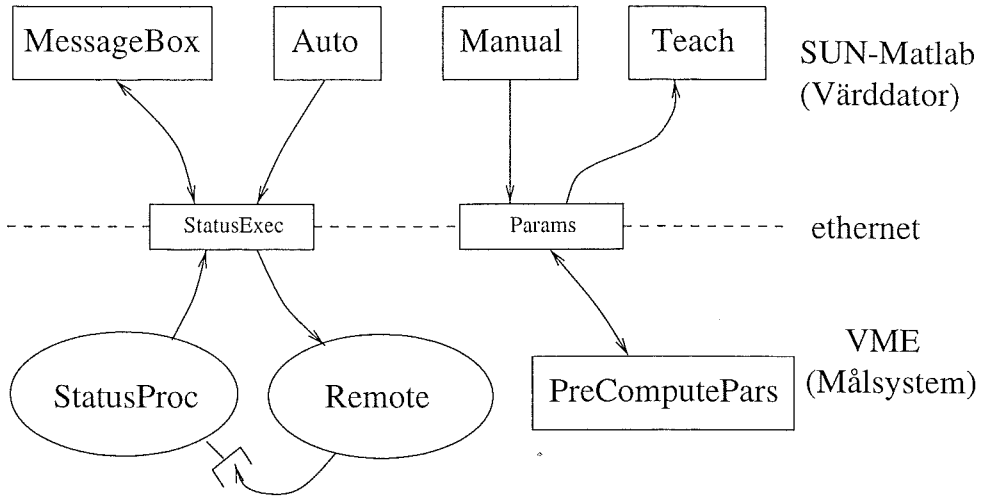


Figur 4.7 Objektdiagram för **Teach** och **TeachModeBox**

4.3 Kommunikation mellan värddatorn och målsystemet

På värddatorn finns kontrollpanelen och Matlab. Matlab understöder med funktionen **matcomm** kommunikation via ethernet till målsystemet. I målsystemet finns olika processer och callback-rutiner som sköter kommunikationen

tillbaka mot Matlab. I kontrollpanelen öppnas två olika kommunikationskanaler mellan Matlab (värddatorn) och målsystemet. Den ena används för att ge kommandon och skicka meddelanden. Den andra används för att skicka parametervärden. Se figur 4.8.



Figur 4.8 Struktur över kommunikationsvägar mellan värddatorn och målsystemet

MessageBox

MessageBox kan visa meddelanden. Det finns även en knapp som heter **update** som används för att be målsystemet att skicka upp av målsystemet buffrade meddelande. När **update** trycks skickas kommandot **update** via matcomm-socketen **StatusExec** ner till målsystemet. Sedan ställer sig **Messagebox** och väntar på att ett eller flera meddelanden ska komma från målsystemet. I målsystemet tar processen **Remote** hand om kommandot **update** som avkodas och utförs. I fallet då kommandot är **update** så betyder det att ett brev med meddelandet **update** läggs i processen **StatusProcs** brevlåda. När **StatusProc** hittar meddelandet **update** i sin brevlåda så skickas alla buffrade meddelanden upp till **MessageBox** via ethernet på matcomm-socketen. **MessageBox** som står och väntar på att meddelanden ska komma skriver ut dem i sitt textfönster så att användaren kan se dem.

Auto

Med de fyra knapparna (**Run**, **Stop**, **Step**, **Exit**) kan exekveringen av robotprogram styras. När en knapp trycks ner anropas en callback-rutin som skickar ner kommandot genom matcomm-socketen **StatusExec** via ethernet till målsystemet. I målsystemet tar processen **Remote** emot kommandona som avkodas och utförs. Dessutom skickas ett meddelande till processen **StatusProcs** brevlåda. Meddelandet tas emot av **StatusProc** och buffras. För att **StatusProc** ska skicka upp meddelande till **MessageBox** kräver den att få kommandot **update**. Därför skickas kommandot **update** från värddatorn direkt efter att kommandona **run**, **stop**, **step** eller **exit** skickades. På detta sätt så kommer meddelanden skrivas ut i **MessageBox** även då något av dessa kommandon ges.

Manual

Manual används för att direkt styra roboten från kontrollpanelen. Genom att välja någon av koordinatmoderna **cartesian** eller **joint** så väljer man hur man ämnar styra roboten. I båda fallen kommer 5 parameterboxar, och en box för att ändra om gripdonet är öppet eller slutet, att visas på skärmen. Genom att ändra värdet i någon av parameterboxarna så anropas en callback-rutin. Callback-rutinen läser värdet av varje parameter i parameterboxarna, samt även värdet om gripdonet är öppet eller stängt. Parametervärdena skickas via ethernet med ett parametergränssnitt i matcomm till målsystemet. I målsystemet anropas callback-rutinen **PreComputePars** av parameterservern vilket flyttar roboten till den nya positionen.

Teach

Teach används för att styra roboten via styrbollen. Istället för att man anger robotens nya position, som i **Manual**, så ska kontrollpanelen nu visa robotens position. För att göra detta används samma parameterboxar som i **Manual**. Enda skillnaden är att man nu inte kan ändra värdena i parameterboxarna utan de är bara till för att visa robotens position. Efter att kontrollpanelen överlämnat kontrollen till styrbollen så har kontrollpanelen ingen aning om när roboten flyttar sig varför callback-rutiner inte kan användas för att uppdatera parameterboxarna. Istället har jag valt att låta användaren själv sköta uppdateringen. Det finns tre olika alternativ för att uppdatera parameterboxarna: **Check**, **Loop** eller **Cont**. **Check** uppdaterar en gång, **Loop** uppdaterar en gång per sekund och **Cont** uppdaterar hela tiden. Detta förklaras mer ingående i stycket Klient-server nedan. I samtliga fall sker uppdateringen på samma sätt. **Teach** skickar en begäran till målsystemet om att få robotens positionskoordinater i parametergränssnittet uppdaterade. Callback-proceduren **PreComputePars** uppdaterar positionskoordinaterna i parametergränssnittet och skickar upp de nya värdena till Matlab där **Teach** tar emot dem och visar dem i parameterboxarna.

PreComputePars

PreComputePars är en callback-rutin som används när **Manual** eller **Teach** använder parametergränssnittet. De parametrar som ingår i denna parameteruppsättning är förutom kartesiska koordinater, ledvinkelkoordinater, gripdonets status även tiden som rörelsens ska ta samt en variabel som talar om vilket koordinatsystem som används (**coordmode**). Beroende på vilket värde **coordmode** har så kommer **PrecomputePars** att utföra olika saker. Variabeln **coordmode** används således som en form av kommando till **PreComputePars**. Värdet av **coordmode** betyder följande:

0 : De kartesiska positionsparametrarna har blivit uppdaterade och att roboten ska flyttas till ny position. **PreComputePars** kollar att den nya positionen och den i rymden rätlinjiga vägen dit ligger innanför robotens arbetsområde och om den gör det så flyttas roboten till den nya positionen.

1 : Ledvinkelparametrarna har blivit uppdaterade och att roboten ska flyttas till ny position. **PreComputePars** kollar att den nya positionen ligger innanför robotens arbetsområde och om den gör det så flyttas roboten till den nya positionen.

2 : Koordinatmoden **tool** är aktiv. Denna mod är inte implementerad ännu utan bara reserverad för framtiden.

3 : Kontrollpanelen har begärt att både kartesiska och ledvinkelkoordinater ska uppdateras. Anropas av kontrollpanelen för att få information om robotens position, exempelvis då kontrollpanelen går in i ny koordinatmod i moden **Manual** eller då moden **Teach** är aktiv och användaren vill uppdatera robotens positionskoordinater. För att kontrollpanelen i moden **teach** ska veta vilken koordinatmod som används av styrbollen, så tilldelas variabeln **coordmode** värdet 0 om kartesiska koordinater används och 1 om ledvinkelkoordinater används. Om inte styrbollen används så tilldelas **coordmode** värdet 4.

4 : Styrbollen ska överta kontrollen.

Remote

Processen **Remotes** uppgift är att upprätthålla matcomm-förbindelsen via socketen **StatusExec** till Matlab på värddatorn. **Remote** väntar på kommando från värddatorn. När den mottagit ett kommando avkodas kommandot och därefter utförs det. Sedan ställer sig processen och väntar på ett nytt kommando. Förutom den beskrivning som tidigare gjorts av de kommandon som finns så följer nedan ytterligare en förklaring som behandlar mer programmeringstekniska aspekter.

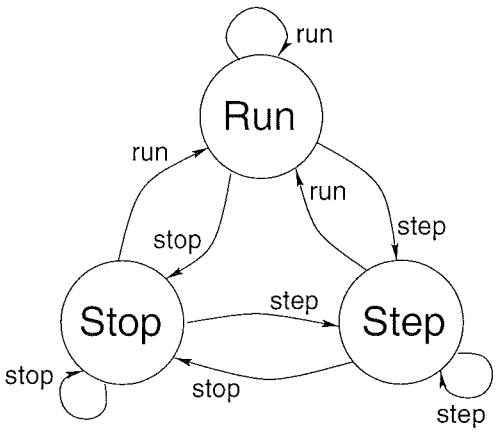
run : Exekveringskontrollen övergår till tillståndet **run**, se figur 4.9. Händelsen **ReadyToExecute** sker vilket leder till att robotinstruktioner som står i kö för att bli utförda nu börja betas av en efter en. Det skickas även ett meddelande till processen **StatusProcs** brevlåda. **StatusProc** buffrar meddelandet tills den får order att sända upp dem till kontrollpanelen.

stop : Om roboten inte hunnit börja på nästa robotinstruktion så hindras den från att börja exekvera till händelsen **ReadyToExecute** sker. Roboten övergår till tillståndet **stop**, se figur 4.9. Det skickas även ett meddelande till processen **StatusProcs** brevlåda. **StatusProc** buffrar meddelandet tills den får order att sända upp dem till kontrollpanelen.

step : Om roboten inte hunnit börja på nästa robotinstruktion så sker händelsen **ReadyToExecute** vilket leder till att robotinstruktionen som står i kö för att bli utförd nu börjar exekveras. Roboten övergår till tillståndet **step**, se figur 4.9. Det skickas även ett meddelande till processen **StatusProcs** brevlåda. **StatusProc** buffrar meddelandet tills den får order att sända upp dem till kontrollpanelen.

exit : Roboten övergår till tillståndet **stop**. Omstarten sker genom att ytterligare en process av typen som kör robotprogrammet startas varefter den gamla processen termineras.

update : Skickar ett meddelande till processen **StatusProcs** brevlåda. **StatusProc** ser att meddelandet är **update** och skickar upp alla buffrade meddelanden via ethernet genom matcomm-socketen **StatusExec** till Matlab på värddatorn.



