DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

MASTEROPPGAVE

Studieprogram/spesialisering: Informasjonsteknologi	Vårsemesteret, 2016 **Åpen**
Forfatter: Jan Marius Vasshus	……………M. Vasshus………………… (signatur forfatter)
Fagansvarlig: Morten Mossige Veileder(e): Morten Mossige, Ståle Freyer	
Tittel på masteroppgaven: Forenklet programmering mot ABB-robot Engelsk tittel: Simplified programming against ABB robot	
Studiepoeng: 30	
Emneord: DATMAS, ABB, Robot, Robot Web Services, PC SDK, API, Python, IronPython, REST	Sidetall: 71 + vedlegg/annet: 1 Stavanger, 13 juni, 2016. dato/år

Forenklet programmering mot ABB-robot

Jan Marius Vasshus

Våren 2016

Institutt for data- og elektroteknikk

Universitetet i Stavanger

Oppgavetekst

Oppgaven går ut på å utvikle et API som gjør det enklere å lage PC programmer som kommuniserer med en ABB-robot. Kommunikasjon med robot skal utvikles med følgende to grensesnitt:

1. PC SDK, som er nært knyttet til .NET teknologien.
2. Robot Web Services, som bruker HTTP.

Disse grensesnittene skal testes mot hverandre med hensyn til synkronisering og oppkoblingstider, i tillegg til å beskrive styrker og svakheter.

Forord

Denne oppgaven markerer slutten av mitt mastergradsstudie i informasjonsteknologi ved Universitetet i Stavanger. Oppgaven har vært utfordrende, lærerik og spennende.

Takk til Morten Mossige og Ståle Freyer for god og konstruktiv veiledning gjennom hele prosessen.

Jeg vil også takke ABB for hjelpen jeg har fått og deres gjestfrihet på Bryne. I tillegg vil jeg takke min familie for støtten jeg har fått gjennom våren 2016.

Stavanger, 2016-06-13

Jan Marius Vasshus

Oppsummering

Kommunikasjon med ABB-robot utføres gjennom to grensesnitt, PC SDK og Robot Web Services. Disse grensesnittene er utviklet av ABB. Grensesnittene er store og inneholder mye funksjonalitet. Det kan være vanskelig å finne frem i grensesnittene, og vanskelig å vite hvordan programmeringen skal utføres. Dette fører til at de er tidkrevende å bli kjent.

Denne oppgaven har som mål å forenkle programmering mot ABB-roboter ved å implementere et API som er bygget opp av enkle funksjonskall. Funksjonene skal utføre kommandoer på robot som for eksempel å koble til og logge på en bruker. API-et skal gjøre det enklere å lage programmer, noe som igjen er tidsbesparende. Det er ikke nødvendig å ha kunnskap om PC SDK eller Robot Web Services.

Det skal kommuniseres med robot gjennom to grensesnitt. På grunn av dette er det valgt å implementere to API-er i stedet for det ene. Et for hvert grensesnitt. Dette fører til at det ene kan utelates om det ikke er brukt, noe som gir bedre oversikt og skaper mindre forvirring. IronPython er brukt til å implementere API-et som kommuniserer gjennom PC SDK. IronPython åpner for å kunne bruke .NET-rammeverket i Python. PC SDK er bygget på .NET-rammeverket. Python er brukt til å implementere API-et som kommuniserer gjennom Robot Web Services. For å bruke API-ene er det nødvendig med grunnleggende Python programmeringsferdigheter fordi de er bygget rundt funksjonskall. I oppgaven blir det gjennomgått hvordan API-ene er implementert, i tillegg til hvordan implementasjonen er verifisert. Verifiseringen inneholder blant annet statisk kodeanalyse av API-ene.

Det er utført tester for å finne forskjeller mellom API-ene. Tester som er utført inkluderer blant annet oppkoblingstid, responstid og koblingspålitelighet. Det er også utført en SWOT-analyse (styrker, svakheter, muligheter, trusler) for hvert API. De resulterende API-ene forenkler kommunikasjon med ABB-robot gjennom enkle funksjonskall. Kravene til brukeren bestemmer hvilket API som egner seg best. API-et som kommuniserer gjennom PC SDK er låst til Windows og har problemer med tredjepartsbiblioteker, men er det mest robuste ut i fra testingen. Det andre API-et som kommuniserer gjennom Robot Web Services er plattform-uavhengig og har god støtte for tredjepartsbiblioteker. Testingen viste at dette API-et ikke er like robust.

Innholdsfortegnelse

Oppgavetekst ... 1
Forord ... 2
Oppsummering ... 3
1. Introduksjon ... 6
2. Relatert arbeid ... 7
3. Bakgrunn ... 8
3.1 Simulerings- og programmeringsverktøy til robot ... 8
3.2 Robotens kontrollenhet ... 11
3.3 Programvaren til en kontroller ... 13
3.4 Programmeringsspråket som er brukt til å kontrollere robot ... 13
3.5 Kommunikasjon med robot gjennom PC SDK ... 17
3.6 Aksessere .NET i Python ... 17
3.7 Kommunikasjon med robot gjennom Robot Web Services ... 18
4. Konstruksjon ... 20
4.1 Oppsett av virtuell robot i RobotStudio ... 21
4.2 API som bruker PC SDK ... 22
4.3 API som bruker Robot Web Services ... 40
5. Resultat ... 57
5.1 Oppkoblingstider ... 57
5.2 Inaktivitetstest ... 58
5.3 Koblingspålitelighet ... 59
5.4 Responstider ... 60
5.5 Kjøretider til blomsterprogram ... 63
6. Diskusjon ... 65
6.1 Diskusjon av resultatene ... 65
6.2 Analyse av API-ene ... 67
6.3 Videre arbeid ... 70
7. Konklusjon ... 71
Referanser ... 72
Figurer ... 75
Tabeller ... 76

Kildekode 77
A. Vedlegg 78

1. Introduksjon

Robotteknologien har i de siste 50 årene hatt enorme fremskritt. I dag er roboter presise verktøy som kan utføre en mengde oppgaver som lakkering, sveising og lasting. Utviklingen til robotene har fulgt datamaskinens utvikling, noe som har vært med på å gjøre robotene til det de er i dag. Utviklingen av datamaskinene har åpnet for muligheten til å kommunisere med roboter via en datamaskin.

Denne avhandlingen bruker en robot av typen *IRB_140 6kg 0.81m* til testing. Navnet tilsier at roboten har håndteringskapasitet på seks kilogram og en rekkevidde på 810 mm. Det er en robot med seks akser som kan monteres på gulvet, opp ned eller på en vegg. Dermed er det en fleksibel robot.

ABB har utviklet to grensesnitt som brukes til å lage programmer som kommuniserer med robot. Grensesnittene er store og inneholder mye funksjonalitet. Det ene grensesnittet heter PC SDK og det andre Robot Web Services. Robot Web Services er nytt og ble sluppet i den siste store oppdateringen fra ABB. Denne avhandlingen går ut på å utvikle et API som forenkler kommunikasjon med robot og bruker begge grensesnittene.

Det kan være vanskelig for nye og eksisterende brukere å sette seg inn i grensesnittene fordi de inneholder mye funksjonalitet. Denne avhandlingen avgrenses til å ha hovedfokus på å behandle variabler i programmeringsspråket RAPID. RAPID brukes til å kontrollere ABB-roboter. Muligheten til å lage programmer åpnes ved å kunne behandle variabler i *RAPID*.

Kapittel to presenterer arbeid som er gjort på området tidligere. I kapittel tre presenteres bakgrunnsinformasjon som er nødvendig for å forstå utførelsen av oppgaven. Det blir blant annet gjennomgått sentrale temaer som simuleringsverktøy, programmeringsspråk og teknologier som er brukt. Konstruksjonen av API-et blir gjennomgått i kapittel fire. Det vil bli forklart prinsipper som er brukt i utviklingen og hvordan utviklingen ble utført. Kapittel fem forklarer testene som ble utført på API-et med en presentasjon av resultatet. I kapittel seks blir resultatene og API-et diskutert i tillegg til en SWOT-analyse. Det siste kapittelet presenterer konklusjonen som baseres på resultatene og analysen i kapittel fem. Flere engelske ord er brukt i oppgaven da det ikke er funnet gode norske ord for uttrykkene.

2. Relatert arbeid

Det å forenkle programmering mot ABB-robot er forsøkt løst før ved Universitetet i Stavanger. Et grensesnitt er utviklet med programmeringsspråket MATLAB. Dette grensesnittet har vist seg å være ustabilt. Oppkoblingen er upålitelig, og uventede feil har forekommet under kjøring. Brukere har også utrykt at det kan være vanskelig å bruke grensesnittet. Kommunikasjonen med robot utføres med PC SDK, og begrenser seg til å behandle RAPID-delen på roboten.

Denne oppgaven baserer seg ikke på det foregående arbeidet, bortsett fra at begrensningen også her er RAPID-delen av roboten.

3. Bakgrunn

3.1 Simulerings- og programmeringsverktøy til robot

RobotStudio er et simulerings- og programmeringsverktøy som gjør det mulig å programmere og teste mot reelle og virtuelle roboter. Programmet gjør det mulig å trene, programmere og optimalisere uten å være nær en reell robot. Det er dermed ikke nødvendig å avbryte roboter som er del av en produksjon. Programvaren er laget for blant annet å redusere risikoen som er forbundet med å være nær roboter, samt redusere tiden det tar å endre i systemet og gi økt produktivitet med robot. [1]

RobotStudio bygger på *ABB VirtualController* (delkapittel 3.2) som er en eksakt kopi av programvaren som er på en reell robot. På grunn av dette er det mulig å få realistiske simuleringer med virtuelle roboter. [1]

Delpunktene under beskriver begreper som brukes ved oppsett av en robot i RobotStudio.

3.1.1 Stasjon (Station)

RobotStudio bruker ordet stasjon som en samlebetegnelse for alle komponentene som er nødvendige for programmering og simulering. [2]

Alle operasjoner som blir utført av en bruker i RobotStudio blir lagret til en stasjon. Operasjoner kan være det å legge til eller opprette komponenter som for eksempel rullebånd, bord og verktøy. [2]

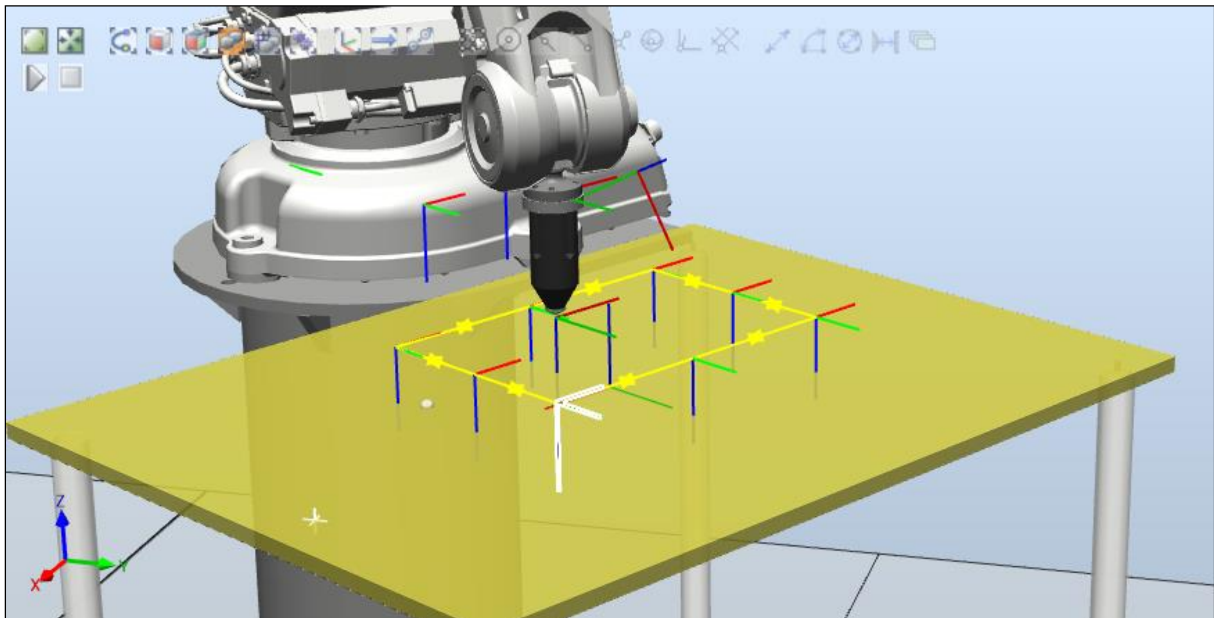
3.1.2 Robotsystem

Robotsystem er noe av det første som blir lagt til en stasjon. Når det legges til et nytt robotsystem så må det spesifiseres hvilken robottype som skal brukes. I tillegg må det spesifiseres hvilken versjon av RobotWare (delkapittel 3.3) som skal kjøres på kontrolleren til robot. [3]

Figur 1 viser et robotsystem. Roboten står i dette tilfellet på en sokkel.

3.1.3 Arbeidsobjekt (Workobject)

Et arbeidsobjekt er et koordinatsystem brukt til å beskrive posisjonen til et arbeidsstykke. Arbeidsstykke er det roboten arbeider på. I figuren under er bordet et arbeidsobjekt. [4]



Figur 1: Bilde av en stasjon bestående av en virtuell robot.

3.1.4 Ramme (Frame)

Ramme er et synonym for koordinatsystem. [5]

En stasjon har grunnleggende rammer, i tillegg til rammer som blir opprettet i det komponenter legges til. Under nevnes noen av koordinatsystemene.

Globalt koordinatsystem:

Hver stasjon har et globalt koordinatsystem. Dette er det grunnleggende koordinatsystemet som alle andre koordinatsystemer relaterer til. Det representerer hele stasjonen. [6]

Base koordinatsystem:

Hver robot har et base koordinatsystem som er lokalisert i basen til roboten. [6]

Verktøysenterpunkt koordinatsystem:

Et verktøy har et koordinatsystem i senter av verktøyet. Roboten beveger senteret av verktøyet til en programmert posisjon. For mer informasjon om verktøy se delkapittelet nedenfor. [6]

Arbeidsobjekt koordinatsystem:

Arbeidsobjekt har to koordinatsystemer, brukerramme og objektramme. Målene som blir programmert blir relatert til objektrammen til arbeidsobjektet. I figuren over vil brukerrammen være koordinatsystemet til bordet og arbeidsstykke vil være objektrammen. [6]

Stasjonvinduet i RobotStudio viser fram rammene. De rammene som ikke vises kan sees ved å trykke på det respektive objektet.

3.1.5 Verktøy (Tool)

Verktøy er det som blir montert på robot for å utføre et arbeid. Det kan for eksempel være en penn som i figur 1. I RobotStudio er det mulig å lage nye verktøy eller importere verktøy som allerede er laget. For å kunne simulere med et verktøy må verktøydata (tooldata) være definert. Verktøydata blir opprettet automatisk ved importering eller opprettelse av nytt verktøy. [7]

3.1.6 Mål (Target)

Et mål er et punkt som er definert i et koordinatsystem. Hensikten med målet er at roboten skal kunne forflytte seg til det. [8]

Målet inneholder informasjon som roboten trenger når den skal forflytte seg til målet.

Figur 1 viser en stasjon med mange mål definert over et bord. Hvert mål har en x, y og z akse (rød, grønn og blå linje). Målet ligger der disse møter hverandre.

3.1.7 Sti (Path)

En sti er en sekvens med forflytningsinstruksjoner. For at roboten skal kunne forflytte seg mellom en sekvens av mål så må en sti være definert. Hver forflytningsinstruksjon inneholder en referanse til målet, bevegelsesdata som hastighet, en referanse til verktøyet og en referanse til arbeidsobjektet. [9]

Figur 1 viser hvordan en sti ser ut i RobotStudio. Stien er det gule rektangelet på bordet og består av mange mål.

3.2 Robotens kontrollenhet

Kontrolleren er robotens kontrollenhet. Det finnes to typer kontrollere; reelle og virtuelle. De har begge sine bruksområder. Reelle kontrollere brukes i produksjon med reelle roboter, mens virtuelle kontrollere blir brukt til testing av programmer på virtuelle roboter. [1]

I RobotStudio er det mulig å utføre operasjoner på både reell og virtuell kontroller. En virtuell kontroller blir automatisk opprettet når en ny stasjon lages. Om det er ønskelig å jobbe med en reell kontroller så kan det det gjøres ved å koble til en kabel eller ved å søke etter kontrolleren over nettverket. [10]

Valgmulighetene som er tilgjengelige i RobotStudio varierer basert på om det er tilkobling til reell kontroller eller virtuell. Med en reell kontroller støttes filoverføring, spørre om skrivetilgang, slippe skrivetilgang, innlogging av brukere osv. Ved tilkobling til en virtuell kontroller er det ikke alltid nødvendig med de samme funksjonene. For eksempel støttes ikke filoverføring da den virtuelle kontrolleren har tilgang til det som er laget i RobotStudio. Det som er nødvendig for en virtuell kontroller er for eksempel å ha tilgang til kontrollpanelet som finnes på en reell kontroller. RobotStudio har dette panelet tilgjengelig for virtuell kontroller.

3.2.1 Reell kontroller

IRC5 (Industrial robot controller) er per i dag den nyeste kontrolleren til ABB. [11]
Den har standard maskinvare som kort skal beskrives under. Dette vil gi en bedre forståelse for hva en kontroller er og hva som er inkludert i en kontroller.

Kontrollmodul (Control module):

Kontrollmodulen inneholder hoveddatamaskinen som kontrollerer bevegelsen til roboten. Dette inkluderer RAPID (delkapittel 3.4) utførelse og signal håndtering. [12]
RAPID er programmeringsspråket som er brukt til å kontrollere roboten.

Kjøremodul (Drive module):

Kjøremodulen inneholder elektronikken som gir strøm til robotens motorer. [12]

FlexController:

FlexController er kontroller-kabinettet til en IRC5 robot. En FlexController består av en kontrollmodul og en kjøremodul for hver robot i systemet. [12]

Disse komponentene utgjør en IRC5 kontroller.

Figur 2: IRC5 kontroller.

3.2.2 Virtuell kontroller

En virtuell kontroller er programvare som emulerer en FlexController. Dette gjør det mulig å kjøre samme programvare til å kontrollere roboter på en PC. Dermed er det samme adferd på robot uten å være oppkoblet. [5]

Figur 3: Kontrollpanelet til en virtuell kontroller.

3.3 Programvaren til en kontroller

Programvaren som kjører på kontroller heter RobotWare. RobotWare refererer til både programvaren som blir brukt til å lage et RobotWare-system, i tillegg til selve RobotWare systemene. Et RobotWare-system er et sett med programfiler som lastes opp på en robot-kontroller. Disse filene gjør at alle funksjonene, konfigurasjonene, data og programmer som styrer robotsystemet blir tilgjengelige. [13]

Det er viktig at RobotWare-versjonen som kjøres på robot-kontroller samsvarer noenlunde med den som det utvikles for i RobotStudio slik det ikke blir noen store forskjeller i det som er støttet.