Figur 4.9 Tillståndsgraf för exekveringskontrollen

StatusProc

Processen **StatusProcs** uppgift är att ta emot textmeddelanden och buffra dem tills den får order om att sända upp dem till kontrollpanelen på värddatorn. **StatusProc** har en brevlåda knuten till sig. I denna brevlåda kan övriga processer lägga meddelanden som de vill att användaren ska kunna se i **MessageBox** på kontrollpanelen. **StatusProc** ligger hela tiden och väntar på att den ska få ett meddelande i sin brevlåda. När den får ett meddelande kollar den vilken typ meddelandet är av. Om det inte är av typen **update** så lagras meddelandet i en länkad lista. När meddelande av typen **update** kommer så skickas alla tidigare lagrade meddelanden upp till kontrollpanelen på värddatorn där de tas emot och visas i **MessageBoxen**.

Klient-server

Kontrollpanelen och målsystemet fungerar som en klient-server vilket betyder att kontrollpanelen ger order om vad målsystemet ska göra och aldrig tvärtom. Orsaken till att jag valt denna struktur beror på att jag inte önskar ha några processer i Matlab för att kontrollpanelen ska fungera. På detta sätt låser inte kontrollpanelen upp Matlab utan samma Matlab kan användas även av andra tillämpningar. Hela kontrollpanelen är istället händelsestyrd och callback-rutiner utför allt användaren begär att kontrollpanelen ska göra.

Det finns dock även nackdelar både för användaren och programmeraren med denna struktur. Nackdelarna för användaren är att kontrollpanelen inte automatiskt uppdateras när något händer i målsystemet. Kontrollpanelen måste begära att målsystemet ska skicka upp information. Detta kan bli ett problem när man ska visa text i **MessageBox** och vid styrning med styrbollen. Lösningen i dessa fall har blivit en knapp där användaren kan begära att få informationen uppdaterad. Till **MessageBox** finns en knapp **update** som skickar en begäran om att få i målsystemet eventuellt buffrade meddelanden uppskickade till kontrollpanelen. I moden **teach** där styrbollen används så finns tre olika varianter för att uppdatera positionskoordinaterna. Man kan välja mellan att uppdatera endast vid begäran (**Check**). Man kan också välja att få positionskoordinaterna uppdaterade varje sekund (**Loop**). Det tredje sättet är att uppdatera hela tiden (**Cont**). Varianterna **Cont** och **Loop** har vissa nackdelar. Med dessa låser man helt eller delvis upp Matlab så att det inte samtidigt kan användas till andra tillämpningar.

För programmeraren är nackdelen att man måste spara all information

som ska bevaras i de grafiska objektens **userdata**[2]. För att sedan hitta de sparade variablerna måste man använda sig av en funktion som tar reda på vilket fönster som är aktivt på skärmen. I det aktiva fönstrets **userdata** måste man tidigare ha lagrat handtag till olika grafiska objekt för att det ska vara möjligt att hitta de grafiska objekt i vars **userdata** man lagrat sina variabler. (Om man istället hade kunnat använda en process så hade allt kunnat sparas i variabler i processen, men detta är inte möjligt i Matlab.)

4.4 Svårigheter och lösningar

De svårigheter som jag stötte på vid implementationen av exekveringskontrollen var följande:

Återstart av avbruten instruktion

För att kunna avbryta roboten mitt i en rörelse och sedan kunna återstarta den kan man inte tvärstoppa roboten. Ty ett tvärstopp av roboten skulle leda till att precisionen hos rörelsen skulle gå förlorad. Istället måste trajektorier för en inbromsning beräknas så att roboten lugnt och sansat kan bromsas in. Samma sak gäller vid återstart av rörelsen. Det går inte att göra en rivstart med roboten utan även i detta fall måste trajektorier beräknas för att lugnt och sansat sätta roboten i rörelse för att inte förstöra precisionen hos robotrörelsen.

Stega fram robotinstruktioner

För att kunna stega fram robotinstruktion för robotinstruktion så måste brytpunkter läggas in mellan varje robotinstruktion. Befintliga procedurer för att utföra robotrörelser används dock vid fler tillämpningar än vid automatisk exekvering av robotprogram varför det vore olämpligt att bygga in dessa brytpunkter i dessa procedurer. Jag har istället valt att kapsla dessa proceduranrop i speciella procedurer som bara används vid exekvering av robotprogram. I dessa speciella procedurer kontrolleras att det är tillåtet att exekvera nästa robotinstruktion innan robotrörelsen utförs. Om det inte skulle vara tillåtet väntar den tills den får tillåtelse att fortsätta.

Återstart av robotprogram

För att kunna återstarta exekveringen av robotprogrammet från första instruktionen igen så måste först pågående robotprogram avslutas. Detta sker genom att avbryta robotens rörelse, terminera processen som exekverade robotprogrammet samt starta en ny likadan process som startar exekvera robotprogrammet från första instruktion.

När kommandot **exit** ges för att avbryta robotprogrammet så avbryts trajetoriegenereringen och roboten stoppas. Processen **Remote** ställer sig att vänta på att händelsen **ReadyToRestart** ska ske. Robotinstruktionen avbryts och robotprogrammet kommer till en brytpunkt där det upptäcks att processen som exekverar robotprogrammet ska termineras. Innan processen termineras så utför den händelsen **ReadyToRestart**. När processen **Remote** ser att händelsen **ReadyToRestart** har skett så skapar den en process av samma typ som den terminerade. Den nya processen börjar exekvera robotprogrammet från första instruktionen.

[2]Userdata är ett fördefinierat attribut i varje grafiskt objekt i Matlab. Se även appendix B.

5. Dynamisk bindning

Det normala sättet att utveckla program är att man skriver sin programkod, kompilerar och länkar samman koden till ett exekverbart program. Detta kapitel beskriver ett sätt där man även under exekveringen har möjlighet att länka med ytterligare exekverbar kod. För att åstadkomma detta är det lämpligt att använda en skiktad programstruktur. Jag kommer i detta kapitel beskriva hur jag använt programstrukturen Open Robot Control (ORC) för att lösa de problem som uppstår vid dynamisk medlänkning av exekverbar kod. Kapitel 5.1 beskriver vilken uppgift den dynamiska länkaren har. Kapitel 5.2 beskriver det som läsaren bör veta om ORC. I kapitel 5.3 och 5.4 beskrivs hur jag tillämpat ORC vid dynamisk bindning av kod på olika nivåer.

5.1 DynamicLinker

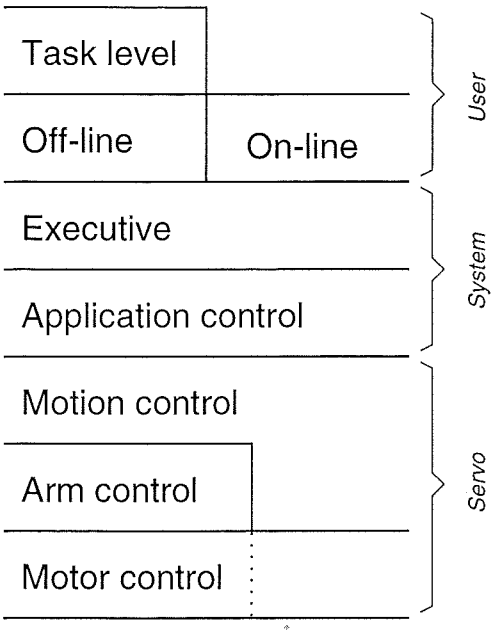
Den dynamiska länkaren har utvecklats inom tidigare arbeten [3, 9, 7, 8]. Den dynamiska länkaren används för att under drift binda ny exekverbar kod till systemet. För att det övriga systemet ska kunna hitta den nya koden och vice versa så måste man normalt länka ihop delarna innan exekveringen startar. Det finns det tyvärr inte alltid möjlighet till, och därför har en dynamisk länkare som löser detta problemet skapats. I målsystemet finns denna tillgänglig via modulen **DynamicLinker**, och på värddatorn finns ett script **vmelink**.

Den dynamiska länkaren skapar en symboltabell över alla operationer som ska kunna utföras från den dynamiskt bundna koden. Detta är nödvändigt eftersom man vid kompileringstillfället inte vet vilken adress de olika operationerna kommer att ha vid exekveringstid. Symboltabellen används sedan av dynamiskt bunden kod för att hitta rätt adress till statiskt bundna operationer.

5.2 Open Robot Control

Open Robot Control (ORC) [7, 8] är ett sätt att dela upp programstrukturen i skikt där hårdvaran finns underst och ovanpå denna definieras flera programmeringsnivåer för olika typer av användare, se figur 5.1. Gränsen mellan två skikt består av ett väl definierat gränssnitt som åskådliggör vilka funktioner som är tillgängliga för det andra skiktet. Uppdelningen är gjord så att de olika skikten tillhandahåller olika typer av programmeringsmöjligheter. System som byggs enligt ORC blir öppna system där avancerade användare har tillgång till avancerad användnings- och omprogrammeringsmöjligheter. Det viktigaste skiktet är förmodligen **application**, som tillhandahåller en ny möjlighet att konfigurera systemet för olika färdigheter.

Gränssnittet till skiktet **user** innehåller två olika skikt, ett som används vid kompileringstid och ett som används i exekveringstid. Gränssnittet som används vid exekveringstid är skiktet **executive**, som innehåller robot programmeringsspråket som används. Gränssnittet som används vid kompileringstid är skiktet **applications** gränssnitt mot skiktet **executive**. I skikten **executive** och **application** är det möjligt att dynamiskt ladda ner exekverbar kod

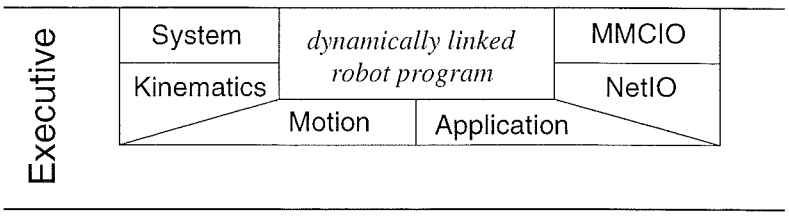


Figur 5.1 Programstruktur enligt Open Robot Control (ORC).

i form av t.ex. robotprogram eller nya färdigheter. Detta beskrivs närmare i de två följande avsnitten.

5.3 Executive

När t.ex. en interpretator, eller ett robotprogram skrivet direkt i Modula-2 eller C, länkas med dynamiskt blir det en del av skiktet **executive**. Skiktet **executive** är förberett för den dynamiskt länkade koden genom att det har ett gränssnitt som talar om vad som är tillgängligt för den dynamiskt medlänkade koden. Den dynamiskt medlänkade koden omges således av ett gränssnitt som talar om vilka operationer som är tillåtna. På detta sätt kan man enkelt bygga in begränsningar för vad som är tillåtet för robotprogrammet att göra. Samtidigt talar man om för den dynamiska länkaren vilka procedurer och funktioner som finns i systemet. Detta krävs för att det överhuvudtaget ska vara möjligt att länka med exekverbar kod under drift.



Figur 5.2 Gränssnittet mellan den dynamiskt bundna koden och den statiskt bundna koden i skiktet **Executive**.

Gränssnittet mellan den dynamiskt och den statiskt bundna koden är uppdelat i olika moduler efter funktionalitet enligt figur 5.2. De olika modulerna innehåller följande operationer:

System: Anrop till realtidskärnan (processhantering, tidshantering), minnesallokering och deallokering, semaforer, monitorer, brevlådor, matematikfunktioner samt även den dynamiska länkaren.

NetIO: Kommunikation via ethernet (MatComm, HostParams samt Params).

MMCIO: Utskrift på skärm, inläsning av tecken från tangentbordet, strängkonverteringsoperationer samt utskrift av CartesianType, JointType, ActuatorType och RangeType.

Application: Tillhandahåller vid kompileringstillfället okända operationer, dvs dynamiskt nedladdade färdigheter i skiktet **Application**.

Motion: Operationer för att flytta roboten, exekveringskontroll av robotprogram, öppna och stänga gripdonet samt inställning av regulatorparametrar.

Kinematics: Transformationer mellan koordinattyperna CartesianType, JointType och ActuatorType.

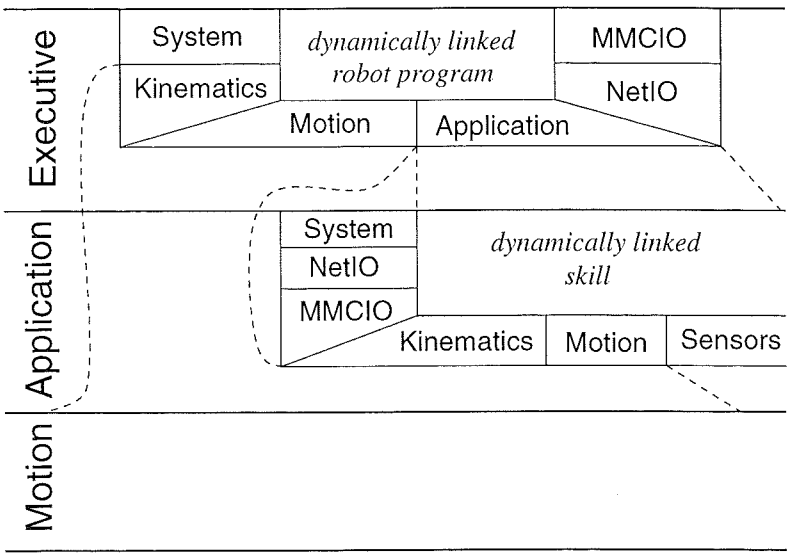
5.4 Application

Skiktet **application** innehåller de färdigheter (eng: skill) som systemet har. Skiktet **executive** använder färdigheterna i skiktet **application** för att utföra operationer på roboten. Det finns även möjlighet att dynamiskt ladda ner nya färdigheter. Liksom i skiktet **executive** finns det även i **application** en gränssnitt mellan den dynamiskt bundna koden och den statiskt bundna koden. Gränssnittet består i stort sett av samma moduler som i skiktet **executive** men det finns två skillnader. **Modulen application** finns inte utan är ersatt av modulen **sensors**. Övriga moduler är de samma som i skiktet **executive**. Modulen **sensors** beskrivs nedan:

Sensors: Operationer på externa givare som finns monterade på roboten. Det finns i skrivandets ögonblick funktioner för att aktivera lasergivaren och mäta avstånd.

Figur 5.3 visar gränssnitten i skikten **executive** och **application**. För att en dynamiskt bunden färdighet ska kunnas användas av ett dynamiskt bundet robotprogram måste den dynamiskt bundna färdigheten finnas i systemet innan det dynamiskt bundna robotprogrammet laddas, annars kan inte den dynamiska länkaren hitta alla färdigheter som robotprogrammet kräver. Det sker ingen automatisk nedladdning av de dynamiskt bundna färdigheter som robotprogrammet använder utan användaren måste själv se till att dessa finns nedladdade för att den dynamiska länkningen av robotprogrammet ska lyckas.

Vidare krävs att modulen **application** ingår i det gränssnitt mellan dynamiskt bunden och statiskt bunden kod som finns i skiktet **executive**. Modulen **application** innehåller en tabell där alla dynamiskt bundna färdigheter finns registrerade. När det dynamiskt bundna robotprogrammet installeras går den dynamiska länkaren igenom tabellen för att hitta adressen till de färdigheter som tidigare blivit installerade. Detta är anledningen till att färdigheten måste vara nedladdad innan robotprogrammet kan laddas.



Figur 5.3 Gränssnittet mellan den dynamiskt bundna koden och den statiskt bundna koden i skikten **Application** och **Executive**.

5.5 Användning och vidareutveckling

Gränssnitten mellan dynamisk och statisk bunden kod är som tidigare nämnts uppbyggda av olika moduler. Ändringar i gränssnittet kan ske på minst tre olika sätt utan att för den del ändra principen för gränssnittets uppbyggnad. För det första kan man lägga till och ta bort operationer från den befintliga modulerna genom att ändra i proceduren **Symbols** som finns i implementationsmodulerna. För det andra kan man ta bort en befintlig modul och ersätta den med en egen (på detta vis är det lättare att återställa systemet om man skulle vilja det). För det tredje kan man lägga till nya moduler om de operationer man vill införa inte passar i någon av de befintliga modulerna. Det hela är ett flexibelt system som går att anpassa efter behov.

Gränssnittet inför också en begränsning av vilka operationer som är tillåtna att utföra av den dynamiskt bundna koden. Detta kan ses som en säkerhet i systemet men givetvis även som en begränsning. Detta är naturligtvis en subjektiv bedömning. En robottillverkare har ett ansvar för sina robotar och kan därför inte tillåta lekmän programmera om roboten så att den kan komma till skada eller kanske t.o.m. orsaka människor skada. Det måste därför finnas ett sätt att begränsa vilka operationer roboten kan tillåtas utföra. Detta vore ett problem om vanlig dynamisk länkning hade använts, men låter sig enkelt göras med tekniken beskriven ovan.

6. Sensorbaserade rörelser

I detta kapitel beskrivs hur det ursprungliga systemet modifierat för att kunna använda roboten för sensorbaserade rörelser. Med detta menas att roboten under en rörelse hämtar information från en yttre givare på eller vid roboten och att roboten beroende på mätdata från givaren kan ändra rörelsen så att den anpassas till de uppmätta förändringarna i omgivningen.

6.1 Trajektoriegenerator

Modulen **Trajec** innehåller operationer för att generera trajektorier. I denna modul har jag gjort vissa övergripande förändringar för att det skulle vara möjligt att utföra avancerade sensorbaserade rörelser. I modulen **Trajec** finns en process som heter **TrajProcess**. Det är denna process som genererar de trajektorier som roboten sedan följer. Den räknar ut punkter i rymden där roboten ska befinna vid varje tidpunkt (med tidpunkt menas samplingstidpunkt). För att man ska kunna påverka robotens rörelse måste således **TrajProcess** ta hänsyn till värdet som läses från givaren och anpassa börvärdena till robotens rörelse efter de nya uppmätta förhållanden. Istället för att skriva en ny trajektoriegenerator till varje tillämpning så används så kallade callback-procedurer.

6.2 CartGoto

När en rörelse ska utföras kan ett flertal olika procedurer användas (beroende på bl.a. vilket koordinatsystem som används) för att beordra att trajektorier för rörelsen ska beräknas. En av dessa rutiner heter **CartGoto**. Vid ett anrop till **CartGoto** måste fyra inparametrar anges. Det är vilken punkt i det kartesiska koordinatsystemet som roboten ska röra sig till, hur lång tid rörelsen ska ta, pekaren till en callback-rutin och en boolesk variabel som talar om rörelsen är sensorbaserad eller ej. Dessutom finns det en utparameter som talar om vilket nummer rörelsen har.

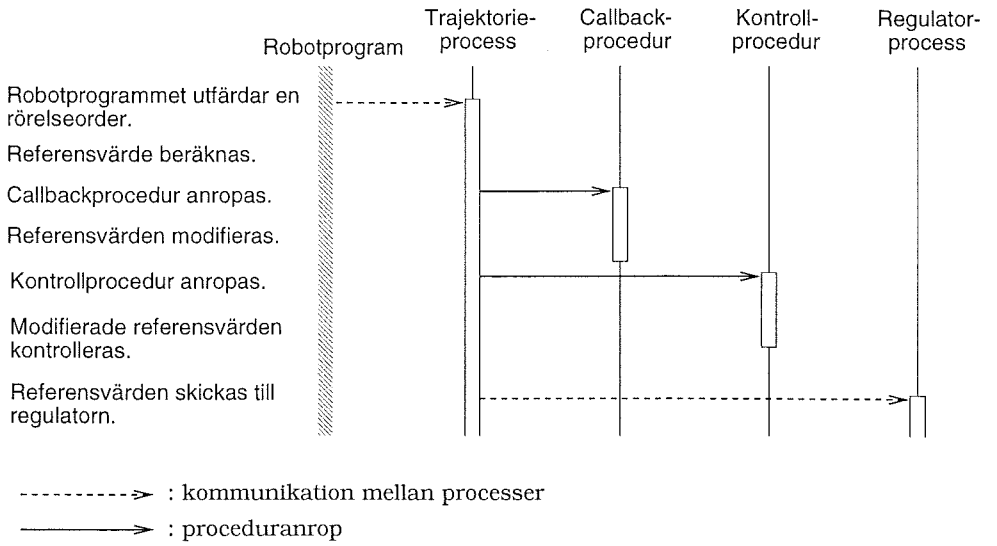
Om föregående rörelse var sensorbaserad så väntar **CartGoto** till rörelsen är klar innan den ger order om en ny rörelse. Först kontrolleras att roboten kan förflytta sig till den nya punkten. Sedan inkrementeras en global variabel som håller reda på rörelsenummer. Proceduren **CheckStartPosAndSendTraj** anropas och denna proceduren lägger ett brev i brevlådan **TrajMailBox**. Brevet innehåller all information som **TrajProcess** behöver för att generera trajektorier till rörelsen.

6.3 TrajProcess

TrajProcess delar upp rörelsen som roboten ska utföra i små steg och skickar med jämna mellanrum (standardvärde f.n. 36 ms) nya rörelseorder till roboten. När processen inte har någon trajektoria att generera väntar tills det kommer

ett brev i brevlådan **TrajMailBox**. När den fått en ny rörelseorder går processen in i en repeat-sats som den inte lämnar förrän trajektorian blivit färdig genererad eller den får order om att av bryta rörelsen.

Varje sampel beräknas en ny punkt i trajektorian. När rörelsen beordrades angavs förutom vart roboten skulle röra sig även hur lång tid rörelsen skulle vara. Det är med utgångspunkt från denna information och var roboten befann sig när rörelsen började som trajektorian beräknas. Efter att trajektorian till nästa steg blivit beräknad men innan den skickats ut som rörelseorder till roboten så anropas den callback-rutin som användaren angav när rörelseordern utfärdades. Det är i denna callback-rutin som användaren har möjlighet att programmera in avläsning av en givare och ändra trajektorian. Callback-rutinen har en inparameter av typen **Context**, som behandlas i avsnitt 6.4. Callback-rutinen kan genom att ändra värdena i **Context** ändra trajektorian. Om det är en sensorbaserad rörelse så måste context granskas efter det att callback-rutinen ändrat den. Systemet kan inte tillåta användaren att ändra trajektorian så att roboten gör rörelser som den inte är konstruerad för. Därför anropas direkt efter callback-anropet proceduren **CartCheckLimitAndUpdate** som kontrollerar att trajektorian håller sig inom vissa gränser och överför i så fall de nya värdena från **Context** till **TrajProcess** lokala variabler. Se figur 6.1.