3.4 Programmeringsspråket som er brukt til å kontrollere robot

RAPID er programmeringsspråket som er brukt til å kontrollere ABB-roboter. RobotStudio har en innebygget editor som brukes til å skrive programmer i RAPID. [14]

RobotStudio gjør det mulig å synkronisere objekter fra stasjonen til RAPID-kode og omvendt. For eksempel når et mål blir opprettet i stasjonen så kan dette synkroniseres til RAPID-kode og brukes i et program. Det samme gjelder for eksempel sti, verktøy og arbeidsobjekt. Når et program er ferdig utviklet så kan det testes på virtuell robot i RobotStudio. For å teste det på en reell robot så kan RobotStudio brukes til å laste programmet opp på reell kontroller og deretter kjøres.

Resten av delkapittelet skal beskrive programmeringskonsepter og relevante RAPID-datatyper og deres hensikt. Det som menes med relevante RAPID-datatyper er datatypene som det er laget støtte for i implementasjonen som gjennomgås i kapittel fire. Kun *robtarget* blir gjennomgått mer detaljert som et eksempel. Det blir referert til en utfyllende beskrivelse for resten av datatypene.

3.4.1 Programmeringskonsepter

Tilkoblet programmering (Online programming)

Tilkoblet programmering vil si å være tilkoblet kontrollmodulen når programmering utføres. Det som ligger i begrepet er at roboten blir brukt til å lage posisjoner (mål) og bevegelse (sti). [5]

Frakoblet programmering (Offline programming)

Frakoblet programmering vil si å programmere uten å være tilkoblet roboten eller kontrollmodulen. Da er det ikke mulig å teste fortløpende. [5]

Sann frakoblet programmering (True offline programming)

Sann frakoblet programmering vil si å koble et simuleringsmiljø til en virtuell kontroller. Dette gir mulighet til å lage programmer i tillegg til å teste og optimalisere uten å være tilkoblet. [5]

3.4.2 Robtarget

Robtarget er brukt til å definere en posisjon som roboten skal kunne gå til. Dermed er robtarget det som kalles posisjonsdata. [15]

I delkapittelet 3.1.6 er det forklart om mål i RobotStudio. Robtarget blir opprettet når et mål lages i RobotStudio. Når det synkroniseres fra stasjonen til RAPID så vil et robtarget bli definert med tilhørende posisjonsdata. [9]

Robtarget sine egenskaper blir gjennomgått her:

Trans

Trans er posisjonen (x, y, z). Posisjonen er definert ut fra arbeidsobjektets koordinatsystem. Om det ikke er spesifisert et arbeidsobjekt så blir posisjonen relatert til globalt koordinatsystem. [15]

Rot

Rot spesifiserer orienteringen til målet. Verktøyets midtpunkt vil ha samme orientering som målet. Orienteringen er definert ut fra arbeidsobjektets koordinatsystem. Dersom et arbeidsobjekt ikke er spesifisert blir globalt koordinatsystem brukt. [15]

Robconf

Robconf er aksekonfigurasjonen roboten må ha for å nå målet. [15]

Extax

Extax er posisjonen av eksterne akser. Om akser ikke er sammenkoblet vil de ha verdien 9E9 og bli ignorert. [15]

3.4.3 Wobjdata

Wobjdata er brukt til å beskrive et arbeidsobjekt som roboten for eksempel beveger seg innenfor. [16]

Når et arbeidsobjekt blir opprettet i RobotStudio og synkroniserer til RAPID, så blir wobjdata laget.

Wobjdata sine egenskaper blir beskrevet her [16].

3.4.4 Tooldata

Tooldata er brukt for å beskrive karakteristikken til et verktøy. [17]

Tooldata vil bli opprettet når et verktøy blir laget eller importert i RobotStudio og synkronisert til RAPID.

Egenskapene til tooldata blir beskrevet her [17].

3.4.5 Jointtarget

Jointtarget blir brukt til å definere posisjonen som roboten og de eksterne aksene beveger seg med. Hver individuelle akseposisjon blir definert for roboten og de eksterne aksene. [18]

Alle RAPID-programmer bør inneholde et jointtarget for å kunne nullstille robotens akser.

Egenskapene til jointtarget blir beskrevet her [18].

3.4.6 Speeddata

Speeddata definerer hastigheten i bevegelsen til roboten og de eksterne aksene. Når flere forskjellige typer bevegelser er sammensatt så vil som regel en av hastighetene begrense alle hastighetene. [19]

Speeddata sine egenskaper blir beskrevet her [19].

RAPID har standard hastigheter som allerede er definert. Det er derfor ikke nødvendig å spesifisere egenskapene, hvis ikke dette er ønskelig.

3.4.7 Zonedata

Zonedata blir brukt til å spesifisere hvordan en posisjon skal bli avsluttet. For eksempel så spesifiseres det hvor nærme aksen må være til den programmerte posisjonen, før roboten kan bevege seg til den neste posisjonen. Det finnes to måter å avslutte en posisjon. Den ene måten er et stopp punkt og den andre er et *fly-by* punkt. [20]

Stopp-punkt vil si at roboten og de eksterne aksene må komme til punktet og stå stille før programmet går videre til neste instruksjon. [20]

Et *fly-by* punkt vil si at den programmerte posisjonen aldri blir oppnådd fordi bevegelsen endrer seg før punktet er nådd. [20]

Egenskapene til zonedata blir beskrevet her [20].

RAPID har soner som allerede er definert. Det er derfor ikke nødvendig å spesifisere alle egenskapene, hvis dette ikke er ønskelig.

3.4.8 Num

Num er en numerisk verdi i RAPID. Det kan være heltall eller desimaltall. I tillegg kan det skrives eksponentielt. [21]

3.4.9 Bool

Bool er boolsk som vil si logiske verdier i RAPID. [22]

3.5 Kommunikasjon med robot gjennom PC SDK

ABB PC SDK er et grensesnitt som gjør det mulig å lage PC programmer som kan kommunisere med en eller flere robot-kontrollere (IRC5) over nettverk. Nettverkskommunikasjonen utføres med TCP/IP [23]. Det er mulig å kommunisere med både reelle og virtuelle kontrollere ved hjelp av grensesnittet. Rammeverket som brukes i PC SDK er .NET. [24] [25]

3.6 Aksessere .NET i Python

IronPython er en åpen implementasjon av programmeringsspråket Python. Det som kjennetegner denne implementasjonen er at den er bygget på .NET plattformen. Som en følge av dette er det mulig å bruke Python-biblioteker og .NET-rammeverket sammen. [26]

3.7 Kommunikasjon med robot gjennom Robot Web Services

Robot Web Services åpner for kommunikasjon med robot-kontroller (IRC5) over nettverket. Nettverkskommunikasjonen utføres med HTTP. HTTP er brukt som applikasjonsprotokoll i Robot Web Services. Det er to byggesteiner i HTTP; nettadresse (*URL*) og verb. En nettadresse identifiserer noe og et verb sier noe om hva som skal utføres. [27]

REST (delkapittel 3.7.1) er arkitekturstilen som Robot Web Services er designet på. En nettadresse i REST identifiserer en ressurs, og et verb sier noe om hva som skal utføres på den spesifiserte ressursen. Informasjonen som kontrolleren sender tilbake kan være XML eller JSON (delkapittel 3.7.2), alt etter hva som er spesifisert i nettadressen. [27]

De viktigste verbene er: GET, PUT, POST, DELETE. [27]

Robot Web Services er nytt og ble sluppet i RobotWare versjon 6.0.

3.7.1 REST

REST er en forkortelse for *Representational State Transfer* og er en arkitekturstil som ofte er brukt i utvikling av internett-tjenester. Som nevnt ovenfor så brukes en nettadresse til å spesifisere hvilken ressurs som skal behandles, og et verb bestemmer hva som skal gjøres med ressursen. REST-arkitektur innebærer å lese en spesifisert nettside og kjører vanligvis over HTTP. [28]

3.7.2 JSON

JSON er en forkortelse for *JavaScript Object Notation,* og er en måte å strukturere data på. Data blir strukturert på en minimalistisk og leservennlig måte. Det primære bruksområdet til JSON er når data skal sendes mellom en server og internettapplikasjon. JSON brukes som et alternativ til XML. [29, 30]

3.7.3 Form data

Form data er kodet data som en klient legger til en nettadresse når en ressurs skal oppdateres på server. Ressursen oppdateres med informasjonen i den kodede data. [27]

3.7.4 Bonjour

Robot Web Services bruker Bonjour. Bonjour er en nettverksprotokoll som ikke trenger konfigurering. For å finne Robot Web Services på et nettverk må Bonjour klientprogramvare brukes. [27]

4. Konstruksjon

Oppgavebeskrivelsen sier at det skal lages et API som bruker to grensesnitt til å kommunisere med robot. Dette ble endret på grunn av at det er mer fornuftig å ha to API-er der det ene bruker PC SDK, og det andre bruker Robot Web Services i kommunikasjon med robot. Med dette vil det bli mer ryddig og oversiktlig. Om det ikke er ønskelig å bruke et av API-ene så kan det fjernes eller ikke bli tatt i betraktning. Det er enklere å videreutvikle på grunn av at ikke alt er samlet under ett API.

Et av kriteriene for hvilket programmeringsspråk som skulle brukes var at det skal være lett for brukerne å lære seg. Python tilfredsstilte dette kriteriet ettersom det ble designet for undervisning. Det er et språk som kan brukes til mange formål og er godt støttet. Det er vist at studenter hvor de fleste var nybegynnere eller hadde lite kunnskap om programmering, vil anbefale Python til andre etter å ha brukt det selv. [31]

For å kunne implementere API-ene så måtte også andre kriterier tilfredsstilles. Støtte for .NET er nødvendig for å kunne bruke PC SDK. Denne støtten gir IronPython som åpner for å kunne bruke .NET i Python. Dermed er det mulig å bruke Python til å kommunisere med robot gjennom PC SDK. Python har støtte for JSON og REST med et HTTP-bibliotek kalt *Requests*. Dette åpner for muligheten til å kommunisere med robot gjennom Robot Web Services. Det er ønskelig å bruke samme programmeringsspråk til begge API-ene fordi de skal testes mot hverandre. Da blir det ikke en test der to forskjellige programmeringsspråk er den avgjørende faktoren.

API-ene implementeres på en slik måte at brukeren ikke trenger å kunne PC SDK eller Robot Web Services. Brukeren utfører det som er ønskelig gjennom funksjonskall i API-ene. Dette er med på å forenkle kommunikasjonen med robot for brukeren. Det forutsettes at brukeren kan grunnleggende Python programmering, og hvordan grunnleggende kommunikasjon med robot foregår. API-ene skal medføre at brukeren sparer tid og kan lære deler av grensesnittene raskere ved at det er mulig å gå inn i implementasjonen av API-ene.

Videre i dette kapittelet blir det forklart hvordan virtuell robot er satt opp i RobotStudio. Det blir også gjennomgått hvordan API-ene er implementert.

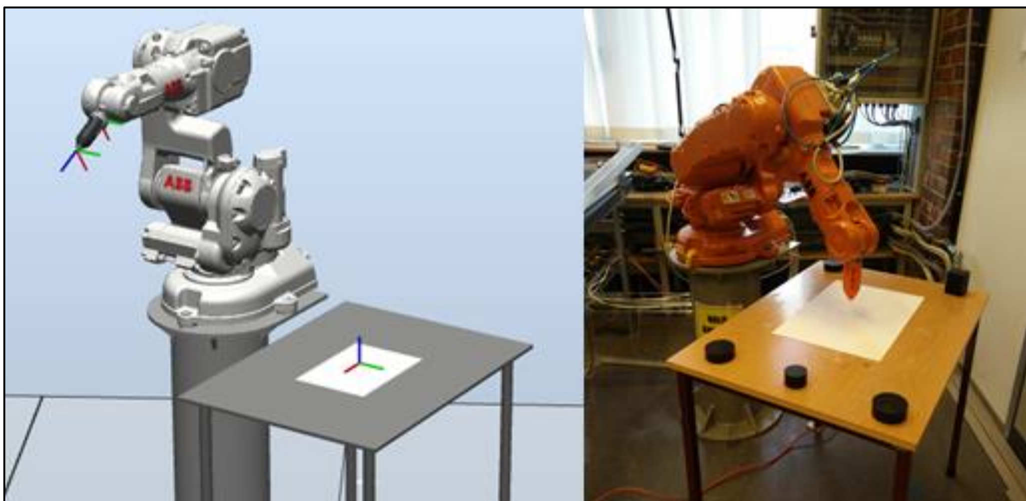
4.1 Oppsett av virtuell robot i RobotStudio

En virtuell robot er satt opp for å åpne muligheten til å simulere og teste lokalt, uten å være avhengig av at det er reelle roboter tilgjengelig eller i nærheten. Programmeringskonseptet som er brukt er sann frakoblet programmering. Simuleringer på virtuell robot skal gi samme resultater som på en reell robot.

Først ble en ny stasjon opprettet i RobotStudio og et robotsystem ble lagt til. Robotsystemet som ble lagt til stasjonen er av robottype *IRB140 6kg 0.81m* med RobotWare 6.02.01 kjørende på kontrolleren. Det var den nyeste versjonen av RobotWare da dette ble satt opp. Grunnen til at denne robottypen er brukt er fordi testingen utføres på en reell robot av denne robottypen.

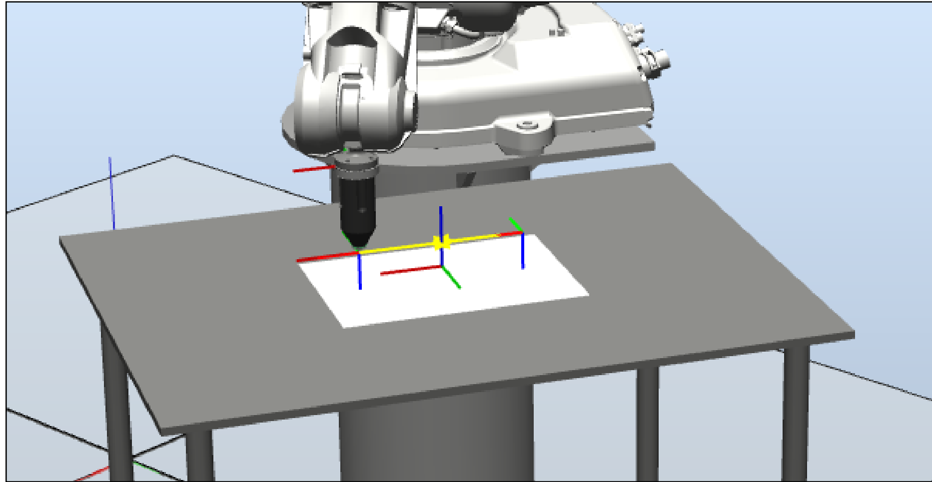
For å gjøre den virtuelle roboten lik den reelle, så ble komponenter lagt til. En sokkel ble importert for å få riktig høyde på robot. Et verktøy ble lagt til for å kunne utføre arbeid på et arbeidsstykke. Verktøyet som ble importert er en pennholder. Dette verktøyet blir brukt til å tegne med roboten. Arbeidsstykke er dermed et ark og arbeidsobjektet er bordet som arket ligger på.

Bordet blir ikke automatisk et arbeidsobjekt. Arbeidsobjektet blir spesifisert ved hjelp av tre punkter som definerer en brukerramme. Origo i koordinatsystemet ble opprettet i det ene hjørnet av bordet, men flyttet da det er mer hensiktsmessig at det er i senter av bordet.



Figur 4: Bilde av reell og virtuell robot.

Virtuell robot er klar til å programmeres og kan brukes til simulering. Det er nå mulig å definere blant annet mål og stier som kan brukes i programmering av roboten.



Figur 5: Bilde av virtuell robot som beveger seg mellom to mål.

Roboten ble testet for å se om alt fungerte som forventet ved å opprette to mål på arket og definere en sti mellom dem. Det viste seg at det ikke var mulig å nå målene fordi koordinatsystemet til målene var feil i forhold til verktøyets koordinatsystem. Koordinatsystemene til målene ble rotert slik de ble like verktøyets. Da kunne roboten nå målene. Roboten beveget seg som ønsket langs stien da koordinatsystemene samsvarte. Et problem som også oppstod var at base koordinatsystemet ikke var lokalisert i basen til roboten. For at alt skulle fungere som forventet måtte koordinatsystemet flyttes tilbake til basen av roboten. Stasjonen fungerte til slutt og var dermed klar for videre arbeid.

4.2 API som bruker PC SDK

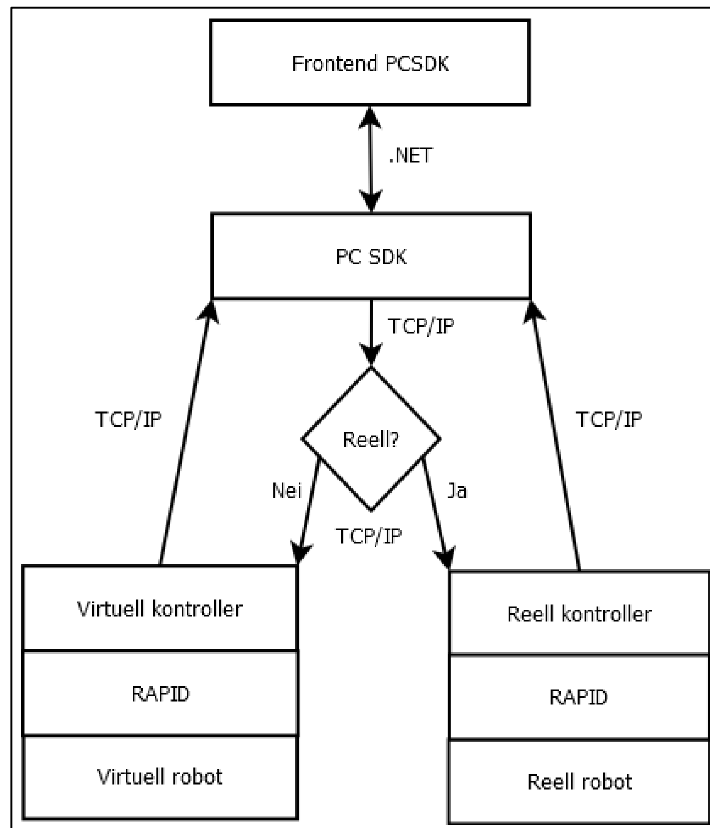
Dette delkapittelet vil gjennomgå oppbygningen og implementasjonen av API-et som implementeres i IronPython, og kommuniserer med robot gjennom PC SDK. Her vil det bli forklart hvordan koden er strukturert, vise flyten i et program og API-et, i tillegg til å gå gjennom implementeringen av utvalgte moduler.

API-et som kommuniserer gjennom PC SDK blir videre referert til som frontend PCSDK.

4.2.1 Kommunikasjonsflyt

I dette delkapittelet skal det gjennomgås hvordan den overordnede flyten er i frontend PCSDK, i tillegg til flyten i et program som bygger på dette API-et.