Figur 6.1 Visar schematiskt hur callback-rutiner och kontrollrutiner används för att åstadkomma tillförlitliga sensorbaserade rörelser.

6.4 Context

Context är en pekare till datatypen **InternalState**. **InternalState** innehåller ett antal olika parametrar som beskriver status för den trajektorian som genererats, se tabell 6.1.

Context används av **TrajProcess** för att skicka information om den beräknade trajektorian då procedurer anropas. Procedurer som använder **Context** som inparameter är bl.a. callback-procedurer och proceduren **CartCheckLimitAndUpdate** som beskrivs i kapitel 6.6

Tabell 6.1 Datatypen **InternalState** som **Context** pekar på.

Variabel	Typ	Förklaring
ActTime	LONGREAL	Tid sedan rörelsen påbörjades
RemainingTime	LONGREAL	Återstående tid av rörelsen
IsAtTarget	BOOLEAN	TRUE om TrajProcess har beräknat sista steget i en rörelse
Coordmode	CoordModeType	Aktiv koordinatmod
CartDesiredOld	CartesianType	Kartesiska koordinaten för föregående steg
CartNom	CartesianType	Den av TrajProcess beräknade kartesiska koordinaten för nästa steg
JointDesiredOld	JointType	Ledvinkel koordinaten för föregående steg
JointNom	JointType	Den av TrajProcess beräknade ledvinkel koordinaten för nästa steg
TimeStep	LONGREAL	Tiden för nästa steg
NormStep	LONGREAL	Normaliserat värde av tiden för nästa steg
CartStep	CartesianType	Längden i kartesiska koordinater av nästa steg
JointStep	JointType	Längden i ledvinkel koordinater av nästa steg

6.5 Callback

Beroende av värdet på variabeln **Coordmode** i **InternalState** kan callback-proceduren avgöra vilken koordinatmod som är aktiv. Detta är nödvändigt för att veta vilka variabler i typen **InternalState** som innehåller sanna värden samt vilka variabler som ska tilldelas nya värden för att trajektorian ska ändra sig. Datatypen **InternalState** innehåller tre variabler med vilka det är möjligt att ändra den genererade trajektorian. Variablerna är **NormStep**, **CartStep** och **JointStep** (se tabell 6.1).

Variabeln **NormStep** innehåller ett normaliserat värde av tiden för nästa steg, den av **TrajProcess** genererade trajektorian svarar mot **NormStep**=1. Genom att ändra denna variabel kan man få rörelsen att gå fortare eller saktare, samt även baklänges genom att tilldela **NormStep** negativt värde. Denna funktion kan t.ex. utnyttjas om roboten rör sig på ett sätt att den maximala korrektionen hos den sensorbaserade regulatorn, som callback-proceduren kan ses som, inte är tillräckligt stor för att korrigera robotens rörelse. Genom att tilldela **NormStep** ett värde mellan noll och ett så tror **TrajProcess** att tiden som rörelsen ska ta är kortare än vad den egentligen är varför stegen blir kortare.

Variablerna **CartStep** och **JointStep** används vid olika koordinatmoder, **CartStep** då kartesiska koordinater används och **JointStep** då ledkoordinater används. Båda innehåller inkrementet för nästa steg (längden av nästa steg). Genom att ändra längden av en av dessa variabler så ändras nästa rörelses längd. Detta används för att korrigera den av **TrajProcess** genererade trajektorian. Det är således dessa variabler som ska ändras för att erhålla sensorbaserad reglering av robotens position utanför den nominella banan. För att beräkna hur stort inkrementet ska vara för att inte påverka den av **TrajProcess** genererade trajektorian används variablerna **CartDesiredOld**, **CartNom** eller **JointDesiredOld** och **JointNom**. **CartNom** minus **CartDesiredOld** eller **JointNom** minus **JointDesiredOld** ger inkrementet beroende på vilket koordinatsystem som används.

6.6 CartCheckLimitAndUpdate

Efter att callback-proceduren anropats måste de variabler i **Context** som callback-proceduren ändrat kontrolleras och kopieras till **TrajProcess** lokala variabler. Detta sker i proceduren **CartCheckLimitAndUpdate**.

För att kunna kontrollera om callback-procedurens ändringar är tillåtna finns definierade gränser för vad som är tillåtet. Tabell 6.2 visar vilka variabler som datatypen innehåller.

Tabell 6.2 Datatypen **Parameters** som innehåller gränserna för vad som är tillåtet för en callback-procedur att ändra. **ParValue** och **LimitParType** är det samma som LONGREAL och ARRAY[1..5] OF LONGREAL.

Variabel	Typ	Förklaring
iMaxNormStep	ParValue	Maximala värdet som NormStep får tilldelas
iMinNormStep	ParValue	Minimala värdet som NormStep får tilldelas
iMaxCartStep	LimitParType	Maximala värdet som CartStep får tilldelas
iMinCartStep	LimitParType	Minimala värdet som CartStep får tilldelas
iMaxCartCorrs	LimitParType	Maximala värdet för summan av alla CartStep
iMinCartCorrs	LimitParType	Minimala värdet för summan av alla CartStep

Anledningen till att variablerna i **Parameters** är av typen **ParValue** och **LimitParType** istället för LONGREAL och ARRAY[1..5] OF LONGREAL är att variablerna ingår i ett parametergränssnitt i Matlab. Parametergränssnittet gör det möjligt för användaren att från värddatorn ändra gränserna för hur mycket callback-proceduren ska kunna ändra trajektorian som **TrajProcess** genererar.

Förutom begränsningen som infördes i och med typen **Parameters** så finns det en fysisk gräns för robotens arbetsområden. Innan variablerna i **Context** kopieras till **TrajProcess** lokala variabler kontrolleras därför även att punkten som roboten beordrats att gå till ligger inom robotens arbetsområde.

Om inte den av callback-proceduren beräknade trajektorian ligger inom alla tillåtna gränser förkastas den och **TrajProcess** egengenererade trajektoria används istället. På detta vis skyddas roboten från att utföra rörelser som den inte är lämpad för. Tilläggas bör att det även ligger ett ansvar på användaren vad det gäller inställning av variablerna i **Parameters** som definierar gränserna för hur mycket callback-proceduren får ändra variablerna i **Context**.

7. Exempel

I detta kapitel presenteras några olika exempel på hur de delar i systemet som implementerats kan användas. Kapitel 7.1 redogör för hur den sensorbaserade Z-regulatorn är implementerad. Kapitel 7.2 visar hur Z-regulatorn kan användas för att detektera gjutskägg hos gjutgods och kapitel 7.3 redogör för hur backning och repetition av rörelse kan tillämpas.

7.1 Dynamiskt bunden kod för sensorbaserad konturföljning

När roboten rör gripdonet över en yta så reglerar Z-regulatorn robotarmens höjdläge så att den hela tiden ska befinna sig 50 mm över den underliggande ytan. För att mäta avståndet till underlaget används en lasergivare som finns monterad i robotens gripdon. Z-regulatorn finns implementerad som en callback-rutin och en pekare till den skickas med som inparameter när en rörelse med Z-reglering ska utföras. Den exekverbara koden för Z-regulatorn finns inte statiskt bunden i systemet utan måste laddas in till målsystemet och länkas med dynamiskt.

Laser

Modulen **Laser** är statisk bunden i systemet eftersom den kan tänkas användas även av andra tillämpningar är Z-regulatorn. Modulen **Laser** tillhandahåller funktioner för att läsa av värdet hos lasergivaren.

Lasergivaren, se figur 7.1, har vissa nackdelar som modulen **Laser** försöker gömma och skydda det övriga systemet ifrån. Lasergivarens arbetsområde är relativt begränsat, den har ett område 50 ± 5 mm från givaren där den ger ett riktigt värde. Utanför detta mätområde ger lasergivaren felaktiga mätvärden. Lasergivaren kan dessutom bara detektera ytor om de reflekterar det ljus som lasern använder, t.ex. så går det inte att detektera vissa svarta ytor. För att kunna använda lasergivaren måste den dessutom vara riktad så att laserstrålen lyser neråt. Detta är en skyddsåtgärd för att undvika ögonskador. En tiltgivare har monterats på lasergivaren för att känna av hur mycket den lutar. Om tiltgivaren lutar för mycket, vilket betyder att även lasergivaren lutar mycket, så stängs lasergivaren av.

Modulen **Laser** innehåller följande operationer:

Init: Initierar modulen **Laser**.

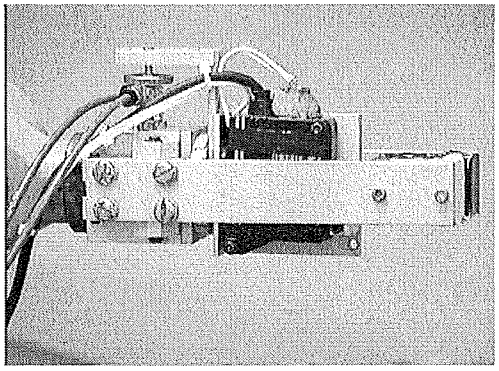
Operate: Sätter på och stänger av lasergivaren.

IsOperating: Kontrollerar om lasergivaren är på eller avslagen. Returnerar **TRUE** om den är påslagen.

Distance: Mäter avståndet till underlaget och returnerar det upp mätta avståndet i mm samt om det ligger inom lasergivarens mätområde.

Zregul

Modulen **Zregul** är dynamiskt bunden till systemet, eftersom den bara används vid en speciell tillämpning, och därför tar onödig plats om den alltid



Figur 7.1 Lasergivaren monterad i robotens gripdon. Givarsignalerna går via den svarta kabeln. Den vita kabeln går till tiltgivaren som applicerats ovanpå lasergivaren.

skulle länkas med statiskt. **Zregul** innehåller funktioner för att reglera roboten i höjdled. Modulen innehåller dels procedurer som kan anropas från ett robotprogram och dels procedurer som ska användas som callback-procedurer till trajektoriegeneratorn. Följande procedurer kan anropas från robotprogram:

FindSurface: Letar upp underliggande yta med utgångspunkt från den (x,y)-koordinat som roboten befinner sig. Den arbetar efter följande algoritm: Först höjs robotarmen i en sekund och om roboten skulle hitta en yta stannar den 50 mm över ytan. Om den inte hittade någon yta sänks robotarmen i 20 sekunder och om den hittar en yta så stannar den 50 mm över ytan. Om roboten inte hittade någon yta så fanns det ingen yta inom arbetsområdet.

MoveLinXYnotrack: Flyttar robotarmen till given (x,y)-koordinat på en bestämd tid utan att reglera i z-led eller skicka mätvärden till värddatorn (Matlab).

MoveLinXYtrackZ: Flyttar robotarmen till given (x,y)-koordinat på en bestämd tid samtidigt som reglering sker i z-led och mätvärden skickas till värddatorn (Matlab).