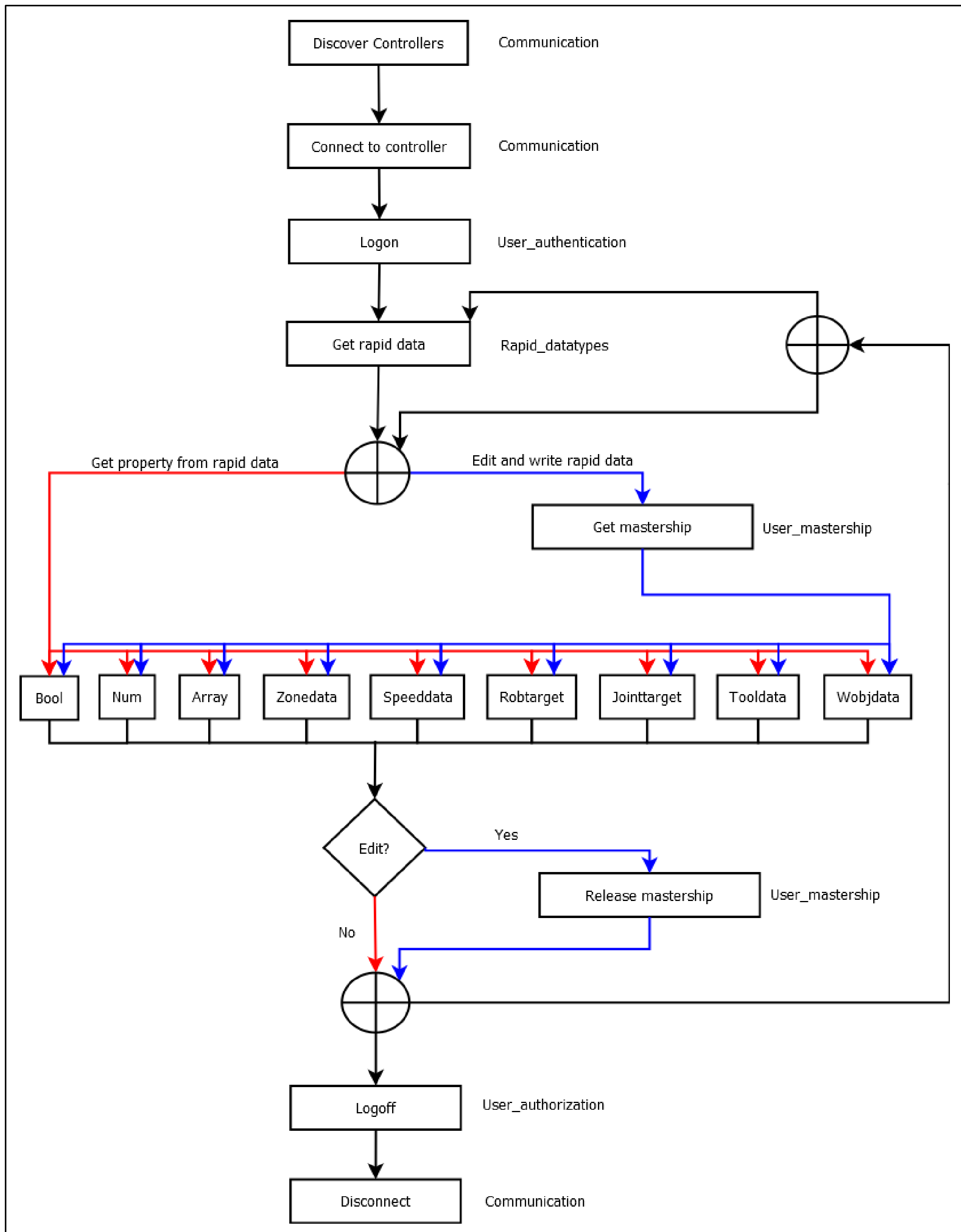
Det er laget et flytskjema for å vise hvordan frontend PCSDK kommuniserer med robot.



Figur 6: Flyten fra frontend PCSDK til en virtuell eller reell robot.

Øverst er frontend PCSDK som er implementert i IronPython. Programmene som brukeren lager aksesserer PC SDK gjennom frontend PCSDK. Aksesseringen er utført med .NET. PC SDK kommuniserer deretter med en reell eller virtuell kontroller, alt etter hva som er ønskelig. På kontrolleren brukes programmeringsspråket RAPID. Roboten utfører det som er spesifisert i RAPID programmet.

Under er et flytskjema som presenterer hvordan modulene i frontend PCSDK brukes i implementasjonen av et program.



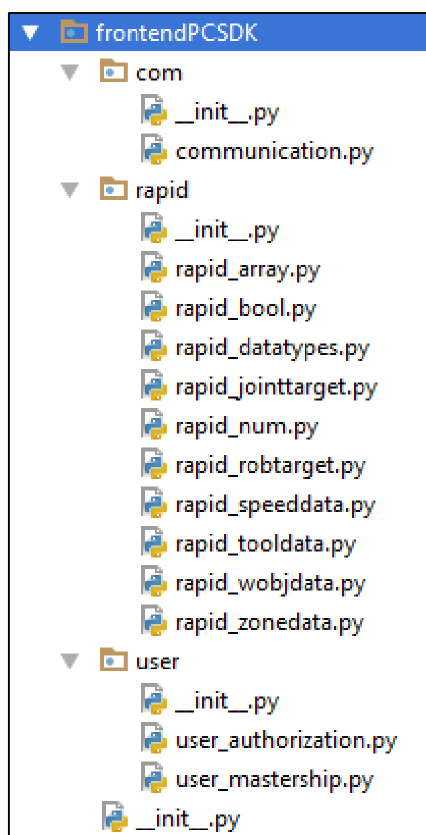
Figur 7: Flytskjema av modulenes bruk ved implementasjon av et program i frontend PCSDK.

Det første som må gjøres for å kommunisere med robot er å finne alle robot-kontrollerne på nettverket. Når alle kontrollerne på nettverket er funnet så kobles det til den ønskede

kontrolleren, virtuell eller reell. Neste steg er å logge på en bruker. Det finnes en forhåndsdefinert (*default*) bruker på kontrollerne. Det er mulig å logge på med den forhåndsdefinerte brukeren eller å spesifisere en egendefinert bruker. Deretter hentes RAPID-variablene som er ønskelige å bruke i programmet. Det er mulig å lese ulike egenskaper fra variablene som er hentet. I flytskjemaet over representeres det ved den røde linjen. Om det er ønskelig å endre en variabel i RAPID så er det også mulig. I flytskjemaet representeres det med den blå linjen. De ulike RAPID-datatypene som støttes kan sees i flytskjema over. Hvis en variabel skal endres så er det viktig at *mastership* til kontrolleren blir sikret rett før variabelen endres på kontrolleren. Å holde *mastership* gir skrivetilgang til robot-kontroller. Når variabelen er oppdatert så er det viktig at *mastership* til kontrolleren slippes. Om det ikke utføres riktig så kan kontrolleren gå i vranglås. Et program kan hente RAPID-variabler fra kontrolleren flere ganger om ønskelig. Det er også mulig å hente egenskaper fra variabler, eller endre variabler flere ganger i løpet av et programs livssyklus. Hoveddelen av programmene som skrives vil i stor grad være det å hente og endre variabler. Når et program skal avsluttes er det viktig at brukeren logges av og at kontrolleren blir frakoblet. Hvis brukeren ikke logges av kan det oppstå feil som at for mange er tilkoblet kontrolleren. Det kan også oppstå minnelekkasje hvis kontrolleren ikke blir frakoblet da den ikke vil bli slettet fra minne.

4.2.2 Prosjektstruktur

Prosjektet er strukturert med tre mapper. Mappene er kalt *com*, *rapid* og *user*. De opprettede filene er plassert i en av disse mappene basert på funksjonaliteten til filen.



Figur 8: Prosjektstruktur til frontend PCSDK bestående av mappene *com*, *rapid* og *user*.

Mappen med navnet *com* inneholder kommunikasjonsmodulen. Denne modulen inneholder funksjoner for å finne kontrollere, koble til kontroller, koble fra kontroller osv. Det er denne modulen som for eksempel blir brukt i starten og i slutten av et program. Mappen med navnet *user* inneholder to moduler. Som sett i figuren over er navnene til filene *user_authorization* og *user_mastership*. Modulen *user_authorization* blir brukt til å logge på en kontroller med den forhåndsdefinerte brukeren, eller med en egendefinert bruker. Når det er ønskelig å endre RAPID-data på kontrolleren så må *mastership* skaffes. Modulen *user_mastership* har funksjonalitet for å få tak i *mastership,* og slippe *mastership* etter at variablene er endret i RAPID på kontroller. Den siste mappen med navnet *rapid* inneholder modulene som henter og behandler RAPID-datatyper. Navnene til filene tilsier hvilken RAPID-datatype filen behandler. For mer informasjon om detaljer rundt RAPID-modulene se delkapittel 4.2.4.

4.2.3 Kodestruktur

Kildekoden er strukturert slik at det skal være så lett som mulig for en bruker å forstå egenskapen til modulen. Dette er opprettholdt ved å ha en liten kommentar som beskriver

modulen og en utfyllende kommentar til hver funksjon. Funksjonene fikk beskrivende navn slik at det er mulig å forstå hensikten med funksjonen ut i fra navnet.

```
"""
Module for handling RAPID data type robtarget. This module makes it possible to
edit and write the rapid data type robtarget, as well as displaying the
different properties of the robtarget.
"""
```

Figur 9: Eksempel på modulkommentar i frontend PCSDK.

Figuren ovenfor er et eksempel på hvordan en modulkommentar ser ut. Denne kommentaren er tatt fra *rapid_robtarget*. Den beskriver hva modulen kan utføre av operasjoner på RAPID-datatypen robtarget.

```
"""
Edits the specified robtarget property and writes it to the controller.
Remember to get mastership before calling this function, and release mastership
right after.

Input:
    ABB.Robotics.Controllers.RapidDomain.RapidData: rapid_data
    String: property (accepted types: trans, rot, robconf, extax)
    String: new_value
Output:
    String: result message or error
Examples:
    message = edit_and_write_rapid_data_property(rapid_data,'trans',
                                                 '[100,100,0]')
    message = edit_and_write_rapid_data_property(rapid_data,'rot',
                                                 '[1,0,0,0]')
    message = edit_and_write_rapid_data_property(rapid_data,'robconf',
                                                 '[1,0,1,0]')
    message = edit_and_write_rapid_data_property(rapid_data,'extax',
                                                 '[9E9,9E9,9E9,
                                                   9E9,9E9,9E9]')
"""
```

Figur 10: Eksempel på funksjonskommentar i frontend PCSDK.

Kommentaren ovenfor er også tatt fra *rapid_robtarget*. Denne kommentaren hører til funksjonen *edit_and_write_rapid_data_property()*. Her beskrives det øverst hva funksjonen gjør; endrer robtarget på den tilkoblede kontrolleren. Så gir kommentaren et hint til brukeren om at *mastership* er nødvendig før funksjonen kjører, i tillegg til å slippe *mastership* etter at denne funksjonen er fullført. Deretter står det spesifisert hvilke argumenter (*Input*) funksjonen

trenger. Her nevnes det hvilken datatype argumentene skal være, og noen ganger følger et eksempel på hva som kan settes inn. Det som returneres fra funksjonen står under *Output*. I dette tilfellet returneres en streng med informasjon om hva som ble resultatet. Det er laget en seksjon i kommentaren som heter *Examples*. Denne seksjonen gir brukeren hjelp til hvordan funksjonen kan kalles, og informasjon om hvordan argumentene skal se ut. Noen funksjoner har flere eksempler, mens enklere funksjoner har ingen. Denne funksjonen har flere eksempler. Grunnen til dette er fordi det kan velges hvilken egenskap som skal endres og hvilken verdi den skal få.

```
def edit_and_write_rapid_data_property(rapid_data, property, new_value):
```

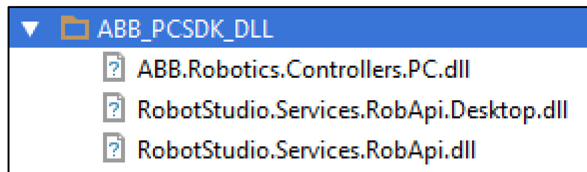
Kildekode 1: Eksempel på funksjonsnavn i frontend PCSDK.

Hver modul er bygget opp av en rekke funksjoner. Ovenfor er et eksempel på navnet til en funksjon og hvordan den er definert. Et av kriteriene i implementasjonen var at det skulle være lett å forstå og lett å utvide funksjonalitet. Navnet til funksjonen skal være selvforklarende for en ny bruker. På grunn av at funksjonen er i *rapid_robtarget* så hører den til datatypen robtarget, og navnet tilsier at funksjonen kan endre en av egenskapene til robtarget. For å gjøre det lett å utvide så er alle funksjoner uavhengig av hverandre. Dette fører til at det er lett å forandre på frontend PCSDK. Om det endres i en funksjon i kildekoden så vil ikke det ha en negativ effekt på resten av kildekoden. Frontend PCSDK er tilstandsløs.

4.2.4 Implementasjon

I dette delkapittelet er et utvalg av moduler og funksjoner dekket. De som utelates bygger på de samme prinsippene som de som er forklart. Funksjoner, typer og klasser fra PC SDK er framhevet med blå skrift videre i dette delkapittelet.

For å kunne bruke PC SDK i IronPython så måtte det opprettes referanse til assembly-filene. Assembly-filene er i dette tilfellet DLL-filer som er opprettet under installasjonen av PC SDK. Filene er kopiert til en egen mappe i prosjektet, og referert til dynamisk slik at brukeren kun trenger å installere PC SDK 6.02 på datamaskinen, og API-et vil fungere.



Figur 11: Assembly-filene som gjør det mulig å bruke PC SDK i IronPython.

Overskriftene under refererer til modulnavnene i frontend PCSDK.

Communication

Modulen *communication* inneholder viktige funksjoner som brukes i blant annet oppkoblingen til robot-kontroller. Først er det nødvendig å finne kontrollerne på nettverket. Funksjonen *discover_controllers_on_network()* finner alle kontrollerne som er tilgjengelige. Dette gjøres ved å opprette en nettverkssøker (*NetworkScanner*), som er i PC SDK, og kalle søk på dette objektet. Kontrollerne som blir funnet er av typen *ControllerInfoCollection*, og blir returnert av funksjonen.

Det er mulig å koble til robot-kontroller med to forskjellige funksjoner. Den ene funksjonen, *connect_robot_with_name()*, gjør det mulig å koble til en robot-kontroller ved å spesifisere navnet til kontrolleren. Den andre, *connect_robot_with_ipaddr()*, gjør det mulig å koble til robot ved å spesifisere IP-adressen. Begge funksjonene tar inn to argumenter. Likt for begge er at de tar inn objektet *ControllerInfoCollection*, som blir returnert fra funksjonen som er beskrevet i forrige avsnitt, i tillegg til en streng med navn eller IP-adressen basert på hvilken funksjon som er brukt. I *connect_robot_with_name()* blir *ControllerInfoCollection* sjekket for å finne informasjonen til kontrolleren (*ControllerInfo*) som er spesifisert i argumentet. Hvis kontrolleren blir funnet så opprettes et kontroller-objekt av typen *ABB.Robotics.Controllers.Controller,* ved å kalle funksjonen *CreateFrom(ControllerInfo)* i klassen *ControllerFactory*. *Connect_robot_with_name()* returnerer kontroller-objektet sammen med en melding med resultatet og en logisk verdi som indikerer om koblingen er opprettet. For *connect_robot_with_ipaddr()* gjelder det samme som beskrevet ovenfor, bare at i stedet for å finne informasjonen til kontrolleren basert på navn, så finnes den basert på IP-adressen.

```
controller, res, connected = connect_robot_with_name(controllers, 'IRB_140')
```

Kildekode 2: Funksjon som kobler til robot-kontroller med et gitt navn.

Når kommunikasjon med robot-kontroller er fullført så kalles funksjonen *disconnect_robot_controller()*. Denne funksjonen tar inn kontroller-objektet, *ABB.Robotics.Controllers.Controller*, som argument. Metoden *dispose()* blir kalt på kontrolleren slik at den blir fjernet fra minne. Ved å gjøre dette så unngås minnelekkasje.

User_authorization

Modulen *user_authorization* brukes til autorisering på robot-kontroller. Den har to funksjoner som gjør det mulig å logge på robot-kontroller. Den ene av disse funksjonene har navnet *logon_robot_controller_default(),* og tar inn kontroller-objektet, som beskrevet tidligere, som argument. Denne funksjonen logger på robot-kontrolleren med den forhåndsdefinerte brukeren. Funksjonen *Logon()* blir kalt på kontroller-objektet, og brukeren blir lagt inn som argument i funksjonen ved å spesifisere *ABB.Robotics.Controllers.UserInfo.DefaultUser*. *Logon_robot_controller_default()* returnerer en logisk verdi som indikerer om operasjonen er vellykket, og en streng som beskriver resultatet. Om det er ønskelig å logge på med en egendefinert bruker så kalles funksjonen *logon_robot_controller_with_username()*. Denne funksjonen tar inn kontrolleren som argument og to strenger, i form av brukernavn og passord. På kontrolleren blir *Logon()* kalt også her. Forskjellen er at argumentet denne gangen er *ABB.Robotics.Controllers.UserInfo(username, password)*. Det samme returneres fra denne funksjonen.

```
is_logged_in, res = logon_robot_controller_default(controller)
```

Kildekode 3: Funksjon brukt til å logge på robot-kontroller med forhåndsdefinert bruker.

Om det er ønskelig å bytte bruker eller om programmet skal avsluttes så kalles *logoff_robot_controller()*. Denne funksjonen tar også inn kontroller-objektet som argument. På dette objektet kalles funksjonen *Logoff()* som logger av brukeren. Det som returneres fra denne funksjonen er en logisk verdi som indikerer om det var vellykket og en streng med resultatet.

User_mastership

For å kunne endre variabler så må *mastership* sikres. Denne modulen har en funksjon som gjør dette. Funksjonen *get_master_access_to_controller_rapid()* prøver å sikre *mastership* på kontrolleren. Funksjonen tar inn kontroller-objektet som argument. For å få tak i *mastership* kalles *ABB.Robotics.Controllers.Mastership.Request()* med ett argument. Argumentet er kontroller-objektets RAPID-domene. Dette hentes fra kontroller-objektet ved å kalle *kontroller_objektet.Rapid*. RAPID-domene brukes her på grunn av at det er RAPID-variabler som skal behandles. *Get_master_access_to_controller_rapid()* returnerer en logisk verdi som indikerer om det var vellykket, en melding av typen streng, og et objekt av typen *mastership*. Det er veldig viktig å få tak i *mastership* rett før en RAPID-variabel skal endres på kontrolleren, da det ikke er mulig å endre uten.

```
is_master, res, mastership = get_master_access_to_controller_rapid(controller)
```

Kildekode 4: Funksjon som sikrer *mastership* på robot-kontroller.

Etter at en RAPID-variabel er endret så er det veldig viktig å slippe *mastership* på kontrolleren. Grunnen til dette er for å ikke låse kontrolleren. For å slippe *mastership* kalles *release_and_dispose_master_access()*. Denne funksjonen tar inn *mastership*-objektet som argument. *Mastership*-objektet er det som ble returnert fra funksjonen som fikk tak i *mastership*. I funksjonen *release_and_dispose_master_access()* blir funksjonen *Release()* kalt på *mastership*-objektet slik at *mastership* slippes. I tillegg blir funksjonen *Dispose()* kalt slik at minne blir frigjort for å unngå minnelekkasje. Funksjonen returnerer en logisk verdi som indikerer om det var vellykket, og en streng med resultatet.