ScanOneLine: Flyttar robotarmen från aktuell position till aktuell position + (Δx, Δy) på en tid som är anpassad till rörelsens längd samtidigt som reglering sker i z-led och mätvärden skickas till värddatorn (Matlab).

ScanWithoutTrack: Flyttar robotarmen från aktuell position till aktuell position + (Δx, Δy, Δz) på en tid som är anpassad till rörelsens längd samtidigt som mätvärden skickas till värddatorn (Matlab). Om lasergivaren skulle hamna utanför arbetsområdet så backar roboten tillbaka och försöker igen.

GotoOperatingPos: Vrider gripdonet så att lasergivare lyser rakt ner så att den går att använda.

SetRobotPars: Ställer in parametrarna till den PID-regulator som Z-regulatorn använder.

Flera av de ovanstående procedurerna startar sensorbaserade rörelser med olika callback-procedurer, för att utföra olika uppgifter. Följande procedurer används som callback-procedurer:

Control: Läser av värdet hos lasergivaren och reglerar robotarmens position i z-led genom att ändra värdet på variabeln **CartStep** i **Context**. Till hjälp vid regleringen används en PID-regulator som finns implementerad i modulen **Pid**. Aktuell position läses av direkt från roboten och plotvärden skickas via en brevlåda till modulen **Zplot** där de sedan skickas upp till värddatorn där de plottas av Matlab. Plotvärdena är (x,y,z)-koordinater, felet som läses från lasergivaren samt tiden som rörelsen pågått.

ControlNoPlot: Fungerar som **Control** men plotvärdena sparas inte.

ControlWithBacking: Fungerar som **Control** men har en extra funktion för att klara av situationer då lasergivaren hamnar utanför sitt arbetsområde. Om detta skulle inträffa så backar roboten tillbaka genom att ändra värdet på variabeln **NormStep** i **Context**. När **NormStep** antar negativa värden så går rörelsen i motsatt riktning och tiden för resterande rörelse ökar (vilket från callback-rutinen kan uppfattas som att tiden går baklänges).

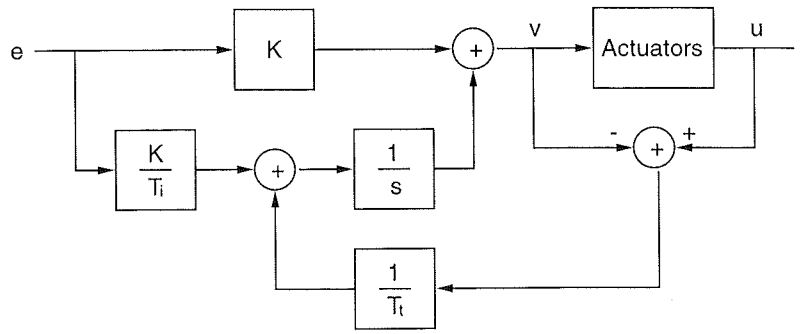
UpSearchCallback: Används av **FindSurface** för att söka uppåt. Anropar i sin tur **ControlNoPlot**.

DownSearchCallback: används av **FindSurface** för att söka neråt. Anropar i sin tur **ControlNoPlot**.

Eftersom samtliga callback-procedurer bara har en inparameter, **Context** så är det svårt att göra generella callback-procedurer, utan istället får man skapa en callback-procedur för varje tillämpning. Ett sätt att lösa detta skulle kunna vara att lägga till en inparameter till samtliga callback-procedurer. Denna nya inparameter skulle i så fall innehålla information om hur användaren vill att callback-proceduren skulle uppföra sig. Med denna metod skulle man inte behöva ha olika callback-procedurer för rörelser beroende på om de t.ex. vill plotta mätvärden till Matlab eller ej.

Pid

Modulen **Pid** innehåller en generell PID-regulator. Den används emellertid endast till Z-regulatorn varför även denna modul är dynamiskt bunden till systemet. Z-regulatorn använder dock bara P och I-delen av regulatorn eftersom D-delen introducerade för mycket brus i systemet. Regulator strukturen visas i figur 7.2.



Figur 7.2 PI-regulator med anti-windup. Parametrarna K, T_i och T_t kan ändras via ett parametergränssnitt i Matlab.

Z-regulatorn använder PI-regulatorn som en inkrementell regulator, dvs PI-regulatorn får bara reda på felet och räknar ut hur mycket robotens position ska korrigeras. Teorin för regulatorn är hämtad ur *Computer Implementation of Control system* [6].

Zplot

Modulen **Zplot** har till uppgift att ta emot mätvärden via en brevlåda och sända upp dem till värddatorn där de plottas i Matlab. **Zplot** är liksom **Zregul** dynamiskt bunden till systemet.

Zplot innehåller en process som tömmer brevlådan med mätvärdena och buffrar dem för att sedan skicka upp dem till värddatorn och Matlab. För att detta relativt tidskrävande arbete inte ska störa regleringen av roboten har processen tilldelats en tämligen låg prioritet, vilket får till följd att processen bara får processortid om processorn inte används till något viktigt. För att snabba upp kommunikationen till värddatorn så samlar processen ihop ett antal mätvärden och skickar sedan över dem på en gång. De mätvärden som **Zplot** tar emot i sin brevlåda och skickar vidare till värddatorn där Matlab tar emot dem beskrivs i tabell 7.1.

Tabell 7.1 De variabler som **Zplot** tar emot och skickar vidare till värddatorn för att plottas i Matlab.

Variabel	Typ	Beskrivning
t	LONGREAL	Tid sedan första samplet i rörelsen
x	LONGREAL	Robotens position i x-led
y	LONGREAL	Robotens position i y-led
z	LONGREAL	Robotens position i z-led
e	LONGREAL	Avvikelsen från önskad position i z-led (uppmätt av lasergivaren)

Följande procedurer kan användas för att kommunicera med den process som tar hand om mätvärdena:

PutInValue: Lägger en uppsättning mätvärden i **Zplots** brevlåda.

PutCommand: Lägger ett kommando samt en uppsättning mätvärden i **Zplots** brevlåda.

När mätvärdena skickats upp till Matlab för att plottas, så måste Matlab veta när samtliga mätvärden kommit eller när mätvärden för nästa rörelse börjar. Därför måste **Zplot** förmedla denna informationen till Matlab. Men eftersom **Zplot** inte kan veta om det ska komma mer mätvärden i dess brevlåda, så finns det kommandon som kan skickas till samma brevlåda som mätvärdena hamnar i. De kommandon som finns redogörs i tabell 7.2.

Tabell 7.2 De kommandon som finns för att kommunicera med **Zplot**.

Kommando	Beskrivning
cStarted	Innehåller första mätvärden till en rörelse
cFinished	Innehåller sista mätvärdet till en rörelse
cScanningFinished	Talar om att samtliga rörelser är klara
cQuit	Stänger förbindelsen med Matlab och terminerar **Zplot**

Kommandona lagras i variabeln e tillsammans med mätvärdena. För att samtidigt kunna lagra värdet av avvikelsen i z-led i e så används en speciell

teknik. Eftersom man vet att värdet av **e** aldrig kan bli stort så kan man lagra kommandon som stora tal i variabeln **e** enligt formeln 7.1. Där **E1** och **E2** är mer än dubbelt så stora som största möjliga **e** och **KommandoNummer** är numret på kommandot.

$$e = e + E1 + E2 \cdot KommandoNummer \tag{7.1}$$

Det finns dock en nackdel med detta förfarande. Noggrannheten hos **e** kommer nämligen att minska när stora tal adderas till **e**. Efter överväganden valde jag ändå denna metod och orsaken var att det redan finns en betydligt större felkälla nämligen det brus som tillkommer vid mätningen med lasergivaren. Bruset gör att de värdesiffror som försvinner ändå inte innehåller någon tillförlitlig information.

Ztracker

Modulen **Ztracker** innehåller ett gränssnitt som talar om för dynamiskt bundna robotprogram vilka operationer som finns tillgängliga i det dynamiskt bundna programpaketet **Ztracker**. Programpaketet **Ztracker** innehåller modulerna **Zregul**, **Pid**, **Zplot**, **Ztracker**. Även modulen **Ztracker** är dynamiskt bunden till systemet.

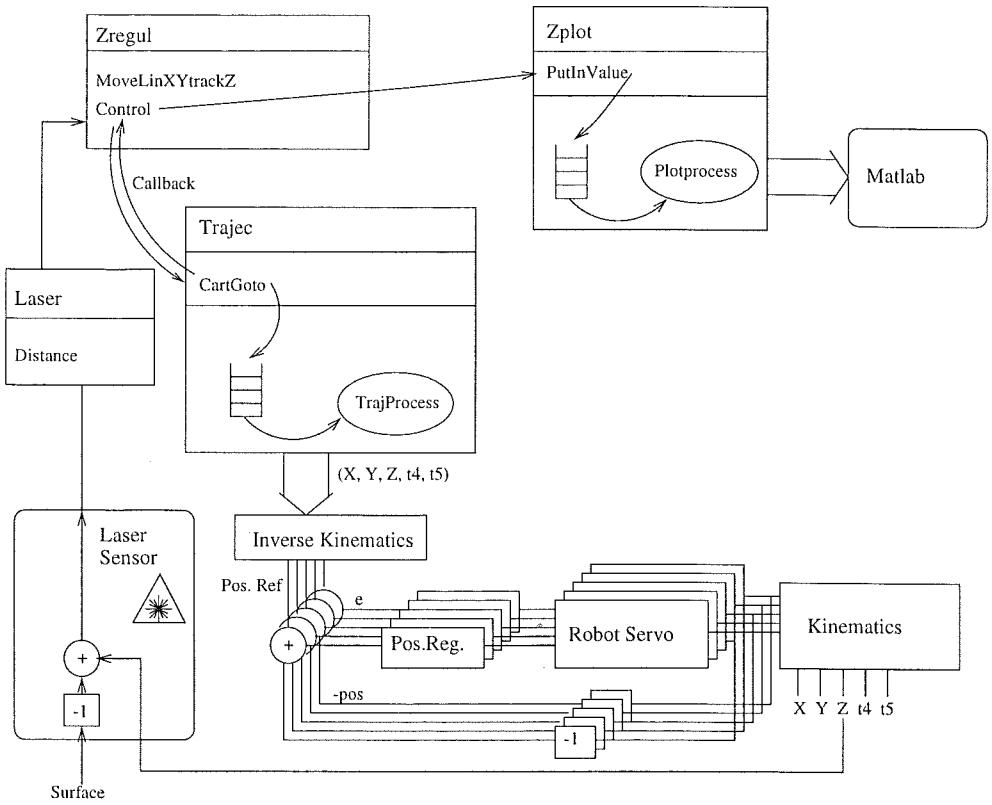
Enligt de termer som definierats tidigare så är programpaketet **Ztracker** en dynamiskt bunden färdighet hos systemet, se figur 5.3. Färdigheten **Ztracker** laddas dynamiskt in i skiktet **application**. För att det dynamiskt bundna robotprogrammet (som laddas in i skiktet **executive**) ska kunna hitta den dynamiskt bundna färdigheten **Ztracker** så registrerar **Ztracker** att den finns i systemet. Registreringen sker genom ett anrop till proceduren **InstallAction** i modulen **application** som finns i gränssnittet mellan dynamiskt och statiskt bunden kod i skiktet **executive**, se figur 5.3. **InstallAction** sparar en pekare till en procedur som vid anrop lägger till de procedurer som finns i **Ztracker** i **DynamicLinkers** symboltabell.