For å sjekke om kontroller-objektet har *mastership* på kontrolleren så brukes funksjonen *is_controller_master()*. Denne funksjonen tar inn kontroller-objektet som argument og sjekker om det holdes *mastership* på kontrolleren. Dette gjøres ved å sjekke om attributtet *IsMaster* på kontroller-objektet er sann eller usann. Funksjonen returnerer den logiske verdien for om det er sant eller usant.

Rapid_datatypes

Modulen *rapid_datatypes* har en funksjon med navnet *get_rapid_data()*. Denne funksjonen er brukt til å hente RAPID-variabler fra robot-kontroller. Argumentene til funksjonen er kontroller-objektet, navnet til programmet som er på kontrolleren, navnet til modulen som variabelen er definert i, og navnet til variabelen. Funksjonen kan hente ut alle variabler definert i RAPID, men frontend PCSDK kan kun behandle RAPID-datatyper som er i *rapid*-mappen i figur 8. For å få tak i en RAPID-variabel så kalles *GetRapidData()* på kontroller-objektet som er gitt inn som argument. Kallet ser slik ut; *kontroller_objektet.Rapid.GetRapidData(prog_name, module_name, var_name)*. Dette kallet returnerer et *RapidData*-objekt som refererer til variabelen på robot-kontrolleren. Funksjonen returnerer en logisk verdi som indikerer om det var vellykket, i tillegg til *RapidData*-objektet eller en eventuell feilmelding hvis det oppstod en feil.

```
got_var, rapid_data = get_rapid_data(controller, 'T_ROB1', 'MainModule', 'x')
```

Kildekode 5: Funksjon brukt til å hente RAPID-variabel fra robot-kontroller.

Rapid_robtarget

Denne modulen behandler RAPID-datatypen robtarget. Modulen har funksjoner for å hente ut informasjon fra et *RapidData*-objekt av typen robtarget. Det er mulig å hente ut *trans*, *rot*, *robconf*, *extax*, eller hele robtarget. Felles for funksjonene er at de tar inn et *RapidData*-objekt som argument. Dette objektet aksesseres for å hente ut den ønskede informasjonen til robtarget. For eksempel i funksjonen *get_trans_tostring()* så hentes *trans* fra *RapidData*-objektet ved å kalle *RapidData_objektet.Value.Trans*. Informasjonen blir formatert og returnert.

Det er to metoder for å endre en variabel av typen robtarget i RAPID på kontrolleren. Den ene har navnet *edit_and_write_rapid_data_property()*. Denne brukes til å endre en spesifisert egenskap. Funksjonen tar inn argumentene; *RapidData*-objekt av typen robtarget, egenskapen som skal endres, og den nye verdien egenskapen skal få. Egenskapene som kan spesifiseres er *trans*, *rot*, *robconf* og *extax*. Den nye verdien til for eksempel egenskapen *trans* kan se slik ut [100,0,0], hvor tallene representerer x, y, og z koordinater. Når en egenskap skal endres i

RAPID så formateres en streng slik at den ser lik ut som et robtarget i RAPID. For eksempel i RAPID så ser et robtarget slik ut:

```
[[0,0,30],[0,1,0,0],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]]
```

Kildekode 6: Eksempel på hvordan et robtarget ser ut i RAPID.

Betydningen til klammene er dette; [[trans],[rot],[robconf],[extax]]. De egenskapene som ikke skal endres hentes fra *RapidData*-objektet som strenger. En ny streng blir laget med de nye verdiene til for eksempel *trans*, og de andre egenskapene som er hentet blir satt inn i strengen uendret. Dette gir en ny streng som vil se slik ut:

```
[[100,0,0],[0,1,0,0],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]]
```

Kildekode 7: Eksempel på den oppdaterte strengen som representerer et robtarget i RAPID.

For å kunne endre denne strengen i RAPID på kontroller så må et objekt av datatypen robtarget være opprettet. Det gjøres ved å for eksempel opprette en variabel med navnet *new_robtarget* som settes lik *RapidData*-objektets verdi. Deretter kalles funksjonen *FillFromString2()* på variabelen. Argumentet til denne funksjonen er den nye strengen som representerer det nye robtarget. Dette fører til at *new_robtarget* får de oppdaterte verdiene. *New_robtarget* endres i RAPID på kontrolleren ved å sette *RapidData*-objektets verdi lik *new_robtarget*. *Edit_and_write_rapid_data_property()* returnerer en streng med resultatet.

```
res = edit_and_write_rapid_data_property(var_rtarget, 'trans', '[100,0,0]')
```

Kildekode 8: Funksjon som endrer en spesifisert egenskap til et robtarget.

Den andre funksjonen heter *edit_and_write_rapid_data()* og brukes til å endre hele robtarget i RAPID. Argumentene til denne funksjonen er *RapidData*-objektet og de nye verdien til *trans*, *rot*, *robconf* og *extax*. Denne funksjonen bruker de samme teknikkene som den forrige. Den eneste forskjellen er at ingen egenskaper blir hentet fra *RapidData*-objektet på grunn av at alle blir definert. Funksjonen returnerer det samme som overnevnte.

Rapid_speeddata

Speeddata er ikke en type i *RapidDomain* i PC SDK. Det vil si at speeddata ikke har egne funksjoner definert til seg, noe som ville gjort det enklere å endre datatypen speeddata på kontroller.

I RAPID er det predefinerte hastigheter som kan brukes. Disse brukes ved å opprette en speeddata-variabel og sette den lik for eksempel *v10*. Her er *v10* en referanse til en predefinert liste av verdier som utgjør en hastighet. Dette forenkler opprettelse av speeddata ved at *v10* skrives i stedet for [10,500,5000,1000]. I modulen *rapid_speeddata* er en liste med predefinerte hastigheter opprettet. Listen brukes når speeddata skal endres på kontroller.

Modulen *rapid_speeddata* har to funksjoner for å endre hastighet. Den ene endrer med predefinerte hastigheter, og den andre endrer med egendefinerte hastigheter på kontrolleren. Funksjonen som endrer med predefinerte hastigheter heter *edit_and_write_rapid_data_base()* og tar inn to argumenter. Det første er *RapidData*-objekt og det andre er den nye hastigheten. I funksjonen sjekkes det om hastigheten er en gyldig hastighet ut ifra listen over predefinerte hastigheter. Om *v10* er spesifisert så opprettes en streng som ser slik ut; [10,500,5000,1000]. For å endre hastighet må et speeddata objekt lages ved å hente verdien til *RapidData*-objektet. Funksjonen *FillFromString2()* blir kalt på det nyopprettede speeddata objektet, med den nye strengen som argument. Deretter settes *RapidData*-objektets verdi lik det nyopprettede speeddata-objektet. Da endres speeddata i RAPID på kontrolleren.

```
res = edit_and_write_rapid_data_base(var_bspeed, 'v100')
```

Kildekode 9: Funksjon som endrer predefinert hastighet til speeddata.

Den andre funksjonen som skriver speeddata heter *edit_and_write_rapid_data()* og tar inn fem argumenter. Det første argumentet er *RapidData*-objektet, og de andre er verdier til egenskapene i speeddata. Oppbygningen av funksjonen er lik som den andre bortsett fra at det spesifiseres verdier til alle egenskapene. Dette kan være ønskelig om de predefinerte hastighetene ikke oppfyller kravene til oppgaven som skal utføres.

Rapid_array

Modulen *rapid_array* gjør det mulig å endre endimensjonale lister av typen num. Det er definert funksjoner for å hente ut lengden til en liste ved å kalle *get_length()*, og dimensjonen ved å kalle *get_dimensions()*. Begge funksjonene tar inn *RapidData*-objekt som argument og henter ut informasjonen fra dette objektet. For å hente ut lengden så aksesseres *RapidData* på denne måten *RapidData_objektet.Value.Length*, og dimensjon hentes ut slik *RapidData_objektet.Value.Rank*.

Det er to funksjoner for å endre en liste av num på kontroller. Den ene funksjonen heter *edit_and_write_rapid_data_num_index()* og tar inn tre argumenter. Det første argumentet er *RapidData*-objekt, det neste er indeksen i listen som skal oppdateres, og det siste er verdien indeksen skal endres til. Funksjonen sjekker blant annet at det er en liste av typen num, i tillegg til om indeksen i argumentet er en gyldig indeks i listen. Hvis sjekkene oppfylles så brukes funksjonen *RapidData_objektet.WriteItem()* til å endre variabelen i indeksen på kontroller. Argumentene til denne metoden er verdien som det skal endres til og indeksen verdien skal endres i. Verdien som skal settes inn i listen må gjøres om fra Python *int* eller *float* til RAPID-datatypen num. Dette gjøres ved å kalle *ABB.Robotics.Controllers.RapidDomain.Num()* med Python verdien som argument.

```
res = edit_and_write_rapid_data_num(var_array, [1, 1, 1])
```

Kildekode 10: Funksjon brukt til å endre en liste av num på kontroller.

Den andre funksjonen som endrer en liste av num på kontroller heter *edit_and_write_rapid_data_num()*. Denne funksjonen tar inn to argumenter. Første argument er *RapidData*-objekt, og det neste er en liste av verdiene som det skal endres til på kontroller. Listen som spesifiseres må være mindre eller like stor som listen i RAPID på kontrolleren. Hvis listen i argumentet er tom, så vil listen på kontroller bli nullstilt og alle verdiene vil bli fjernet. Hvis listen er like stor som den definerte listen i RAPID, så vil den byttes ut med den nye. Hvis størrelsen på listen i argumentet er mindre så vil verdiene settes inn fra starten i listen i RAPID, og de resterende verdiene som ikke er spesifisert i listen vil bli satt til null. Listen oppdateres på kontroller med de samme teknikkene som beskrevet for blant annet speeddata. Funksjonen *FillFromString2()* kalles på listen som hentes fra *RapidData*-objektet

med et argument. Argumentet er den nye listen konvertert til en streng. Deretter settes *RapidData*-objektets verdi lik det objektet som metoden *FillFromString2()* ble kalt på, og listen i RAPID vil bli endret.

Gjennom dette delkapittelet er et utvalg av moduler og funksjoner beskrevet. I dette delkapittelet er det vist at enkle funksjoner er laget der brukeren ikke trenger å ha kunnskap om PC SDK. Dette fører til tidsbesparelse på grunn av at brukeren ikke trenger å lære seg PC SDK. All logikk som gjør det mulig å kommunisere med robot gjennom dette grensesnittet ligger i funksjonene. Med dette er det også enkelt for en bruker å lære seg deler av PC SDK ved å studere kildekoden.

Mappe	*Modulnavn*	*Funksjoner*
Com	Communication	Discover_controllers_on_network, Connect_robot_with_name, Connect_robot_with_ipaddr, Disconnect_robot_controller, Is_connected_to_controller
User	User_authorization	Logon_robot_controller_default, Logon_robot_controller_with_username, Logoff_robot_controller
	User_mastership	Is_controller_master, Get_master_access_to_controller_rapid, Release_and_dispose_master_access
Rapid	Rapid_array	Get_length_array, Get_dimensions_array, Edit_and_write_rapid_data_num_index, Edit_and_write_rapid_data_num
	Rapid_bool	Get_state_tostring, Get_state, Edit_and_write_rapid_data
	Rapid_datatypes	Get_rapid_data
	Rapid_jointtarget	Get_robax_tostring, Get_extax_tostring, Get_jointtarget_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_num	Get_value_tostring, Get_value, Edit_and_write_rapid_data
	Rapid_robtarget	Get_trans_tostring, Get_rot_tostring, Get_robconf_tostring, Get_extax_tostring, Get_robtarget_tostring,

		Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_speeddata	Get_speeddata_tostring, Edit_and_write_rapid_data_base, Edit_and_write_rapid_data
	Rapid_tooldata	Get_robhold_tostring, Get_tframe_tostring, Get_tload_tostring, Get_tooldata_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_wobjdata	Get_robhold_tostring, Get_ufprog_tostring, Get_ufmec_tostring, Get_uframe_tostring, Get_oframe_tostring, Get_wobjdata_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	rapid_zonedata	Get_zonedata_tostring, Edit_and_write_rapid_data_base, Edit_and_write_rapid_data

Tabell 1: Liste av modulene i frontend PCSDK og de tilhørende funksjonene.

4.2.5 Verifisering av implementasjon

Koden som er skrevet skal være av høy kvalitet. For å oppnå dette er det brukt statisk kodeanalyse. Flake8 er verktøyet som er brukt, og gir blant annet tilbakemelding på om stilveiledningen, PEP8, er fulgt. Endringer i koden er utført basert på hva Flake8 gav av tilbakemeldinger. Tilbakemeldingene sier blant annet om det er ubrukte importeringer, ubrukte variabler og formateringsfeil i kode.

Det er kjørt McCabe kompleksitetstest på koden for å få en indikasjon på hvor kompleks koden er. Kompleksitetstesten har navnet syklomatisk kompleksitet og sjekker hvor mange lineært uavhengige stier det er i kildekoden. Det som fører til at koden får en høyere kompleksitet er mengden av veivalg i koden [32]. Testen ble satt til å rapportere kode der kompleksitet oversteg ti [32].

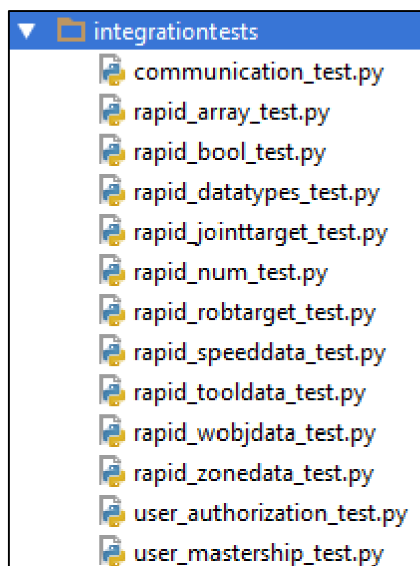
Resultatet indikerte at enkelte av funksjonene som endrer variabler i RAPID på kontroller har høy kompleksitet og oversteg ti. Grunnen til dette er fordi funksjonene inneholder en del

veivalg som sjekker om brukeren har gitt inn riktig data, og veivalg for hva brukeren ønsker å gjøre. For å redusere kompleksiteten kan det bli utført færre sjekker. Det er ikke ønskelig å gjøre færre sjekker som kan føre til dårligere tilbakemeldinger til brukeren. Kildekoden er gjennomgått for å se om det er mulighet til å forenkle i form av å redusere veivalg uten å la det føre til dårligere tilbakemeldinger. Det ble funnet veivalg i funksjoner som ved flytting kunne redusere antall veivalg, uten å føre til dårligere tilbakemeldinger. Endringer som dette ble utført.

Testing av implementasjonen er en viktig del av utviklingen. På grunn av at alt er avhengig av noe annet i kommunikasjon med robot så er testingen en form for integrasjonstesting. For eksempel så er det ikke mulig å endre en variabel på en kontroller uten å koble seg til og logge på. Det at mange moduler må være med for å teste en modul gjorde at valget falt på integrasjonstesting.

Testmetoden som er brukt er kalt hvit boks testing (*White Box Testing).* Hvit boks testing utføres ved å gi inn data og sjekke at systemet prosesserer data korrekt, og gir ut ønsket resultat. Dette tester at API-et oppfører seg som ønsket. Denne metoden kan brukes på blant annet integrasjonstestnivå og er dermed tatt i bruk her. [33]

Python-biblioteket *unittest* er brukt til å implementere integrasjonstestene. Integrasjonstestene skal teste at funksjonene i API-et gir ønsket resultat, basert på data som er gitt inn. Dette biblioteket åpner for muligheten til å implementere tester for hver funksjon, der hver test kan ha et oppsett før den kjører, og en opprydning etter at den er ferdig. Det kan spesifiseres hvilken test som skal kjøres hvis det kun er gjort endringer i en funksjon, eller så kan alle testene til en hel modulen kjøres. Dette er med på å skape fleksibilitet ved testing. Det er laget tester hvor alle argumentene er korrekte for å sjekke at alt fungerer som ønsket. I tillegg er det laget tester hvor argumentene ikke er korrekte, for å se at funksjonene gir ønsket resultat ved feil argumenter. Dette er ikke sett på som en *unit test* på grunn av at flere moduler må være med for å kunne teste en funksjon.



Figur 12: Mappe med integrasjonstester til frontend PCSDK.

I kildekoden nedenfor er testen til skrivefunksjonen i modulen *rapid_num*. Før en test kjøres så kalles en funksjon med navnet *SetUp*. Denne funksjonen utfører grunnleggende oppsett før hver test, samtidig som det er mulig å spesifisere ekstra oppsett for enkelte tester. Før hver test i *rapid_num_test* blir alle kontrollerne funnet på nettverket. Deretter kobles det til den virtuelle kontrolleren. Så logges det på kontrolleren med den forhåndsdefinerte brukeren. Spesifikt for testen nedenfor er at *mastership* må skaffes. Dette er ekstraoppsett. Testen blir kjørt når dette er utført.

Kildekoden nedenfor viser at en num variabel med navnet *var_number* hentes fra RAPID på kontrolleren. Deretter blir det forsøkt å endre variabelen til verdi ti. For at testen skal være vellykket så sjekkes det om variabelen på kontrolleren har fått verdi ti.

Når en test er ferdig så kalles en funksjon med navnet *tearDown*. Denne funksjonen rydder opp etter testen. Det blir sjekket hvilken test som er kjørt for å kunne rydde opp spesifikt for den enkelte testen. I dette tilfellet så endres variabelen på kontrolleren tilbake til det den var før den ble endret, slik at testen kan kjøres igjen uten å måtte manuelt endre variabelen tilbake. Deretter slippes *mastership* på kontrolleren. Etter at den spesifikke opprydningen er utført så kjøres den generelle opprydningen. Denne opprydningen utføres etter hver test, og innebærer å logge av og koble fra kontrolleren.

```
# Tests edit_and_write_rapid_data with correct input data
def test_edit_and_write_rapid_data_correct(self):
    """ Tests edit_and_write_rapid_data with correct input data """
    got_var, var_number = rapid_datatypes.get_rapid_data(self.controller,
                                                         'T_ROB1', 'MainModule',
                                                         'var_number')
    if not got_var:
        print 'Could not get variable from controller. Test will not be run.'
        sys.exit()
    _ = rapid_num.edit_and_write_rapid_data(var_number, 10)
    self.assertEqual(float(var_number.Value), 10)
```

Kildekode 11: Integrasjonstest til skrivefunksjonen i modulen rapid_num i frontend PCSDK.

Det er ønskelig å kjøre kodedekning (*code coverage*) for å sjekke hvor mye av koden i modulene som blir kjørt ved testing. Hvis store deler av koden blir kjørt og testet så er det mindre sannsynlig at feil vil oppstå. Problemet som oppstod var at det ikke ble funnet kodedekningsverktøy som kunne kjøres på IronPython. Det ble forsøkt å bruke *Coverage.py* som er et kodedekningsverktøy i Python. Når verktøyet kjøres fra *terminal* eller *cmd* så kalles standard Python tolker. Dette gir problemer da den ikke håndterer .NET. Det er også forsøkt å implementere *Coverage.py* i kildekoden ved å bruke grensesnittet som følger med, men det gav feilmeldinger ved kjøring. Se delkapittel 6.2 under diskusjon for mer om dette. Dermed er det kjørt integrasjonstester på alle funksjoner men uten å vite hvilken dekning integrasjonstestene gir.

Sammenlignet med API-et i neste delkapittel, så ville testene med stor sannsynlighet gitt et kodedekningsresultat i samme område. Grunnen er fordi testene tester de samme scenarier.

Alle integrasjonstester er kjørt på virtuell kontroller med RobotWare 6.01 og 6.02 for å se at det fungerte slik det skulle. Resultatet viste at alt fungerte. Noen vilkårlige tester ble også kjørt på reell kontroller som resulterte i det samme som på virtuell. Grunnen til at det ble kjørt på RobotWare 6.01 på virtuell kontroller er fordi den reelle kontrolleren som dette skal testes på også har RobotWare 6.01.

4.3 API som bruker Robot Web Services

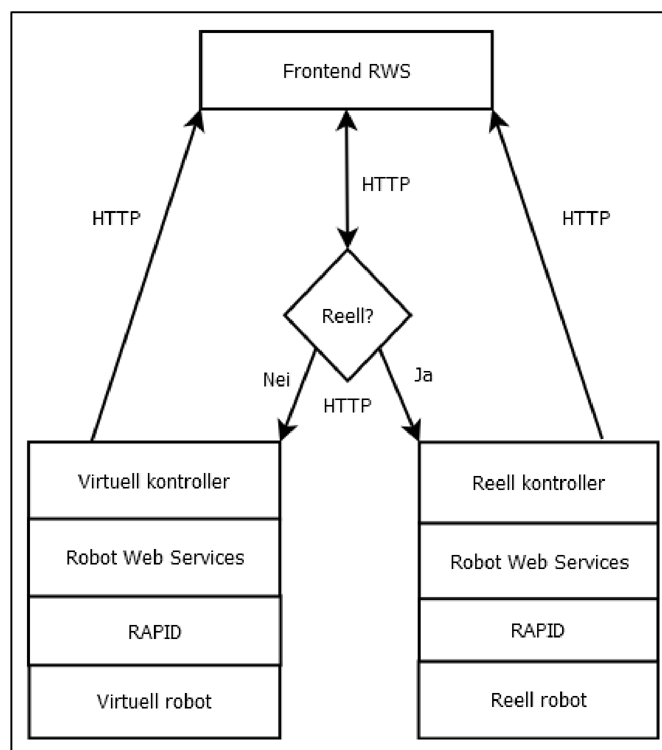
Dette delkapittelet vil gå gjennom oppbygningen og implementasjonen av API-et som implementeres i Python, og kommuniserer med robot gjennom Robot Web Services. Her vil

forskjellene mellom API-ene bli presentert og forklart. Kodestruktur blir ikke nevnt da delkapittelet 4.2.3 også er gjeldene her.

API-et som kommuniserer gjennom Robot Web Services blir videre referert til som frontend RWS.

4.3.1 Kommunikasjonsflyt

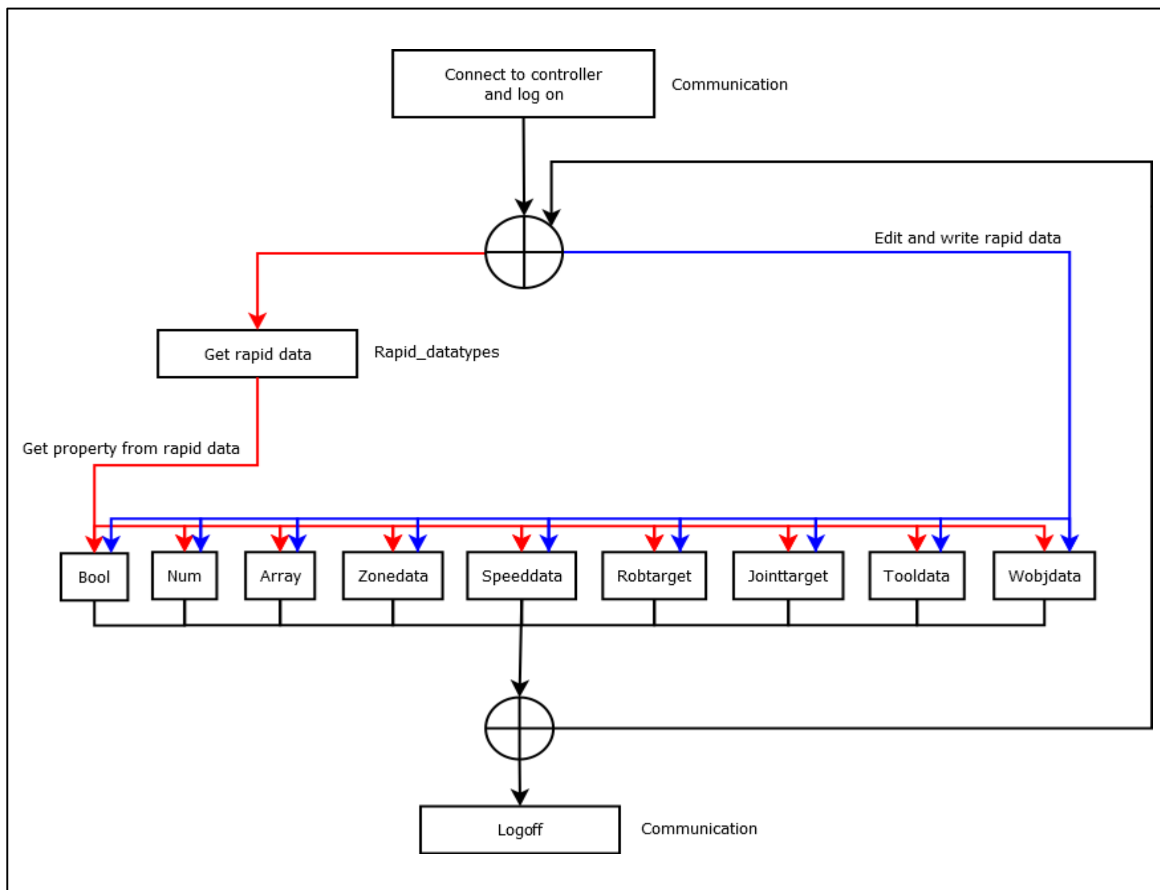
I dette delkapittelet blir det presentert et flytskjema for den overordnede kommunikasjon mellom frontend RWS og robot. Det blir i tillegg presentert et flytskjema som viser hvordan flyten er i et program som bygger på dette API-et. Flytskjemaene skal gjøre det enkelt å kunne se forskjellen på frontend RWS og frontend PCSDK.



Figur 13: Flyten fra frontend RWS til robot.

Øverst er frontend RWS som er implementert i Python. Programmene som brukeren lager kommuniserer med Robot Web Services, som kjører på kontroller, gjennom frontend RWS. Arkitekturstilen til Robot Web Services er REST, og RAPID-variablene på kontrolleren er dermed en ressurs. RAPID blir aksessert via Robot Web Services. Roboten utfører det som er

spesifisert i RAPID-programmet. Kommunikasjonen mellom frontend RWS og kontroller utføres med HTTP.



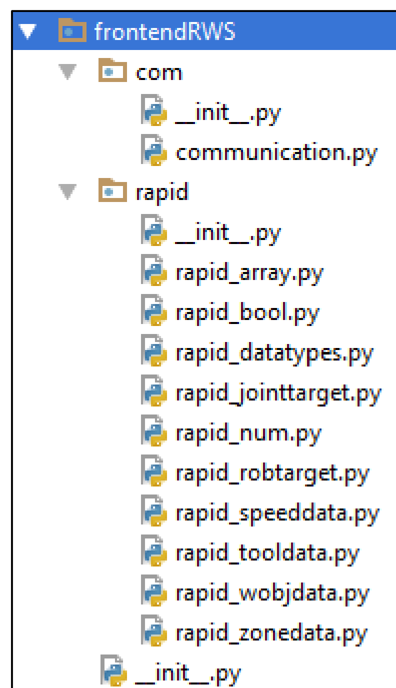
Figur 14: Flytskjema av modulenes bruk ved implementasjon av et program i frontend RWS.

Flytskjemaet over er annerledes sammenlignet med flytskjemaet til frontend PCSDK. Det første steget er å koble til en kontroller med en spesifisert IP-adresse. Det er ikke mulig å finne de tilgjengelige kontrollerne på nettverket gjennom Robot Web Services. For å finne tilgjengelige kontrollere som har Robot Web Services må Bonjour programvare brukes. Når det skal kobles til en kontroller så må også en bruker spesifiseres, forhåndsdefinert eller egendefinert. Det er ikke mulig å koble til kontroller uten å spesifisere en bruker slik som med frontend PCSDK. Neste steg er å behandle RAPID-variabler. En av de store forskjellene sammenlignet med frontend PCSDK er at det ikke er nødvendig å først hente RAPID-variabler for så å kunne behandle dem. Det er kun når egenskaper skal hentes at RAPID-variabelen må hentes først. Dette er representert med den røde linjen i skjemaet over. Om det er ønskelig å endre en variabel på kontrolleren så kan det gjøres uten å hente variabelen først. Dette representeres med den blå linjen i skjemaet over. Det er ikke nødvendig å få tak i

mastership da det skjer automatisk på kontrolleren når en variabel skal endres. *Mastership* blir også sluppet automatisk. Når programmet er ferdig med å behandle variabler så logges brukeren av. Problemene som er nevnt under kommunikasjonsflyt i frontend PCSDK vil ikke oppstå her. Minnelekkasje er unngått fordi Robot Web Services er tilstandsløs på grunn av REST, og en kontroller er dermed ikke lagret i minne. At for mange brukere er innlogget samtidig er også noe som kan oppstå ved bruk av PC SDK, men i Robot Web Services blir brukere logget ut etter en gitt tid.

4.3.2 Prosjektstruktur

Prosjektet er strukturert med to mapper. Den ene er kalt *com* og den andre *rapid*. Mappen med navn *user* er ikke opprettet fordi modulen *user_mastership* ikke er nødvendig og *user_authorization* er flyttet.



Figur 15: Prosjektstruktur til frontend RWS.

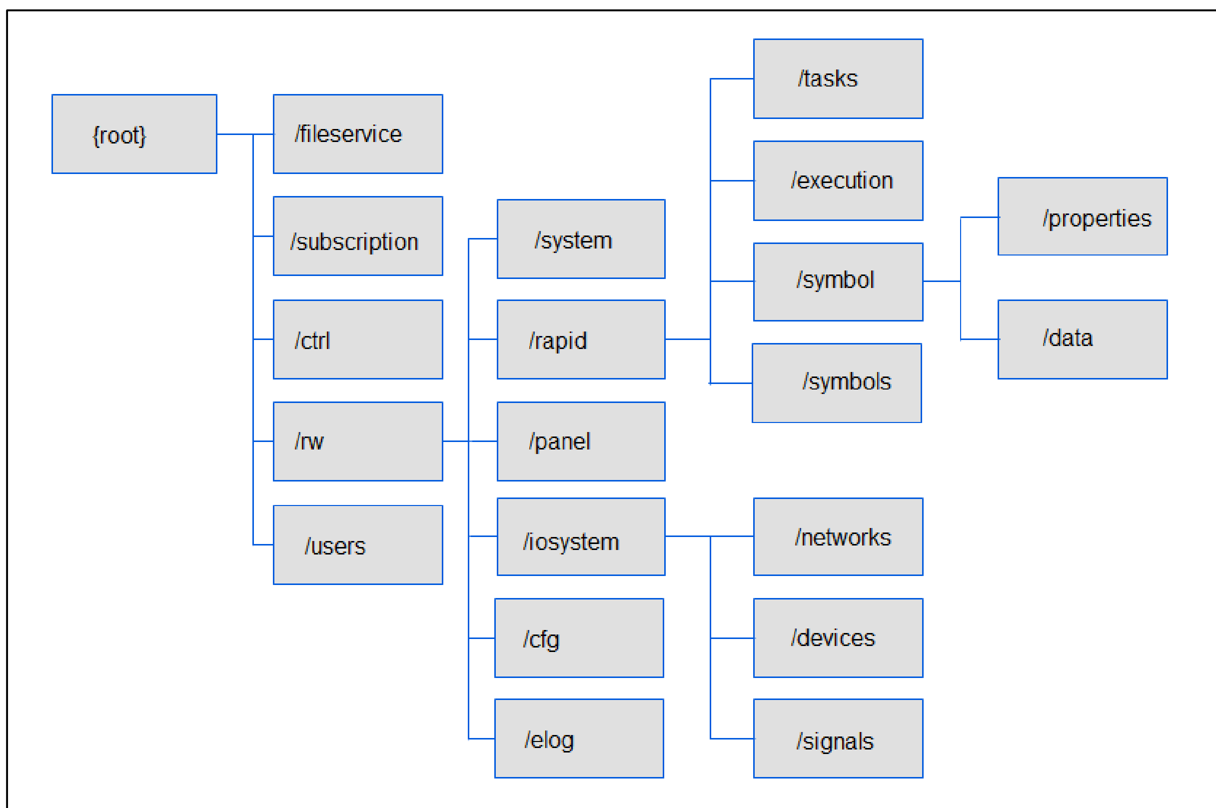
Mappen *com* inneholder kommunikasjonsmodulen slik som i frontend PCSDK. Forskjellen er at modulen med navnet *user_authorization*, som inneholdt funksjoner for pålogging og avlogging av en bruker, er en del av *communication*. Grunnen til dette er fordi en bruker må spesifiseres samtidig som det kobles til en kontroller. Mappen med navnet *rapid* inneholder

alle modulene som får tak i og behandler RAPID-datatyper. Navnene til filene tilsier hvilke RAPID-datatyper de kan behandle, og dette er likt som for frontend PCSDK. Forskjellen er at modulen med navnet *user_mastership* ikke lenger er definert. Det er ikke nødvendig, som nevnt ovenfor, å få tak i *mastership* eller slippe *mastership* da dette skjer internt på kontrolleren som kjører Robot Web Services.

4.3.3 Implementasjon

I likhet med frontend PCSDK vil et utvalg av moduler og funksjoner bli gjennomgått. De som utelates bygger i stor grad på de samme teknikkene som det som blir forklart. Det vil bli nevnt ulikheter og likheter mellom frontend PCSDK og frontend RWS.

En av de store forskjellene mellom frontend PCSDK og frontend RWS er at nettadresser må opprettes for å spesifisere stien til en ressurs på kontrolleren. Ressurser kan for eksempel være RAPID-variabler. Figuren under viser et utsnitt av hvordan Robot Web Services er strukturert.



Figur 16: Strukturen til Robot Web Services. Hentet fra [34].

Som et eksempel så vil nettadressen *http://127.0.0.1/rw* referere til *rw* på kontrolleren med IP-adressen *127.0.0.1*. Hva som skal gjøres med den refererte ressursen baserer seg på hvilket HTTP-verb som er brukt *GET*, *POST*, *PUT*, *DELETE*.

For å kunne håndtere nettadresser så må et HTTP-bibliotek brukes. Valget falt på tredjepartsbiblioteket *Requests* som er et enkelt HTTP-bibliotek med støtte for *digest*-autentisering, noe Robot Web Services krever da brukernavn og passord ikke skal sendes i klartekst. For å kunne aksessere en kontroller gjennom Robot Web Services må et verb spesifisere. I tillegg kreves det en nettadresse som spesifiserer hvilken ressurs som skal aksesseres og en *digest* med brukernavn og passord.

Svaret som robot-kontroller sender tilbake kan være strukturert i XML eller JSON format. Valget falt på JSON fordi det er et mer komprimert data utvekslingsformat sammenlignet med XML [35].

Overskriftene under refererer til modulnavnene i frontend RWS.

Communication

Modulen *communication* inneholder funksjoner for å koble seg til og logge på en kontroller, i tillegg til en funksjon for å logge av kontroller.

For å koble til en kontroller er det definert to funksjoner. Den ene funksjonen har navnet *connect_robot_with_ipaddr_def_user()* og tar inn IP-adressen til kontrolleren som argument. IP-adressen er en streng og kan spesifiseres på to måter. Den ene ved å skrive *local* som vil si *localhost:80*. Da kobles det til virtuell kontroller som kjører på den lokale datamaskinen med den forhåndsdefinerte brukeren. Om en IP-adresse er spesifisert så kobles det til kontrolleren med den spesifiserte IP-adressen, hvis den er gyldig, og logger på. Om *local* er spesifisert som argument i denne funksjonen så ser nettadressen slik ut:

```
http://localhost:80/rw/system?json=1
```

Kildekode 12: Eksempel på en nettadresse brukt ved oppkobling til robot-kontroller.

Nettadressen peker til *system* under *rw* hos *localhost:80*, og *?json=1* sier at resultatet skal returneres i formatet JSON. Grunnen til at *system* blir aksessert er fordi det er ønskelig å få informasjon om kontrolleren. Informasjonen som hentes ved oppkobling er RobotWare versjon og systemets navn. Før det er mulig å aksessere kontrolleren med nettadressen så må det opprettes en *digest* av brukeren som skal logges inn. Dette gjøres ved å kalle *HTTPDigestAuth()* fra *Requests* med brukernavn og passord som argumenter til funksjonen. I dette tilfellet er brukernavnet *Default User* og passordet *robotics*. For å aksessere kontroller brukes *Requests* med HTTP-verbet *GET*, som spesifiserer at informasjon skal hentes. Funksjonen ser slik ut *requests.get()* og tar inn nettadressen og *digest* som argumenter. Kontrolleren sender tilbake den ønskede informasjonen og en *cookie*. Det spesielle er at *cookie* representerer koblingen til kontroller, og den skal dermed være med i all kommunikasjon som utføres mot kontroller etter dette. *Cookie* og *digest* er blant det som blir returnert fra *connect_robot_with_ipaddr_def_user()*. Disse brukes videre i andre funksjoner.

```
is_con, res, digest, cookies = connect_robot_with_ipaddr_def_user('local')
```

Kildekode 13: Funksjon brukt til å koble til robot-kontroller i frontend RWS.

Den andre funksjonen har navnet *connect_robot_with_ipaddr_and_user()* og er implementert på samme måte som den ovenfor. Forskjellen her er at funksjonen tar inn brukernavn og passord som argument, slik at det skal være mulig å logge på med en egendefinert bruker. Dermed er forskjellen at *HTTPDigestAuth()* fra *Requests* tar inn det spesifiserte brukernavnet og passordet som argument.

Den siste funksjonen i modulen er *logoff_robot_controller()* og tar inn IP-adressen og *cookie* som argument. En nettadresse blir laget med IP-adressen. For eksempel hvis IP-adressen er *local* så blir nettadressen:

```
http://localhost:80/logout
```

Kildekode 14: Eksempel på nettadresse brukt til å logge av robot-kontroller.

Fra *Requests* kalles HTTP-verbet *GET* slik som sist, *requests.get()*. Nettadressen og *cookie* blir lagt inn i funksjonen som argumenter. Med denne nettadressen slettes *cookie* og koblingen er dermed borte. Forskjellige meldinger kan oppstå fordi *cookie* kan være for gammel og brukeren kan allerede være logget ut.