Tillsammans bildar modulerna som beskrivits i detta kapitel ett fungerande programpaket. Figur 7.3 visar en schematisk bild över hur z-regulatorn fungerar. Modulen **Zregul** innehåller procedurer som startar sensorbaserade rörelser, t.ex. **MoveLinXYtrackZ**. Dessa procedurer anropar **CartGoto** i modulen **Trajec**. **CartGoto** ger order åt **TrajProcess** att beräkna referensvärden till de olika robotservona. Callback-proceduren **Control** i modulen **Zregul** kan ändra referensvärdena med utgångspunkt från mätvärdet från en lasergivare som avläses via proceduren **Distance** i modulen **Laser**. Proceduren **Control** skickar sedan mätdata till modulen **Zplot** som skickar upp dem till Matlab på värddatorn. Trajec återfår kontrollen och skickar de ändrade referensvärdena till robotservona och önskad rörelse utförs.

7.2 Bortslipning av gjutskägg hos gjutgods

Bortslipning av gjutskägg[1] hos gjutgods kan ske genom att röra en slipmaskin längs kanten där gjutskägget finns. Beroende på hur stort gjutskägget är så tar det olika lång tid att slipa bort det. Genom att använda positionsreglering

[1] Gjutskägg uppkommer om det finns skarvar i gjutformen som används vid gjutningen.



Figur 7.3 Schematiskt bild över hur z-regulatorn fungerar.

av slipmaskinen så är det möjligt att registrera om slipningen lyckades eller om och var det behövs en omslipning av gjutgodset.[8]

Jag har inriktat mig på att lösa positionsregleringsproblemet. Någon slipmaskin har jag inte tillgång till utan jag använder istället en robot med lasergivare för avståndsmätning för att mäta storleken hos gjutskägget.

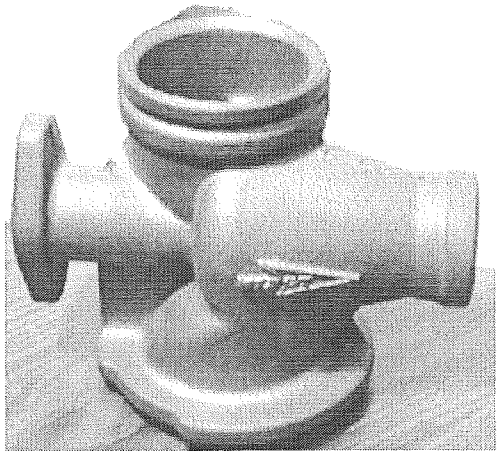
Gjutgodset

Gjutgodset som ser ut enligt figur 7.4 placeras framför roboten. Roboten laddas med den dynamiskt bundna färdigheten Ztracker så att det blir möjligt att utnyttja sensorbaserad reglering i z-led. Roboten ska sedan följa en cirkulär bana runt gjutgodsets översta del. Genom att låta lasergivaren lysa rakt ner på gjutskägget är det möjligt att registrera var gjutskägget är stort och var det är litet.

För att kunna följa en cirkulär bana längs gjutgodsets kant måste gjutgodset exakta position vara känd. Manuell programmering av rörelsen var inte praktisk möjlig p.g.a. att laserstrålen är osynlig. Istället valde vi att låta roboten söka upp kanterna och sedan beräkna gjutstyckets position.

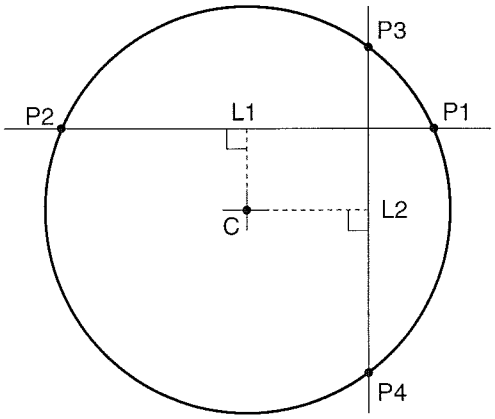
Beräkning av gjutgodsets position

För att kunna bestämma gjutgodsets exakta position antar vi att vi vet dess ungefärliga position. När vi vet detta kan vi använda lasergivaren för att leta upp punkterna **P1** och **P2** (se figur 7.5) genom att röra robotarmen med lasergivare i en rak linje över gjutgodset. På samma sätt hittar vi punkterna **P3** och **P4**. När vi gjort detta kan vi sedan beräkna centrumpunkten för cirkeln genom att normalerna till linjerna **L1** och **L2** skär varandra i centrumpunkten



Figur 7.4 Gjutgodset utseende. Överst syns hur gjutskägget bildar en uppskjutande ojämn kant.

då normalerna utgår från punkterna mitt emellan punkterna **P1** och **P2** resp. punkterna **P3** och **P4**, se figur 7.5.



Figur 7.5 Beräkning av centrumpunkt utifrån kända punkter **P1** till **P4**.

Efter antagande att gjutgodset står horisontellt så kan man i 2-dimensioner beräkna centrumpunkten, **C** enligt formel 7.4.

$$A = \begin{pmatrix} 1 & 0 \end{pmatrix} \left(\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} P2 - P1 & P3 - P4 \end{pmatrix} \right)^{-1} \tag{7.2}$$

$$B = \frac{1}{2}(P3 + P4 - P1 - P2) \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} (P2 - P1) \tag{7.3}$$

$$C = \frac{1}{2}(P1 + P2) + A \cdot B \tag{7.4}$$

När roboten rör sig över gjutgodset för att söka upp de fyra punkterna **P1** till **P4** skickas alla mätdata upp till värddatorn där Matlab tar emot dem. Ett

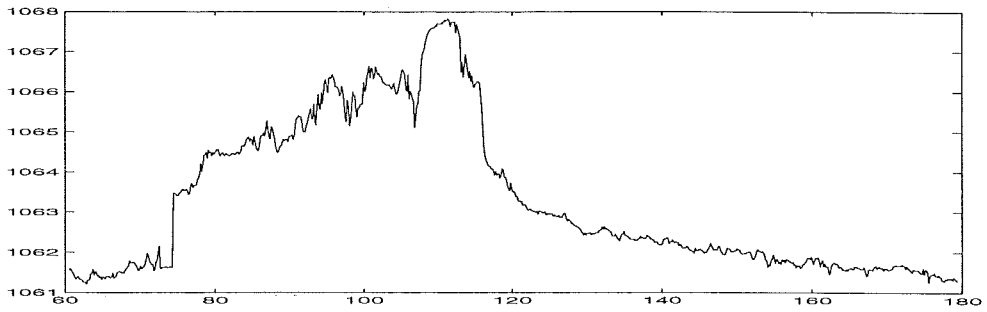
Matlabscript analyserar mätdatan för att urskilja vad de fyra punkterna finns. När punkterna fastställt så används formlerna 7.2, 7.3 och 7.4 för att beräkna centrumpunkten, **C** för cirkeln. Centrumpunktens koordinater skickas sedan ner till målsystemet där de tas emot av robotprogrammet som sedan utför den cirkulära rörelsen över gjutgodset.

Detektering av gjutskägg

När centrumpunkten är beräknad kan man sedan låta roboten följa en beräknad cirkulärbana runt centrumpunkten för att detektera var gjutskägg finns. Detta moment var dock svårare än vad jag trodde. Problemen jag hade var följande:

Svårigheter att detektera punkterna P1 till P4 exakt: Detta berodde på att lasergivaren har svårigheter att detektera svarta ytor (gjutgodset har ganska mörk färg). Det berodde också på oförklarliga variationer av origos position hos robotens koordinatsystem vilket resulterade i att lasergivaren hamnade utanför sitt arbetsområde. Det berodde även på att det finns ett visst glapp i robotens leder. Om inte punkterna **P1** till **P4** får rätt koordinater så blir naturligtvis även den beräknade centrumpunkten, **C** fel.

Svårigheter att få lasergivarens laserstråle exakt över gjutskägget: Detta berodde på att gjutskägget är smalt, ca 1-2 mm, och laserstrålen är mellan 0.45 och 0.9 mm vilket gör att laserstrålen lätt hamnar vid sidan av gjutskägget om gjutskägget lutar eller om centrumpunkten beräknats fel.



Figur 7.6 En del av gjutskägget som detekterats av lasergivaren. Den plottade höjden (given i mm över golvet) mot sträckan utmed gjutgodsets övre kant (given i mm från startpunkten).

Genom att belasta roboten kunde inverkan av glappet i robotens leder minimeras och genom att färga gjutskägget med vit krita var det möjligt att få detekteringen av gjutskägget att fungera. Dock kvarstår problemet med att robotens koordinatsystems origo flyttar sig mellan olika försök. Orsaken till detta kan jag inte förklara utan det blir jag tvungen att lämnar till efterföljande användare av systemet att lösa.

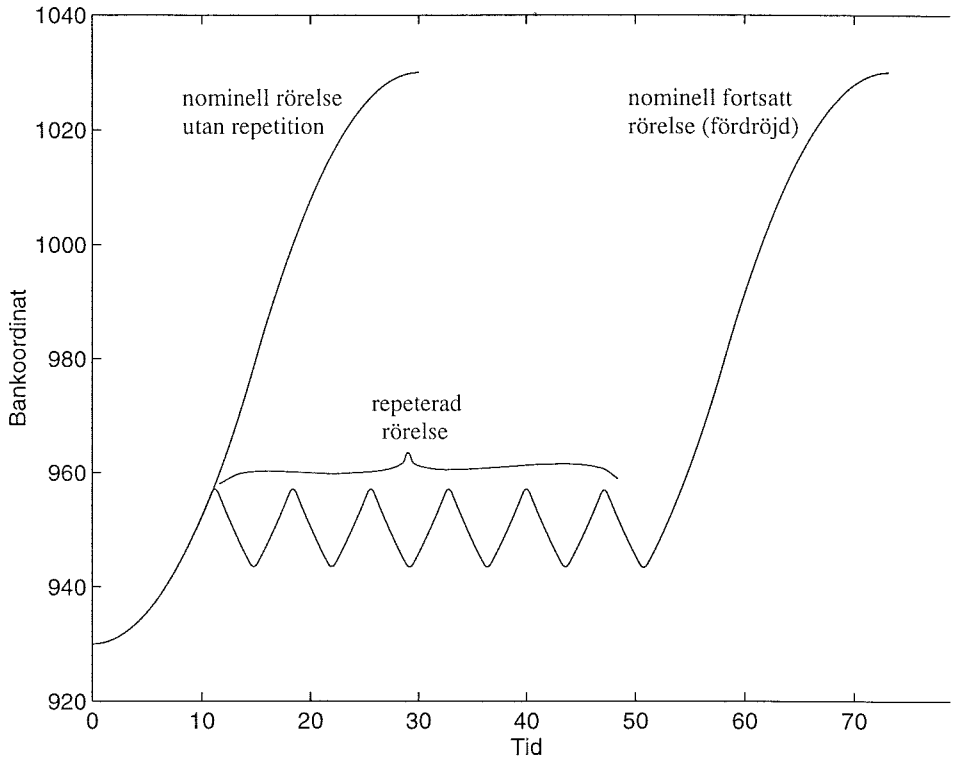
Figur 7.6 visar en bild av hur detekteringen av gjutskägget set ut. Bara den del av som är intressant visas, det stora gjutskägget som syns överst i figur 7.4 är det stora utslaget som syns mellan x-koordinaterna 75 och 115. Figuren är gjord i Matlab.

7.3 Backning och repetition av rörelse

Lasergivarens arbetsområde är relativt litet och det är lätt att hamna utanför arbetsområdet (se kapitel 7.1) om underlaget lutar för kraftigt. För att lösa detta så vill man kunna backa tillbaka roboten och repetera rörelsen när lasergivaren hamnar utanför sitt arbetsområdet. Kapitel 6.4 beskriver hur detta kan implementeras. Här ges nu ett exempel på hur detta kan utnyttjas.

Tillämpning

I föregående exempel "bortslipning av gjutskägg på gjutgods" skulle man kunna använda backning och repetition av rörelsen. När slipmaskinen med konstant hastighet följer den cirkulära rörelsen runt gjutgodsets översta del för att slipa bort gjutskägget (se kapitel 7.2) är det möjligt att gjutskägget är för stort för att det ska kunna slipas bort på den tid som det tar för slipmaskinen att röra sig över gjutskägget. Det skulle då vara möjligt att backa och repetera rörelsen för att slipa bort hela gjutskägget.



Figur 7.7 Robotens rörelse då den stöter på en för brant lutning och använder backning och repetition av rörelse. Y-axeln sammanfaller i detta fall med bankoordinaten som visar den sträcka som roboten rör sig och x-axeln tiden sedan rörelsen påbörjades.

Strategi

Strategin för hur det hela går till är följande. Genom att ändra variabeln **NormStep** i **Context** (se kapitel 6.4) så kan man ändra tiden för nästa steg. När **NormStep** är 1 så tar nästa steg normal tid, om **NormStep** är 0 så tar nästa steg ingen tid alls (roboten rör sig inte) och om **NormStep** är negativ så tar nästa steg negativ tid (roboten rör sig baklänges). När lasergivaren

hamnar utanför sitt arbetsområde så bromsas först roboten in och stannar (**NormStep** minskar stegvis från 1 till 0), sedan börjar den accelerera bakåt (**NormStep** minskar från 0 till -1), sedan arbetar roboten baklänges i normal hastighet (**NormStep** lika med -1), sedan bromsar roboten (**NormStep** ökar från -1 till 0) för att sedan fortsätta som vanligt.

En robot som följer strategin beskriven ovan har implementerats och roboten rör sig enligt figur 7.7, då den stöter på en kant i underlaget som gör att lasergivaren hamnar utanför sitt arbetsområde. Implementationen finns beskriven nedan.

Implementation

Vid implementationen av backning och repetition av rörelse så användes programpaket **Ztracker**, som beskrivits i kapitel 7.1. Implementationen av strategin ovan är gjord i callback-rutinen **ControlWithBacking**, se kapitel 7.1. Eftersom callback-rutiner inte kan lagra information i den egna procedurens lokala variabler använder **ControlWithBacking** i modulen **Zregul** lokala variabler för att komma ihåg om callback-proceduren tidigare kommit utanför sitt arbetsområde och i så fall hur långt i strategin som den kommit. I övrigt fungerar **ControlWithBacking** som callback-proceduren **Control**.

8. Möjliga vidareutvecklingar

Detta kapitel beskriver kortfattat några möjligheter till vidareutveckling av just de delar av systemet som jag använt eller utvecklat. Jag behandlar främst mjukvaruaspekter och inte så mycket hårdvaru- eller reglertekniska aspekter hos systemet.

8.1 Kontrollpanelen

Kontrollpanelen är uppbyggd av block och kan utökas med ytterligare block för nya eller förbättrade funktioner.

Auto

Möjliga vidareutvecklingar av moden **Auto** (exekveringskontroll av robotprogram) är:

- Stega fram flera instruktioner, nu understöds bara att stega fram en instruktion i taget. Målsystemet är förberett för att stega fram flera instruktioner i taget. Hos värddatorn med Matlab krävs att användargränssnittet utökas och att kommunikationen med målsystemet stödet överföring av kommandon med parametrar.

Manual

Möjliga vidareutvecklingar av moden **Manual** är:

- Koordinatmoden Tool finns ännu inte implementerad men det har lämnats plats för den i kontrollpanelen. Implementeras enkelt genom att lägga till ytterligare en uppsättning koordinatvariabler samt en rutin för att konvertera befintliga koordinatsystem till tool-koordinater och tvärt om.

Teach

Möjliga vidareutvecklingar av moden **Teach** är:

- Robotens koordinater visas i parameterboxar som egentligen är gjorda för att ändra parametrar. Det vore därför lämpligt att implementera en annan parameterbox som är mer lämpad för att åskådliggöra robotens position samt för att användaren ska kunna se skillnad mellan de parameterboxar som används i moden **Manual** och det som används i moden **Teach**. Kanske är det tillräckligt att ändra färgen på texten i parameterboxarna.

8.2 Dynamisk bindning

Mellan dynamiskt och statiskt bunden kod så finns ett gränssnitt. Detta gränssnitt är mycket flexibelt och kan utvecklas och ändras. Möjliga vidareutvecklingar av dessa gränssnitt och omgivande struktur är:

- Avallokering av de delar av den dynamiskt bundna koden som inte längre används av systemet. Som det fungerar nu ligger den dynamiskt bundna koden kvar även då den inte används längre.
- Automatisk medlänkning av de färdigheter som det dynamiskt bundna robotprogrammet kräver. Som det fungerar nu så måste användaren själv se till att de färdigheter som robotprogrammet använder finns inladdade i systemet.

8.3 Sensorbaserade rörelser

De sensorbaserade rörelser som jag implementerat är ett kraftfullt redskap vars möjligheter jag bedömer som mycket stora. Utveckling av kraftfulla sensorbaserade callback-rutiner kan därför bli en intressant tillämpning av systemet. Möjliga vidareutvecklingar är:

- Implementering av sensorbaserade rörelser för ledvinkelkoordinater. Nu finns sensorbaserade rörelser implementerat för kartesiska koordinater och denna implementation bör användas som modell för eventuell implementation för ledvinkelkoordinater.
- Införande av ytterligare en inparameter till callback-procedurerna. Den nya inparametern kan innehålla en specifikation som talar om vad callback-proceduren ska utföra. På detta vis kan man göra betydligt mer generella callback-procedurer. Nuvarande callback-procedurer ser ut på följande vis:
  `f(serverkontext)`
  medan de nya callback-procedurerna skulle se ut enligt:
  `f(serverkontext, klientkontext)`
  Inparametern **klientkontext** är en pekare på en datatyp som användaren själv kan skapa och kan användas för att instruera callback-proceduren vad den ska utföra. Detta skulle göra callback-procedurerna betydligt mer flexibla än de nuvarande, där det oftast krävs en specifik callback-procedur för varje tillämpning.

8.4 Fjärrstyrning

En annan intressant vidareutveckling är att låta fjärrstyra roboten via internet. Möjligheten finns att använda Netscape som klient hos användaren och ett cgi-script till http-servern som kontinuerligt skickar uppdatering av bilder på roboten. På detta vis kan användaren få kontroll över hur roboten rör sig.

Universitet vid Ulm i Tyskland har utvecklat en programvara, *WebVideo - WWW live video on demand*, som understöder animerade bilder. Information om programvaran finns att hämta på följande adress:
`http://www-vs.informatik.uni-ulm.de/soft/wv/`
Programvaran kan än så länge laddas hem kostnadsfritt.

För att styra roboten kan ett formulär implementeras i språket html. Från klienten är det möjligt att från ett formulär skicka information till en http-server. På http-server finns ett cgi-script som tar emot informationen och vidareförmedla den till robotsystemet.

9. Sammanfattning och kommentarer

Sammanfattning

Under de senaste åren har flera projekt utvecklat robotmjukvaran för olika tillämpningar. Flera av projekten har drivits parallellt vilket resulterat i att programkoden divergerat. För att lösa detta har därför resultaten från de parallella projekten sammanfogats till ett komplett system.

Sedan implementerades en kontrollpanel med grafiskt användargränssnitt i Matlab. Från kontrollpanelen är det möjligt att styra roboten manuellt eller med en styrboll. Den innehåller också en exekveringskontroll för robotprogram. Exekveringskontrollen gör det möjligt att starta, stoppa och stega igenom ett robotprogram. Kontrollpanelen är objektorienterad uppbyggd trots att Matlab saknar egentligt stöd för objektorientering.

Därefter utvecklades ett gränssnitt som använder dynamiskt bindning av exekverbarkod med möjlighet att ladda in både nya robotprogram och nya färdigheter till systemet utan att för den delen bli tvungen att starta om hela systemet. Gränssnittet är flexibelt och enkelt att utöka och ändra. Gränssnittet tillåter även dynamiskt bundna robotprogram att använda dynamiskt bundna färdigheter hos systemet.

Systemet har även modifierats så att det nu är möjligt att använda sensorbaserad robotstyrning. Styrningen sker i callback-rutiner som ändrar till roboten beräknade referensvärden innan de skickas till robotens servon. Eftersom man inte kan tillåta en callback-rutin att ändra robotens referensvärden hur mycket som helst sker en kontroll av de modifierade referensvärdena innan de skickas till robotens servon.

Slutligen så har flera exempel på hur systemet kan användas med mina nya tillskott i programvaran implementerades. En dynamiskt bunden sensorbaserad Z-regulator, som kan reglera robotens position i höjdled så att robotarmen kan följa en underliggande yta, har implementerats. Sedan har ett program för att detektera gjutskägg hos ett speciellt gjutgods implementerats. Dessutom har ett exempel där roboten backar och repeterar en rörelse under vissa premisser implementerats.

Erfarenheter och slutsatser

En av de viktigaste erfarenheterna jag dragit ur detta projektet är att lära mig tolka andra människors okommenterade eller felkommenterade programkod. Att tolka andra människors programkod är väldigt tidsödande. Slutsatsen är att man alltid bör kommentera sin kod väl.

Matlab har i detta examensarbete används för att skapa användargränssnitt. Detta innebär dock vissa begränsningar, Matlab är långsamt och saknar egentligt stöd för objektorientering. Det finns ingen naturlig hierarki mellan olika objekt på skärmen utan detta måste programmeraren själv ordna om han vill ha ordning och reda.

Kommunikationen mellan värddatorn och målsystemet sköts av MatComm och har inte varit direkt problemfri. Kommunikationen har en förmåga att brytas efter en tid. Om problemet ligger hos mjukvaran eller hårdvaran är

inte klarlagt. Den enda slutsatsen jag kan dra är att nätverksproblemen sätter i nuläget gränsen för hur länge roboten kan användas utan omstart.