Rapid_datatypes

En av forskjellene mellom frontend PCSDK og frontend RWS er at funksjonene i det sistnevnte API-et tar inn flere argumenter. Dette er fordi REST og frontend RWS er tilstandsløse. Det vil si at funksjonene må vite hvilken kontroller som skal aksesseres, hva som skal aksesseres, *cookie* for å slippe å logge på, *digest* i tilfelle koblingen har endret seg og pålogging kreves på nytt. Selv om frontend PCSDK også er tilstandsløs så tar funksjonene inn *RapidData*-objektet som argument. Dette objektet har all nødvendig informasjon og har dermed en tilstand. På grunn av dette har frontend PCSDK mindre argumenter.

Modulen *rapid_datatypes* inneholder en funksjon. Denne funksjonen har navnet *get_rapid_data()* og henter informasjonen til en variabel fra RAPID på kontroller. Slik som andre funksjoner som skal kommunisere med kontroller, så tar også denne funksjonen inn IP-adresse, *cookie*, *digest*, variabelnavnet, og i hvilken modul og program på robot-kontroller denne variabelen befinner seg. Det første som gjøres er å lage nettadressene. I denne funksjonen lages det to nettadresser fordi informasjonen til variabelen er på to lokasjoner. Informasjonen til en RAPID-variabel, som for eksempel RAPID-datatype, er på en lokasjon, mens variabelens verdi er på en annen. Om *local* blir spesifisert som IP-adresse så vil nettadressen for å hente informasjonen til variabelen se slik ut:

```
http://localhost:80/rw/rapid/symbol/properties/RAPID/T_ROB1/MainModule/x?json=1
```

Kildekode 15: Eksempel på nettadresse brukt til å hente ut informasjon til en RAPID-variabel.

Fra argumentene til funksjonen er det i dette tilfellet oppgitt at programmets navn er *T_ROB1*, modulen sitt navn er *MainModule* og at det er *x* sin informasjon som skal hentes. Dette vil variere ut i fra hva som gis inn som argumenter. For å hente verdien til *x* så spesifiseres denne nettadressen;

```
http://localhost:80/rw/rapid/symbol/data/RAPID/T_ROB1/MainModule/x?json=1
```

Kildekode 16: Eksempel på en nettadressen som er brukt til å hente verdien til en RAPID-variabel.

Forskjellen på disse to nettadressene er at *properties* er byttet ut med *data*. Informasjonen og data hentes ved å bruke HTTP-verbet *GET*. Først hentes data ved å kalle *requests.get()* med den sistnevnte nettadressen og *cookie* som argumenter. Svaret som mottas blir sjekket for å se om det var en suksess eller ikke, og om det er en endring med *cookie*. Hvis en *cookie* blir returnert så er det den nye *cookie* som skal brukes videre. Eventuelt om svaret som blir mottatt sier at brukeren ikke er autorisert lenger, så må *requests.get()* kalles på nytt med nettadressen, *cookie* og *digest* som argumenter. *Cookie* som blir returnert i svaret fra kontroller blir brukt videre. For å hente informasjonen til RAPID-variabelen så gjøres dette på samme måte som nevnt ovenfor. Informasjonen og data som er av interesse blir hentet ut fra JSON-strukturene som kommer fra kontrolleren, og den blir lagret i en Python *dictionary*-datastruktur. Det som returneres fra funksjonen *get_rapid_data()* er blant annet en *dictionary* med informasjonen og data som er hentet fra kontrolleren, og en *cookie* som brukes videre i programmet.

```
got_num, num_dict, cookies = get_rapid_data('local', cookies, digest, 'T_ROB1',
                                            'MainModule', 'const_num')
```

Kildekode 17: Funksjon brukt til å hente informasjonen til en RAPID-variabel med frontend RWS.

Rapid_robtarget

Alle funksjonene som henter ut informasjon i modulen *rapid_robtarget,* tar inn *dictionary* som returneres fra funksjonen *get_rapid_data(). Dictionary* må inneholde informasjon om et robtarget for å kunne kalle disse funksjonene i denne modulen. Funksjonene henter ut og formaterer den ønskede informasjonen fra *dictionary* og returnerer den til brukeren. I frontend PCSDK blir *RapidData*-objektet tatt inn som argument til funksjonene, og med dette objektet kan den ønskede informasjonen hentes direkte fra kontrolleren.

Det finnes to funksjoner i *rapid_robtarget* som endrer et robtarget på kontrolleren. Den ene endrer en spesifisert egenskap, mens den andre gjør det mulig å endre hele robtarget.

Funksjonen som endrer en egenskap heter *edit_and_write_rapid_data_property()*, slik som i frontend PCSDK. Denne funksjonen tar inn argumenter som IP-adresse, *cookie*, *digest*, informasjon om hvor robtarget befinner seg, egenskapen som skal endres og hva den skal endres til. Alle funksjoner som endrer på RAPID-variabler i frontend RWS må definere nettadresser. Det lages en nettadresse som brukes når variabelen skal oppdateres på kontroller. I tillegg lages en nettadresse for å hente verdien til det ønskede robtarget. Først hentes verdien fra kontrolleren med denne nettadressen:

```
http://localhost:80/rw/rapid/symbol/data/RAPID/T_ROB1/MainModule/target?json=1
```

Kildekode 18: Eksempel på nettadresse brukt til å hente verdien til et robtarget.

Denne nettadressen er et eksempel der IP-adressen er spesifisert som *local*, programmet heter *T_ROB1*, modulen heter *MainModule* og robtarget har navnet *target*. Verdien som blir hentet formateres og legges i en *dictionary*. Den består da av verdiene til egenskapene som robtarget har, og brukes i oppbygningen av det nye robtarget. Det nye robtarget er en streng hvor strukturen ser identisk ut sammenlignet med strukturen til et robtarget i RAPID. Alle verdiene i *dictionary* blir satt inn i strengen på sine respektive plasser, unntatt verdiene til den egenskapen som skal endres. Der settes de nye verdiene inn. I frontend PCSDK er det mulig å hente ut egenskapene fra *RapidData*-objektet og sette inn i strengen. I frontend RWS må hvert enkelt element i hver egenskap settes inn. Dette fører til at oppbygningen av strengen blir mer komplisert. I delkapittel 4.2.4 er det mulig å se et eksempel på hvordan et robtarget ser ut.

For å kunne oppdatere en ressurs på kontrolleren så må data, som i dette tilfellet er et robtarget, sendes som form data i nettadressen. Form data blir laget ved å bruke en Python *dictionary* med navn/verdi-par, *{'value': robtarget}*. Her spesifiserer *value* at det er verdien som skal endres, altså verdien til hele robtarget, og den nye verdien er strengen som er blitt formatert og har fått navnet *robtarget*. For å oppdatere på kontrolleren brukes HTTP-verbet *POST* og kallet vil se slik ut; *requests.post()*. Argumentene til denne metoden er nettadressen som skal brukes til å oppdatere, *cookie* og *dictionary* med navn/verdi-par. Nettadressen som brukes til å oppdatere ser slik ut;

```
http://localhost:80/rw/rapid/symbol/data/RAPID/T_ROB1/MainModule/target?json=1&
action=set
```

Kildekode 19: Eksempel på nettadresse brukt til å oppdatere et robtarget i RAPID.

Den har kun en liten endring sammenlignet med nettadressen som er brukt til å hente verdien. Det er lagt til *action=set* på slutten av nettadressen og spesifiserer at noe skal endres. Funksjonen *edit_and_write_rapid_data_property()* returnerer en melding med resultatet og *cookie* som skal brukes videre i programmet. Argumentene *cookie* og *digest* er ikke beskrevet her da de blir behandlet slik som beskrevet over i *rapid_datatypes*.

```
res, cookies = edit_and_write_rapid_data_property('local', cookies, digest,
                                                  'T_ROB1', 'MainModule',
                                                  'target', 'trans', '[0,0,0]')
```

Kildekode 20: Funksjon brukt til å endre en egenskap i et robtarget med frontend RWS.

Funksjonen *edit_and_write_rapid_data()* gjør det mulig å endre hele robtarget. Argumentene som funksjonen tar inn er IP-adresse, *cookie*, *digest*, informasjon om hvor robtarget befinner seg og de nye verdiene til egenskapene. Forskjellen mellom denne funksjonen og den beskrevet ovenfor, er at kun en nettadresse er nødvendig fordi alle egenskaper skal endres. Det er dermed ikke ønskelig å hente ut verdien til robtarget på kontrolleren. Nettadressen som brukes til å endre robtarget i denne funksjonen har identisk oppbygning sammenlignet med nettadressen i den forrige funksjonen. Denne funksjonen bygger på de samme teknikkene som den ovenfor.

Rapid_array

Modulen behandler endimensjonale lister av datatypen num.

Funksjonen *edit_and_write_rapid_data_num_index()* endrer en num verdi i den spesifiserte indeksen i en liste. Argumentene til funksjonen er IP-adresse, *cookie*, *digest*, informasjon som navnet til listen, hvor den befinner seg i RAPID og indeksen som skal oppdateres, i tillegg til den nye verdien. Denne funksjonen bruker også to nettadresser. Den ene blir brukt til å hente ut listen fra kontrolleren og den andre blir brukt til å endre listen. Nettadressene har samme

oppbygning som de som er beskrevet under modulen *rapid_robtarget,* gitt at programmet har navnet *T_ROB1* og modulen har navnet *MainModule*. Den eneste forskjellen er at variabelnavnet *target* må byttes ut med navnet til listen som skal aksesseres. Listen må først hentes med HTTP-verbet *GET* slik som dette; *requests.get()*. Argumentene som brukes i dette kallet er nettadressen som skal brukes til å hente listen og *cookie*. Listen hentes ut fra JSON-strukturen i svaret fra kontroller, og formateres til en liste i Python. Deretter blir listen i Python endret ved at den spesifiserte indeksen får den nye verdien fra argumentet. Listen blir deretter formatert til en streng slik at den er identisk i oppbygning sammenlignet med en liste i RAPID. Før listen kan oppdateres i RAPID så må form data lages med navn/verdi-par. Det er gjort slik som beskrevet tidligere, bare at verdien nå er strengen som representerer listen. HTTP-verbet POST blir brukt til å skrive strengen til RAPID. Kallet ser slik ut; *requests.post()*. Argumentene til kallet er nettadressen som skal brukes til oppdatering, *cookie* og form data. Dette er mer komplisert sammenlignet med frontend PCSDK. For å endre en verdi med gitt indeks i frontend PCSDK så måtte kun *WriteItem()* kalles på *RapidData*-objektet, med verdien og indeksen som argument. Funksjonen *edit_and_write_rapid_data_num_index()* returnerer en melding med resultatet og *cookie* som skal brukes videre i programmet. Argumentene *cookie* og *digest* er behandlet slik som beskrevet over i *rapid_datatypes*.

```
res, cookies = edit_and_write_rapid_data_num_index('local', cookies, digest,
                                                    'T_ROB1', 'MainModule',
                                                    'array', 0, 1)
```

Kildekode 21: Funksjon brukt til å endre en variabel i en liste med gitt indeks gjennom frontend RWS.

Gjennom dette delkapittelet er et utvalg av moduler og funksjoner beskrevet slik som for frontend PCSDK. Dette delkapittelet har vist at enkle funksjoner er laget der brukeren ikke trenger å ha kunnskap om Robot Web Services, som igjen gjør dette tidsbesparende. All logikk som gjør det mulig å kommunisere med robot gjennom Robot Web Services ligger i funksjonene. På grunn av dette er det også enkelt for en bruker å lære seg hvordan det kommuniseres gjennom grensesnittet ved å studere kildekoden.

Mappe	Modulnavn	Funksjoner
Com	Communication	Connect_robot_with_ipaddr_and_user, Connect_robot_with_ipaddr_def_user, Logoff_robot_controller
Rapid	Rapid_array	Get_length_array, Get_dimensions_array, Edit_and_write_rapid_data_num_index, Edit_and_write_rapid_data_num
	Rapid_bool	Get_state_tostring, Get_state, Edit_and_write_rapid_data
	Rapid_datatypes	Get_rapid_data
	Rapid_jointtarget	Get_robax_tostring, Get_extax_tostring, Get_jointtarget_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_num	Get_value_tostring, Get_value, Edit_and_write_rapid_data
	Rapid_robtarget	Get_trans_tostring, Get_rot_tostring, Get_robconf_tostring, Get_extax_tostring, Get_robtarget_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_speeddata	Get_speeddata_tostring, Edit_and_write_rapid_data_base, Edit_and_write_rapid_data
	Rapid_tooldata	Get_robhold_tostring, Get_tframe_tostring, Get_tload_tostring, Get_tooldata_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	Rapid_wobjdata	Get_robhold_tostring, Get_ufprog_tostring, Get_ufmec_tostring, Get_uframe_tostring, Get_oframe_tostring, Get_wobjdata_tostring, Edit_and_write_rapid_data_property, Edit_and_write_rapid_data
	rapid_zonedata	Get_zonedata_tostring, Edit_and_write_rapid_data_base, Edit_and_write_rapid_data

Tabell 2: Liste av modulene i frontend RWS og de tilhørende funksjonene.

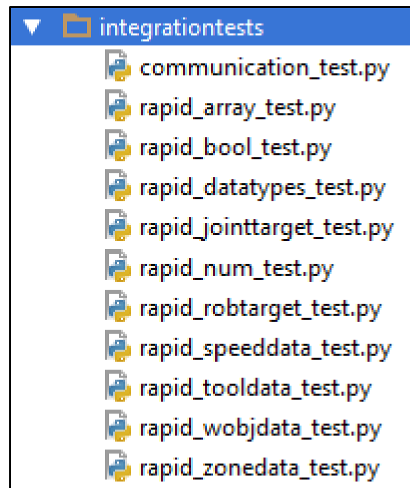
4.3.4 Verifisering av implementasjon

Det er viktig at koden som er skrevet er av samme kvalitet som i frontend PCSDK. På grunn av dette ble Flake8 brukt også her. Koden er endret basert på resultatet av den statiske kodeanalysen.

McCabe kompleksitetstest er også utført. Det er viktig å få en indikasjon på hvor kompleks koden er. Testen ble satt til å rapportere kompleksitet som oversteg ti, slik som for frontend PCSDK. Det viste seg at også i denne implementasjonen er funksjonene som endrer RAPID-variabler de mest komplekse. I tillegg så er kompleksiteten også litt høyere sammenlignet med frontend PCSDK på grunn av at *cookie* blir behandlet. Dette medfører ekstra veivalg i kildekoden. Det er ikke gjort endringer for å få færre sjekker i funksjonene. Grunnen til dette er fordi funksjonene skal gi best mulig tilbakemelding til brukeren. Slik som for frontend PCSDK så er kildekoden gjennomgått for å se om det er mulig å forenkle i form av å redusere veivalg, uten at det fører til dårligere tilbakemeldinger. Det ble funnet veivalg i funksjoner som ved flytting reduserte antall veivalg, uten å føre til dårligere tilbakemeldinger. Endringer som dette ble utført.

Testing av implementasjonen er en viktig del av utviklingen slik som for frontend PCSDK. Alt avhenger av noe annet i kommunikasjon med robot og dermed er også frontend RWS testet med en form for integrasjonstester. Testmetoden som er brukt er hvit boks testing slik som for frontend PCSDK.

I implementasjonen av integrasjonstestene er Python-biblioteket *unittest* brukt. Grunnene er de samme som nevnt for frontend PCSDK. Det er ønskelig med den samme fleksibiliteten og kjøremåte, der en eller flere tester kan kjøres. Integrasjonstestene skal teste at funksjonene i API-et gir ønsket resultat, basert på data som er gitt inn.



Figur 17: Mappe med integrasjonstester til frontend RWS.

I kildekoden nedenfor testes skrivefunksjonen i modulen *rapid_num*. Sammenlignet med testen i frontend PCSDK så er færre funksjonskall i denne testen. I oppsettet før testen er det nødvendig å koble til og logge på kontroller. Dette gjøres med et funksjonskall. Det er ikke nødvendig å ha ekstra oppsett for testen da *mastership* skaffes automatisk. Deretter kjøres testen under. Etter at testen er kjørt spesifiseres det en ekstra opprydning slik som i frontend PCSDK, der variabelen blir endret tilbake til opprinnelig verdi. Dette gjøres fordi testen da kan kjøres på nytt. Det er ikke nødvendig å slippe *mastership* da dette også skjer automatisk. Felles for alle tester er at utlogging skjer under opprydningen.

```
# Tests edit_and_write_rapid_data with correct input data
def test_edit_and_write_rapid_data_correct(self):
    """ Tests edit_and_write_rapid_data with correct input data """
    res, self.cookies = rapid_num.edit_and_write_rapid_data('local',
                                                            self.cookies,
                                                            self.digest_auth,
                                                            'T_ROB1',
                                                            'MainModule',
                                                            'var_number', 10)
    self.assertEqual(res, 'Value updated.')
```

Kildekode 22: Integrasjonstest til skrivefunksjonen i modulen rapid_num i frontend RWS.

Det er ønskelig å kjøre kodedekning (*code coverage*) også her for å sjekke hvor mye av koden i modulene som blir kjørt ved testing for å minimere feil. Dette er utført ved å bruke

kodedekningsverkøyet *Coverage.py*. Det er brukt to typer kodedekning; programinstruksdekning (*statement coverage*) og forgreiningsdekning *(branch coverage*).

Først er det kjørt programinstruksdekning i integrasjonstestfilene for å sjekke at alle kodelinjer kjøres som forventet. Resultatet til programinstruksdekningen gav en indikasjon på at ikke alle kodelinjer ble kjørt. Feilen var at opprydningen ikke ble utført som ønsket. Dette ble fikset. Det er viktig å vite at alt fungerer i test-filene slik at det ikke resulterer i feil ved testing av modulene.

Forgreiningsdekning er brukt til å teste modulene for å se hvor mange prosent av koden i modulene som blir kjørt. Dette er en viktig teknikk i hvit boks testing [33]. Ved å bruke forgreiningsdekning så sjekkes hver vei ved et veivalg. Dette er viktig fordi det er mange veivalg i implementasjonen av API-et.

Integrasjonstest	Dekningsgrad
Communication_test	85%
Rapid_array_test	78%
Rapid_bool_test	80%
Rapid_datatypes_test	68%
Rapid_jointtarget_test	80%
Rapid_num_test	86%
Rapid_robtarget_test	79%
Rapid_speeddata_test	81%
Rapid_tooldata_test	81%
Rapid_wobjdata_test	82%
Rapid_zonedata_test	84%

Tabell 3: Dekningsgraden til integrasjonstester som tester frontend RWS.

Integrasjonstestene viser høy dekningsgrad. Resultatet viser at veivalg der *cookie* er utløpt og eventuelle uventede feil på kontroller ikke er testet. Dette gjør at dekningen er lavere for noen moduler som for eksempel *rapid_datatypes*. Resultatet viser at integrasjonstestene dekker de viktigste delene av implementasjonen. Det å få rundt 80 % dekning er sett på som tilstrekkelig og bra [36].

Alle integrasjonstester er kjørt på virtuell kontroller med RobotWare 6.01 og 6.02 for å se at alt fungerte som ønsket. Resultatet viste at alt fungerte. Noen vilkårlige tester ble også kjørt på reell kontroller som resulterte i det samme som på virtuell kontroller.

5. Resultat

I dette kapittelet blir testene og resultatet presentert. Testingen er utført slik at viktige deler av API-ene blir testet for å finne styrker og svakheter. Det er utforsket hvordan arkitekturstilen REST testes. Testing med hensyn til responstid gjentar seg flere steder og er dermed inkludert [37-39]. Viktige deler inkluderer dermed blant annet oppkoblingstider, responstider og påliteligheten til koblingen. Sammenlignet med oppgavebeskrivelsen så er det lagt til flere tester for å få en grundig testing av API-ene. Dette gir et bedre grunnlag til analysen i delkapittel 6.2.

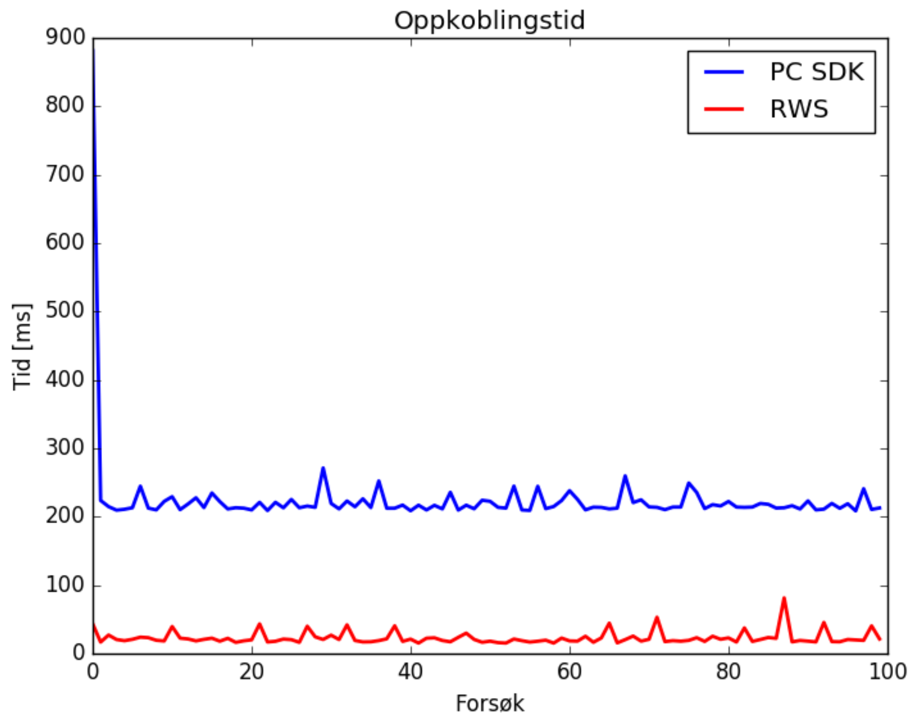
Oppgavebeskrivelsen spesifiserer at det skal blant annet testes basert på synkronisering. Det som menes med synkronisering i denne sammenhengen, er at det brukeren foretar seg i et API gjenspeiles på robot-kontroller. For eksempel at en funksjon som endrer en variabel faktisk endrer den på robot-kontroller. At synkronisering fungerte ble sett på som et minimumskrav, og er allerede testet i integrasjonstestene for begge API-ene. Synkronisering er ikke tatt med videre i dette kapittelet.

Kommunikasjon med robot-kontroller er utført over trådløs tilkobling. Under testing er det utført overvåkning på datamaskinen. Prosessor, minne og nettverkstilkoblingen ble overvåket for å sikre at ikke andre applikasjoner hadde betydelig innvirkning på resultatet.

5.1 Oppkoblingstider

En oppkobling defineres som tiden det tar før det er mulig å for eksempel utføre endring eller lesing av RAPID-variabler på robot-kontroller. I frontend PCSDK innebærer det å finne kontrollere på nettverket, koble til og deretter logge på en bruker. I frontend RWS må en kontroller tilkobles og en bruker må logges på. Det er ikke nødvendig å finne kontrollere på nettverket.

For å få en presis tid ble Python-biblioteket *time* brukt. Funksjonen *time.clock()* gir tider som er mer presise enn et mikrosekund i *Microsoft Windows* [40]. Oppkobling ble utført 100 ganger med hvert API. For at en oppkobling er gyldig må alle operasjoner være vellykket. Hvis ikke blir testen stoppet og må kjøres på ny.



Figur 18: Graf av resultatet etter oppkoblingstest.

	Frontend RWS	*Frontend PCSDK*
Minimum	15 ms	208 ms
Maksimum	81 ms	881 ms
Gjennomsnittlig	23 ms	225 ms

Tabell 4: Data fra oppkoblingstest.

5.2 Inaktivitetstest

Inaktivitetstest tester en kobling etter en gitt tid. Dette er av interesse fordi et brukerprogram kan for eksempel være kommandobasert, og det kan ta tid for brukeren å legge inn data. Andre grunner kan være at programmet skal utføre tidkrevende prosesser eller at brukeren er inaktiv over en tidsperiode. Det er viktig at koblingen ikke brytes og at programmet ikke terminerer med en feil etter kort tid. Dette kan føre til at progresjon forsvinner og kan føre til irritasjon på grunn av at programmet må kjøres på nytt.

Test-programmet kobler til robot-kontroller og logger på, for så å vente en gitt tid før en variabel av RAPID-datatypen num oppdateres. Ventetidene som er valgt er 15, 30 og 60

minutter. Det er ikke valgt å teste med ventetider utover dette da det er sett på som godt nok om en kobling fungerer opptil en time.

I frontend RWS så vil koblingen være borte da *cookie* vil utløpe. Det er dermed interessant å se om koblingen kan automatisk etableres igjen uten problem i det en variabel skal oppdateres etter den spesifiserte tiden. Om dette er mulig så vil koblingen klassifiseres som OK.

	Frontend RWS	*Frontend PCSDK*
15 minutter	OK	OK
30 minutter	OK	OK
60 minutter	OK	OK

Tabell 5: Resultatet av inaktivitetstestingen.

5.3 Koblingspålitelighet

Koblingspålitelighetstest skal teste om koblingen til kontrolleren er robust. Denne testen tegner en blomst med robot der ett og ett punkt blir kalkulert, og oppdatert i RAPID-programmet på kontrolleren. Robot går til hvert punkt som blir kalkulert. Koordinatene kalkuleres med disse formlene:

$$X = amplitude * \cos(k\theta)\cos(\theta)$$
$$Y = amplitude * \cos(k\theta)\sin(\theta)$$

Variabelen *k* spesifiserer hvor mange blader blomsten har. Om k er et partall så blir det tegnet 2k blader. Hvis k er et oddetall så blir det tegnet k blader. I testen er k satt til fire. Theta er den nåværende graden og økes for hvert punkt som kalkuleres. Den starter på null og går opp til 360 grader når k er et partall. Når k er et oddetall så går den fra null til 180 grader. Theta blir inkrementert med to grader om gangen, og når theta har maks antall grader så er en blomst tegnet. Amplitude er en faktor som spesifiserer hvor lange bladene skal være og er satt til 100.

I RAPID-programmet er en boolsk variabel definert for å indikerer om roboten tegner. Det er også definert en boolsk variabel som indikerer om robot venter på et nytt punkt. I Python blir disse sjekket kontinuerlig og oppdatert.

Ved å tegne en blomst på denne måten blir koblingen brukt kontinuerlig over en tidsperiode. Python sjekker variablene i RAPID programmet kontinuerlig, og gir nye koordinater til robot basert på hva de boolske variablene indikerer.

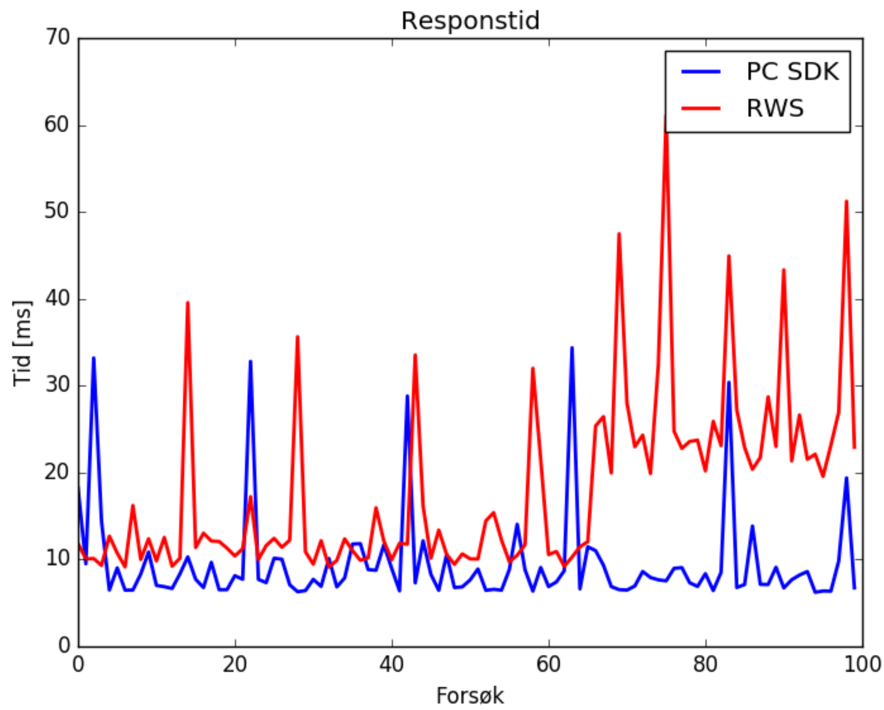
Denne testen skal se om koblingen ble brutt eller om feil forekommer etter hvert som kommunikasjon pågår.

	Frontend RWS	*Frontend PCSDK*
En blomst	OK	OK
To blomster	OK	OK
Tre blomster	OK	OK
Ti blomster	OK	OK

Tabell 6: Resultatet av koblingspålitelighetstest.

5.4 Responstider

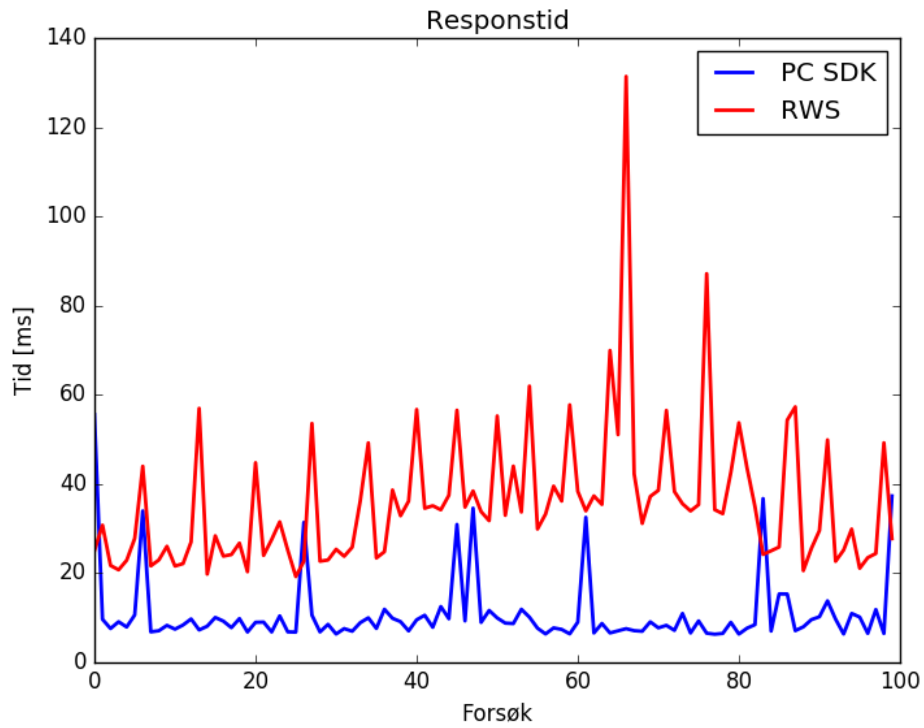
Det er implementert tre responstidtester. Disse testene skal avdekke hvilket API som har lavest responstider. Først er det testet hvor lang tid det tar å endre en num. Deretter er det testet hvor lang tid det tar å endre to elementer i en liste av num. Til slutt er det testet hvor lang tid det tar å endre fire elementer i en liste av num. Disse tre testene er kjørt 100 ganger hver og tiden er tatt ved å bruke Python-biblioteket *time* med funksjonen *clock()*.



Figur 19: Graf med responstider ved endring av num.

	Frontend RWS	*Frontend PCSDK*
Minimum	9 ms	6 ms
Maksimum	61 ms	34 ms
Gjennomsnittlig	17 ms	9 ms

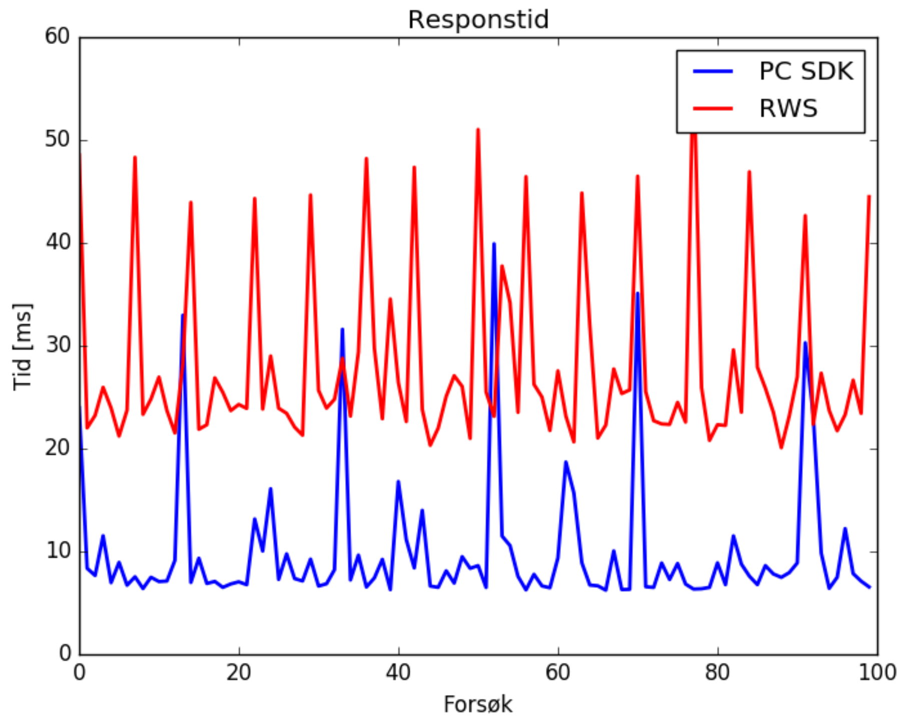
Tabell 7: Data fra responstidtest ved endring av num.



Figur 20: Graf med responstider ved endring av to elementer i en num liste.

	Frontend RWS	*Frontend PCSDK*
Minimum	19 ms	6 ms
Maksimum	131 ms	55 ms
Gjennomsnittlig	35 ms	10 ms

Tabell 8: Data fra responstidtest ved endring av to elementer i en num liste.



Figur 21: Graf med responstider ved endring av fire elementer i en num liste.

	Frontend RWS	*Frontend PCSDK*
Minimum	20 ms	6 ms
Maksimum	58 ms	39 ms
Gjennomsnittlig	28 ms	9 ms

Tabell 9: Data fra responstidtest ved endring av fire elementer i en num liste.

5.5 Kjøretider til blomsterprogram

Blomsterprogrammet er det samme programmet som ble kjørt under koblingspålitelighetstesten. Det skal brukes til å sammenligne kjøretiden til API-ene.
Det ble tegnet tre blomster. Tiden ble målt med Python-biblioteket *time* slik som beskrevet tidligere.

	Frontend RWS	*Frontend PCSDK*
Test en	161 sekunder	141 sekunder
Test to	164 sekunder	135 sekunder
Test tre	164 sekunder	135 sekunder

Tabell 10: Resultatet av kjøretider til blomsterprogram.

6. Diskusjon

Dette kapittelet vil først diskutere resultatene som er presentert ovenfor. Deretter blir det utført en SWOT-analyse av API-ene for å oppsummere og sammenligne dem. Til slutt blir det presentert hva som kan være videre arbeid.

6.1 Diskusjon av resultatene

Oppkoblingstider

Fra grafen og tabellen i delkapittel 5.1 kommer det frem at frontend RWS har en lavere oppkoblingstid sammenlignet med frontend PCSDK. Maksimum i frontend RWS er lavere enn minimum i frontend PCSDK. Ut i fra gjennomsnittet er oppkoblingstiden med frontend PCSDK nesten ti ganger høyere. Grunnen til den store forskjellen i oppkoblingstider er fordi frontend PCSDK må gjøre mer arbeid før oppkoblingen er utført. Først må det bli skannet for kontrollere på nettverket, for så å koble til den ønskede kontrolleren. Deretter må det logges inn med en bruker. Det må utføres tre funksjonskall før oppkoblingen er utført. I frontend RWS er det kun nødvendig med ett funksjonskall til den spesifiserte kontrolleren med brukeren som skal logges inn. I grafen har frontend PCSDK høyere oppkoblingstid i første forsøk. Selv om testen ble kjørt flere ganger så var alltid det første forsøket høyere enn resten av forsøkene. Med stor sannsynlighet er det lastingen av *DLL*-filene som det blir referert til i kildekoden som resulterer i høyere oppkoblingstid. Dette medfører at et program som utfører oppkobling en gang, alltid vil få maksimum oppkoblingstid. Om et program skal koble til kontroller flere ganger så kan oppkoblingstidene bli lavere slik som i grafen, fordi DLL-filene allerede er lastet. Testen viser at frontend RWS har lavere oppkoblingstider enn frontend PCSDK.

Inaktivitetstest

Tabellen i delkapittel 5.2 viser at det ikke vil oppstå problem om et program ikke kommuniserer kontinuerlig med robot-kontroller. Det er mulig å lage programmer som kommuniserer sjeldent, eller programmer som bruker en del tid før det sender noe til kontroller etter oppkobling. Selv om det ikke er testet med ventetid over 60 minutter så er det stor sannsynlighet for at det også vil fungere. Det viser seg at både frontend RWS og frontend PCSDK gir samme resultat og er dermed like i denne testen.

Koblingspålitelighet

API-ene fikk resultatet ok i alle forsøkene. Frontend PCSDK utførte testene uten problemer og fikk ok som resultat. Frontend RWS hadde en begrensning. Blomstene ble tegnet og resultatet ble ok, men problemet er at roboten stoppet opp og ventet en stund før den fortsatte å tegne. Det skjedde tilfeldig, men som regel etter at roboten hadde tegnet en liten stund. Under det ene forsøket ble feilmeldingen «A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond» returnert, og etter en liten stund fortsatte roboten. Under testing på virtuell robot har det samme hendt bare at feilmelding tilsa at buffer er overfylt. Med stor sannsynlighet skjer det samme på reell robot når feilmeldingen ovenfor blir returnert. Robot-kontroller kan ikke ta imot flere forespørsler. For å teste dette ble det lagt inn en vilkårlig ventetid på 100 ms mellom hver gang det blir sjekket om robot er klar for et nytt punkt. Dette førte til at roboten ikke stoppet opp over en lengre periode, og ingen feilmeldinger oppstod. Det er dermed viktig å ikke overfylle buffer med forespørsler når det sjekkes for forandring på robot. Det må sjekkes i intervaller. Intervalltiden må angis basert på hva som utføres på robot. Det viktigste er at robot får tid til å behandle alle forespørslene. Denne testen viser at frontend PCSDK er mer robust.