Metoden för att dynamiskt binda kod som utnyttjats har fungerat mycket väl. Metoden ger det öppna robotstyrsystemet möjligheten att under drift byta ut delar av programkoden för att anpassa robotsystemet till olika tillämpningar. De nya gränssnitt som jag skapat för nedladdning av kod på olika nivåer har tillfört systemet dels möjligheten att från dynamiskt bundna robotprogram använda dynamiskt bundna färdigheter, och dels ett skydd som begränsar vilka operationer användaren tillåts utföra från dynamiskt bunden kod.

Mycket arbete har även lagts på sensorbaserad rörelsestyrning. Lasergivaren som finns monterad på robotens gripdon har använts för att mäta avståndet till underlaget. Lasergivaren har emellertid relativt begränsat arbetsområde vilket har orsakat mycket problem. För att lösa lasergivarens ofullkomlighet har olika kontroller implementerats utan att uppnå ett heltäckande funktionalitet. För att erhålla bättre funktion krävs en funktionsdugligare givare eller kanske flera givare men olika arbetsområden och känslighet.

Litteraturförteckning

[1] R. Braun, L. Nielsen, and K. Nilsson. "Reconfiguring an ASEA IRB-6 robot system for control experiments." Report TFRT-7465, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, October 1990.

[2] M. Karlsson. "Implementering av experimentellt robotstyrsystem," (Implementation of an experimental robot control system). Master thesis ISRN LUTFD2/TFRT--5475--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, November 1993.

[3] O. Laurin. "Öppna regulatorer för inbyggda system," (Open controllers for embedded systems). Master thesis ISRN LUTFD2/TFRT--5528--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, April 1995.

[4] The MathWorks, Inc., Natick, MA. *MATLAB, Building a Graphical User Interface.*, June 1993.

[5] L. Nielsen, L. Andersson, M. Andersson, and K.-E. Årzén. "A real-time kernel with graphics support modules." Report ISRN LUTFD2/TFRT--7510--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[6] L. Nielsen and K. E. Årzén. "Computer implementation of control systems." Report ISRN LUTFD2/TFRT--7476--SE, August 1995.

[7] K. Nilsson. *Application Oriented Programming and Control of Industrial Robots.* Lic Tech thesis ISRN LUTFD2/TFRT--3212--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, June 1992.

[8] K. Nilsson. *Industrial Robot Programming.* PhD thesis TFRT-1046, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1996.

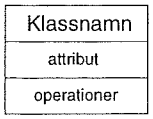
[9] K. Nilsson, A. Blomdell, and O. Laurin. "Open embedded control." *Submitted to: Real-Time Systems – The international journal of time critical computing*, 1996.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[11] J. Sonnerfeldt. "Matlab som 'compute server' för inbyggda system – beräkning av robotrörelser," (Matlab as a compute server for embedded systems—trajectory computation). Master thesis ISRN LUTFD2/TFRT--5527--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, February 1995.

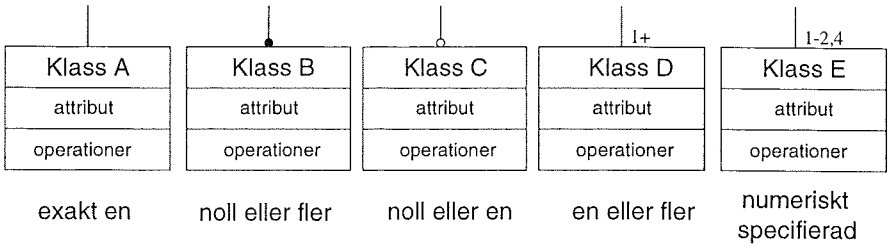
A. Objektdiagram

Detta appendix beskriver hur den notationen för objektdiagram enligt OMT som används i kapitel 4 ska tolkas. För mer utförlig beskrivning om OMT hänvisar jag till *Object-oriented modeling and design* [10].



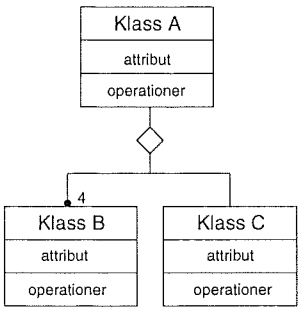
Figur A.1 Ensam klass med attribut och operationer

Figur A.1 visar hur en klass ser ut. Överst står klassnamnet där under står alla väsentliga attribut och under den står de viktigaste operationerna nedtecknade. Detta är grundklossen i ett objektdiagram. Ett objektdiagram består av en samling klasser som är sammanbundna med olika associationer. Associationer åskådliggörs genom att klasserna sammanbinds med linjer.



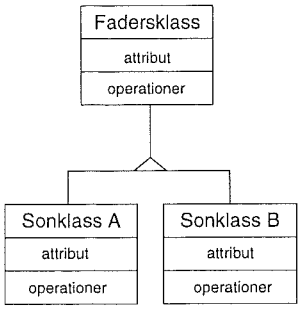
Figur A.2 Multiplicitet hos associationer

Figur A.2 visar hur multiplicitet hos associationer kan beskrivas. Finns det exakt en uppsättning av klassen behövs inget speciellt markeras. Kan det finnas noll eller fler uppsättningar av klassen så markeras det med en fylld ring. Kan det bara finnas noll eller en uppsättning av klassen så markeras det av en ring som inte är fylld. När det gäller mer avancerade förhållande markeras detta genom att skriva vilka förhållanden som gäller.



Figur A.3 Aggregation

Figur A.3 visar hur aggregationer ser ut. Aggregation är en speciell sorts association som fått ett eget namn. Aggregation används när en klass består eller är uppbyggd av objekt från andra klasser. T.ex. en bil är uppbyggd av en motor och fyra hjul. Detta förhållande är en aggregation. Detta förhållande ska inte blandas ihop med ärvning.



Figur A.4 Ärvning

Figur A.4 visar hur ärvning åskådliggörs i ett objektdiagram. Med ärvning menas att en klass kan ärva en annan klass egenskaper. En fadersklass kan ha flera sonklasser (söner). Ett exempel kan vara att klassen fordon kan ha sonklasserna bil, buss och lastbil. Om det ska vara frågan om ärvning ska man kunna säga att bil är ett fordon, buss är ett fordon och lastbil är ett fordon. Ärvning har inte använts någonstans i rapporten, men eftersom det är ett så viktigt begrepp inom objektorientering har jag valt att ändå förklara begreppet för att undvika missförstånd.

B. Matlabs grafiska användargränssnitt

Matlab 4.2 har ett inbyggt grafiskt användargränssnitt (GUI) med vilket det är möjligt att skapa fönster, knappar, menyer mm. Detta appendix beskriver kort hur detta GUI fungerar. För utförligare beskrivning hänvisar jag till *Matlab, Building a Graphical User Interface* [4].

B.1 Fönster

Kommandot **figure** skapar ett nytt grafiskt fönster och returnerar ett handtag till fönstret i form av ett heltal. **Figure(h)** gör det **h**te fönstret aktivt. Om fönstret **h** inte finns så skapas det.

B.2 Grafiska objekt

Detta stycke behandlar de olika typer av grafiska objekt som kan skapas. Matlab tillhandahåller åtta olika grafiska objekt. Tabell B.1 beskriver kortfattat de olika grafiska objekten som finns att tillgå i Matlab.

Tabell B.1 Grafiska objekt som understöds av Matlab.

Typ	Beskrivning
Push button	Tryckknapp som utför ett kommando
Check box	Låter användaren att välja en eller flera inställningar
Radio button	Genom ömsesidig uteslutning kan användaren välja ett av flera alternativ
Slider	Låter användaren välja mellan ett värde inom ett givet intervall
Pop-up Menus	Låter användaren välja mellan olika alternativ
Static text	Visar en rad med text som inte kan ändras
Editable text	Visar ett fält med redigerbar text
Frame	Visar en rektangulär ram runt ett eller flera grafiska objekt

Alla grafiska objekt har ett antal attribut, se tabell B.2. Dessa attribut talar om vilken typ det grafiskt objekt är, vilken färg de har mm. Attributet **UserData** är reserverat till de data som programmeraren önskar lagra i objektet. **UserData** används således för att lagra information som behövs när användaren aktiverar objektet under användningen. Det kan t.ex. vara handtag till andra objekt som ska påverkas om användaren ändrar någon inställning hos objektet.

För att skapa grafiska objekt används funktionen **uicontrol**, som anropas med följande inparametrar: först anges handtaget till det grafiska fönster som objektet ska skapas i, sedan anges en lista som innehåller *'attributnamn', attributvärde*. Det första attributet som anges är vilken typ av objekt som ska skapas. Därefter kan ett godtyckligt antal attribut tilldelas värden. Attribut som inte tilldelas något värde får ett förinställt värde. **Uicontrol** returnerar ett handtag till det nya objektet. För att ta reda på och ändra objekts attribut

används kommandona **get** och **set** med objektets handtag som inparameter. Följande exempel visar hur en **Push button** skapas. Knappen placeras i positionen (10, 10), där (0,0) finns längst ner till vänster på skärmen. Knappen får storleken x=100 punkter och y=25 punkter, det kommer att stå skrivet **Start Plot** på knappen samt funktionen **myplot** kommer att startas när knappen trycks ner.

```
pbstart = uicontrol( gcf, ...
                     'Style', 'push', ...
                     'Position', [10 10 100 25], ...
                     'String', 'Start Plot', ...
                     'CallBack', 'myplot' );
```

Tabell B.2 De attribut som finns till grafiska objekt.

Attribut	Beskrivning
BackgroundColor	Specificerar färgen som ska fylla den rektangel som det grafiska objektet utgör
ButtonDownFcn	Specificerar vad som ska göras när användaren klickar med musen på objektet
Callback	Specificerar vad som ska ske när objektet aktiveras
Children	Pekare till eventuella barn (tom eftersom objekt inte har barn)
ForegroundColor	Specificerar färgen hos texten som visas i objektet
HorizontalAlignment	Specificerar positionen för objektets titel
Interruptable	Specificerar om callback-rutinen är avbrytbar
Max	Specificerar det högsta värde som attributet Value kan anta
Min	Specificerar det lägsta värde som attributet Value kan anta
Parent	Handtag till objektets förälder (figuren som objektet visas i)
Position	Specificerar placering och storlek av objektet
String	Definierar den titel som kan visas i objektet
Style	Talar om vilken typ objektet är
Type	Talar om vilken grupp av objekt som objektet tillhör (uicontrol)
Units	Specificerar enheten för de värden som angetts i attributet Position
UserData	Specificerar en matris som kan användas för att lagra lokala variabler
Value	Specificerar det aktuella värdet för objektet
Visible	Indicerar om objektet ska visas på skärmen eller ej

B.3 Menyer

Matlab innehåller även funktioner för att skapa rullgardinsmenyer. Funktionen **uimenu** används för att skapa en meny. Följande inparametrar används: först ett handtag till figuren som menyn ska vara i, sedan en lista med *attribut, attributvärde* på liknande sätt som hos de grafiska objekten.