For å unngå at robot-kontroller blir overbelastet av forespørsler så er det korrekt å bruke *Web Sockets* i frontend RWS. *Web Sockets* åpner for muligheten til å abonnere på blant annet variabler. Om en variabel blir forandret så får programmet vite det over en kanal slik at spørring i intervaller blir unødvendig. *Web Sockets* er utforsket men ikke implementert fordi ingen god løsning er funnet. Løsningene ville komplisert frontend RWS for brukeren, noe som ikke var ønskelig.

Responstider

Ved endring av num har frontend RWS i de fleste forsøk høyere responstid enn frontend PCSDK. Frontend PCSDK har lavest minimum og maksimum responstid. Maksimum responstid til frontend RWS er opptil to ganger høyere. Den viktigste målingen er gjennomsnittlig responstid fordi kommunikasjon med robot utføres kontinuerlig. Gjennomsnittlig skiller det en del. Frontend PCSDK er nesten dobbelt så raskt over 100 forsøk ved endring av num. Dette fører til at frontend PCSDK er best i dette scenariet.

Ved endring av to elementer i en num liste har frontend RWS også her høyere responstider i de fleste forsøk. Minimum responstid er litt over tre ganger høyere for frontend RWS sammenlignet med frontend PCSDK. Maksimum responstid er over to ganger høyere og gjennomsnittlig responstid er over tre ganger høyere. Det er gjennomsnittlig responstid som er viktigst som nevnt ovenfor. Frontend PCSDK er bedre i dette scenariet også.

Ved endring av fire elementer i en num liste så har frontend PCSDK også i dette scenariet lavere responstider enn frontend RWS. Minimums responstid er over tre ganger høyere for frontend RWS. Maksimum responstid er ca. 45 % høyere og gjennomsnittlig responstid er over tre ganger høyere.

Basert på alle scenariene konkluderes det at frontend PCSDK har de beste responstidene. Frontend PCSDK har både lavest minimum, maksimum og gjennomsnittlig responstid i alle scenarier. Grafene til hvert API har varierende responstider og variasjonen mellom forsøkene kan skyldes at testingen ble utført på trådløs tilkobling. En trådløs tilkobling kan påvirkes av faktorer som avstand, dekning og interferens. Overvåkningen av datamaskinen viste ikke tegn til at andre applikasjoner hadde innvirkning på resultatene.

Kjøretider til blomsterprogram

Frontend PCSDK var raskest som forventet. Grunnen til det er fordi den største delen av programmet omhandler å kommunisere med robot-kontroller. Når det gjelder responstider så er frontend PCSDK raskest slik som resultatet fra tidligere testing viser. Oppkobling til kontroller utføres bare en gang, og det er i det tilfellet frontend RWS er raskest med god margin. Under kontinuerlig kommunikasjon med robot-kontroller er det også verdt å påpeke at robot kan stoppe en stund, ved at buffer blir overfylt ved bruk av frontend RWS. Dette er beskrevet i koblingspålitelighetstesten. I denne testen er det ikke lagt inn ventetid mellom hver gang en variabel sjekkes på robot-kontroller. Dette ville ført til enda lenger kjøretid med frontend RWS.

6.2 Analyse av API-ene

Det er utført en SWOT-analyse på API-ene for å liste styrker, svakheter, muligheter og trusler. Analysen skal gi en ryddig oppsummering av API-ene.

Frontend PCSDK

Styrker	Svakheter
• Robust. • Gode responstider. • Enklere å bruke.	• Plattform-avhengig. • Problemer med tredjepartsbiblioteker. • Vanskelig å installere tredjepartsbiblioteker. • Høyere oppkoblingstid enn frontend RWS.
Muligheter	**Trusler**
• Mono.	

Tabell 11: Analyse av frontend PCSDK.

Frontend RWS

Styrker	Svakheter
• Plattform-uavhengig. • God støtte for tredjepartsbiblioteker. • Lavere oppkoblingstid enn frontend PCSDK.	• Vanskeligere å implementere på grunn av *cookie*. • Ikke like robust som frontend PCSDK. • Høyere responstider. • Vanskeligere å bruke på grunn av flere argumenter i funksjonene. • Robot Web Services er nytt og dokumentasjonen er enda mangelfull.
Muligheter	**Trusler**
• Styre robot eksternt på grunn av Robot Web Services.	• Uønskede kan koble til robot på grunn av Robot Web Services.

Tabell 12: Analyse av frontend RWS.

Testingen resulterte i at frontend PCSDK er et mer robust API med gode responstider sammenlignet med frontend RWS. Det er også lettere å bruke fordi funksjonene ikke tar inn like mange argumenter. I tillegg er det ingen *cookie* som må behandles riktig. Svakhetene til frontend PCSDK er plattform-avhengighet til *Windows* på grunn av at PC SDK er bygget på .NET. Oppkoblingstiden er også høyere fordi DLL-filene til PC SDK må lastes. I tillegg er det problem med tredjepartsbibliotekstøtte i IronPython. Grunnen er fordi tredjepartsbiblioteker kan bruke *CPython*-moduler som ikke er en del av IronPython. *CPython*

er den originale Python implementasjonen [41]. For å installere tredjepartsbiblioteker må kommandovinduet brukes i *Windows*. Installasjon av for eksempel tredjepartsbiblioteket *Coverage.py* resulterte i problemer. Det ble løst ved at installasjonsverktøyene *pip* og *setuptools* ble slettet fra installasjonsmappen til IronPython, og en eldre versjon av disse ble installert. Dette førte til at installasjon av biblioteker er mulig. Selv om *Coverage.py* ble installert riktig så oppstod det feilmeldinger når funksjoner ble kalt.

Plattform-uavhengighet kan oppnås ved å bruke *Mono*-rammeverket. *Mono* åpner for muligheten til å kjøre IronPython på Linux og Mac OS X [42].

En av styrkene til frontend RWS er plattform-uavhengighet. Det er mulig å kjøre på Linux og Mac OS X uten ekstra arbeid. Det er også god støtte for tredjepartsbiblioteker på grunn av at Python er brukt. Svakheten er at det er vanskeligere å implementere og bygge videre på dette API-et fordi *cookie* må håndteres i implementasjonen. Brukeren må også gi inn flere argumenter i funksjonene på grunn av at Robot Web Services og REST er tilstandsløse. På grunn av at *Web Sockets* ikke er implementert så er det også mindre robust sammenlignet med frontend PCSDK.

Dokumentasjonen til Robot Web Services kan være mangelfull fordi det er nytt og kom ut med RobotWare versjon 6.0. Det er funnet feil der HTTP-responskoden som ble returnert fra kontroller ikke er den som er spesifisert i dokumentasjonen. Dette kan føre til at det er vanskeligere å arbeide med Robot Web Services. PC SDK er enklere å bruke da dokumentasjonen er god og lett å søke i.

Robot Web Services åpner for muligheten til å styre robot fra en ekstern lokasjon. Det som forutsettes er at roboten er koblet til internett. Herav følger en trussel der uønskede kan trenge seg inn og få tak i informasjon eller ta styring over robot. Angrep som *denial of service* (DoS) kan også utføres mot robot ved å overbelaste kontrolleren slik at robot blir utilgjengelig. Hvis roboten er i automatisk modus så kan inntrenger utføre handlinger uten godkjenning. Manuell modus utelukker dette da handlinger må godkjennes. Ved å kun koble roboten til et intranett så kan Robot Web Services brukes lokalt uten eksterne trusler.

6.3 Videre arbeid

Det er mye som kan gjøres i videre arbeid. Implementasjonen har stort sett tatt for seg RAPID del. API-ene kan utvides til å ta i bruk andre deler av robot-kontroller. Eksempler på andre deler er *event log* og *IO system*. Utvidelsen må basere på hvilken informasjon som er ønskelig å kunne behandle.

For å gjøre frontend RWS mer robust så kan det implementeres *Web Sockets*. Som nevnt tidligere så ble det ikke funnet en god løsning på dette uten å gjøre API-et mer avansert for brukeren. Noe som ikke var ønskelig.

Frontend PCSDK er låst til Windows. Som allerede nevnt åpner *Mono* for muligheten til å bruke IronPython på Linux og Mac OS X. Det er dermed mulig å gjøre API-et plattform-uavhengig, noe som ville opphevet en av de store svakhetene. *Mono* er ikke blitt utforsket i denne avhandlingen.

For å gjøre API-ene bedre er det mulig å videreutvikle basert på tilbakemeldinger fra brukere. API-ene er ikke brukertestet. Dette kan være med å bidra til enda mer brukervennlige API-er.

7. Konklusjon

API-ene forenkler kommunikasjon med robot og åpner for å kunne lage programmer raskere uten å måtte ha kunnskap om PC SDK eller Robot Web Services. Det som forutsettes er grunnleggende kunnskap om Python og ABB-roboter. Det som bestemmer hvilket API som er best må baseres på hva brukeren krever. Hvis Windows er den eneste plattformen som skal støttes og tredjepartsbiblioteker ikke er nødvendig så er frontend PCSDK det beste alternativet. Begrunnelsen er at det er mer robust, noe som er funnet gjennom testing. Om det dermed er ønskelig med plattform-uavhengighet og god støtte for tredjepartsbiblioteker så er frontend RWS det eneste alternativet. Dette er et godt alternativ, men det er viktig å ikke overbelaste robot-kontroller.

Referanser

[1] ABB. (2016, 2016.02.12). *RobotStudio*. Tilgjengelig: http://new.abb.com/products/robotics/robotstudio

[2] ABB. (2015, 2016.02.12). *RobotStudio Station and Rapid Sync*. Tilgjengelig: http://developercenter.robotstudio.com/BlobProxy/devcenter/RobotStudio/html/c582b5c6-af69-40b5-a2ec-ae4c354deb2c.htm

[3] ABB, "RobotStudio," 6.02.01 utg: ABB, 2016.

[4] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 92.

[5] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 20.

[6] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, ss. 22, 23.

[7] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 83.

[8] ABB. (2015, 2016.02.12). *RobotStudio Targets*. Tilgjengelig: http://developercenter.robotstudio.com/BlobProxy/devcenter/RobotStudio/html/0e0e4fab-77ee-4520-aff3-6f1a0f220df8.htm

[9] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 21.

[10] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, ss. 71, 142.

[11] ABB. (2016.03.04). *Controllers*. Tilgjengelig: http://new.abb.com/products/robotics/controllers

[12] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 15.

[13] ABB, "Operating Manual RobotStudio," Revision: D utg: ABB, 2010, s. 17.

[14] H. Berlin. (2012, 2016.02.26). *Text based robot programming made easy*. Tilgjengelig: http://www.abb.com/blog/gad00540/1DDE6.aspx?tag=RAPID%20programming

[15] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, ss. 65, 66.

[16] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, ss. 97, 98.

[17] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, ss. 85, 86, 87.

[18] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, s. 31.

[19] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, s. 73.

[20] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, ss. 105, 107, 108.

[21] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, s. 47.

[22] ABB, "RAPID Reference Manual," 3HAC 0966-13 utg, s. 7.

[23] ABB, "Application manual PC SDK," 3HAC036957-001 utg: ABB, 2012, s. 31.

[24] ABB. (2015, 2016.03.04). *Development Environment*. Tilgjengelig: http://developercenter.robotstudio.com/Index.aspx?DevCenter=RobotCommunication&OpenDocument&Url=html/43415bd9-1385-4439-a440-5745545b65c6.htm

[25] ABB. (2015, 2016.03.04). *Robot Communication Development*. Tilgjengelig: http://developercenter.robotstudio.com/Index.aspx?DevCenter=RobotCommunication&OpenDocument&Url=html%2fea7d0bad-2991-4c46-9053-56f3571d0929.htm

[26] P. F. Dubois, "ANest OF PYTHONS," *Computing in Science & Engineering,* vol. 7, s. 81, 2005.

[27] ABB. (2015, 2016.03.04). *Robot Web Services*. Tilgjengelig: http://developercenter.robotstudio.com/Index.aspx?DevCenter=Robot_Web_Services&OpenDocument&Url=html/index.html

[28] M. Rouse. (2014, 2016.03.04). *REST (representational state transfer)*. Tilgjengelig: http://searchsoa.techtarget.com/definition/REST

[29] C. Severance, "Discovering JavaScript object notation," *Computer,* ss. 6-8, 2012.

[30] N. Nurseitov, M. Paulson, R. Reynolds, og C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study," *Caine,* vol. 2009, ss. 157-162, 2009.

[31] A. Radenski, "Python First: A lab-based digital introduction to computer science," i *ACM SIGCSE Bulletin*, 2006, ss. 197-201.

[32] T. J. McCabe, "A Complexity Measure," *IEEE transaction on software engineering,* vol. SE-2, 1976.

[33] M. E. Khan, "Different forms of software testing techniques for finding errors," *International Journal of Computer Science Issues,* vol. 7, ss. 11-16, 2010.

[34] ABB. (2015, 2016.04.14). *Robot Web Services*. Tilgjengelig: http://developercenter.robotstudio.com/Index.aspx?DevCenter=Robot_Web_Services&OpenDocument&Url=html/index.html

[35] M. Rouached, S. Baccar, og M. Abid, "RESTful sensor web enablement services for wireless sensor networks," i *Services (SERVICES), 2012 IEEE Eighth World Congress on*, 2012, ss. 65-72.

[36] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, og B. Skibbe, "The relationship between test coverage and reliability," i *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, 1994, ss. 186-195.

[37] A. Dudhe og S. Sherekar, "Performance Analysis of SOAP and RESTFul Mobile Web Service in Cloud Environment," *International Journal of Computer Applications,* 2014.

[38] H. Hamad, M. Saad, og R. Abed, "Performance Evaluation of RESTful Web Services for Mobile Devices," *Int. Arab J. e-Technol.,* vol. 1, ss. 72-78, 2010.

[39] F. Belqasmi, J. Singh, og S. Melhem, "SOAP-Based Web Services vs. RESTful Web Services for Multimedia Conferencing Applications: A Case Study," *IEEE Internet Computing,* 2012.

[40] Python. (2016.04.12). *15.3. time- Time access and conversions*. Tilgjengelig: https://docs.python.org/2/library/time.html

[41] H. Cao, N. Gu, K. Ren, og Y. Li, "Performance research and optimization on CPython's interpreter," i *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, 2015, ss. 435-441.

[42] S. Tkachev, D. Aldoshin, og A. Golubev, "Virtual Laboratory on Nonlinear Control," i *ACE*, 2012, ss. 166-171.

Figurer

Figur 1: Bilde av en stasjon bestående av en virtuell robot. 9
Figur 2: IRC5 kontroller. 12
Figur 3: Kontrollpanelet til en virtuell kontroller. 13
Figur 4: Bilde av reell og virtuell robot. 21
Figur 5: Bilde av virtuell robot som beveger seg mellom to mål. 22
Figur 6: Flyten fra frontend PCSDK til en virtuell eller reell robot. 23
Figur 7: Flytskjema av modulenes bruk ved implementasjon av et program i frontend PCSDK. 24
Figur 8: Prosjektstruktur til frontend PCSDK bestående av mappene *com*, *rapid* og *user*. 26
Figur 9: Eksempel på modulkommentar i frontend PCSDK. 27
Figur 10: Eksempel på funksjonskommentar i frontend PCSDK. 27
Figur 11: Assembly-filene som gjør det mulig å bruke PC SDK i IronPython. 29
Figur 12: Mappe med integrasjonstester til frontend PCSDK. 39
Figur 13: Flyten fra frontend RWS til robot. 41
Figur 14: Flytskjema av modulenes bruk ved implementasjon av et program i frontend RWS. 42
Figur 15: Prosjektstruktur til frontend RWS. 43
Figur 16: Strukturen til Robot Web Services. Hentet fra [34]. 44
Figur 17: Mappe med integrasjonstester til frontend RWS. 54
Figur 18: Graf av resultatet etter oppkoblingstest. 58
Figur 19: Graf med responstider ved endring av num. 61
Figur 20: Graf med responstider ved endring av to elementer i en num liste. 62
Figur 21: Graf med responstider ved endring av fire elementer i en num liste. 63

Tabeller

Tabell 1: Liste av modulene i frontend PCSDK og de tilhørende funksjonene. 37
Tabell 2: Liste av modulene i frontend RWS og de tilhørende funksjonene. 52
Tabell 3: Dekningsgraden til integrasjonstester som tester frontend RWS. 55
Tabell 4: Data fra oppkoblingstest. 58
Tabell 5: Resultatet av inaktivitetstestingen. 59
Tabell 6: Resultatet av koblingspålitelighetstest. 60
Tabell 7: Data fra responstidtest ved endring av num. 61
Tabell 8: Data fra responstidtest ved endring av to elementer i en num liste. 62
Tabell 9: Data fra responstidtest ved endring av fire elementer i en num liste. 63
Tabell 10: Resultatet av kjøretider til blomsterprogram. 64
Tabell 11: Analyse av frontend PCSDK. 68
Tabell 12: Analyse av frontend RWS. 68

Kildekode

Kildekode 1: Eksempel på funksjonsnavn i frontend PCSDK. 28
Kildekode 2: Funksjon som kobler til robot-kontroller med et gitt navn. 30
Kildekode 3: Funksjon brukt til å logge på robot-kontroller med forhåndsdefinert bruker. 30
Kildekode 4: Funksjon som sikrer *mastership* på robot-kontroller. 31
Kildekode 5: Funksjon brukt til å hente RAPID-variabel fra robot-kontroller. 32
Kildekode 6: Eksempel på hvordan et robtarget ser ut i RAPID. 33
Kildekode 7: Eksempel på den oppdaterte strengen som representerer et robtarget i RAPID. 33
Kildekode 8: Funksjon som endrer en spesifisert egenskap til et robtarget. 33
Kildekode 9: Funksjon som endrer predefinert hastighet til speeddata. 34
Kildekode 10: Funksjon brukt til å endre en liste av num på kontroller. 35
Kildekode 11: Integrasjonstest til skrivefunksjonen i modulen rapid_num i frontend PCSDK. 40
Kildekode 12: Eksempel på en nettadresse brukt ved oppkobling til robot-kontroller. 45
Kildekode 13: Funksjon brukt til å koble til robot-kontroller i frontend RWS. 46
Kildekode 14: Eksempel på nettadresse brukt til å logge av robot kontroller. 46
Kildekode 15: Eksempel på nettadresse brukt til å hente ut informasjon om en RAPID-variabel. 47
Kildekode 16: Eksempel på en nettadressen som er brukt til å hente verdien til en RAPID-variabel. 48
Kildekode 17: Funksjon brukt til å hente informasjonen til en RAPID-variabel med frontend RWS. 48
Kildekode 18: Eksempel på nettadresse brukt til å hente verdien til et robtarget. 49
Kildekode 19: Eksempel på nettadresse brukt til å oppdatere et robtarget i RAPID. 50
Kildekode 20: Funksjon brukt til å endre en egenskap i et robtarget med frontend RWS. 50
Kildekode 21: Funksjon brukt til å endre en variabel i en liste med gitt indeks gjennom frontend RWS. 51
Kildekode 22: Integrasjonstest til skrivefunksjonen i modulen rapid_num i frontend RWS. 54

A. Vedlegg

1. Kildekode.
2. Løsning fra RobotStudio (Stasjon, RAPID kildekode).
3. Kort trailer til oppgaven.