Gezamenlijke opleiding UHasselt en KU Leuven

2020•2021
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: elektromechanica

Masterthesis

Ontwikkeling van een ROS-driver voor de Human Interface Mate en validatie in een framework voor mens-robotassemblage

PROMOTOR :
Prof. dr. ir. Eric DEMEESTER

BEGELEIDER :
ing. Martijn CRAMER

Brandon Lemmens, Brecht Van de Heijning
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektromechanica

Masterthesis

Ontwikkeling van een ROS-driver voor de Human Interface Mate en validatie in een framework voor mens-robotassemblage

PROMOTOR :

Prof. dr. ir. Eric DEMEESTER

BEGELEIDER :

ing. Martijn CRAMER

Brandon Lemmens, Brecht Van de Heijning

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

KU LEUVEN

Woord vooraf

Als masterstudenten van de gezamenlijke opleiding industriële ingenieurswetenschappen van UHasselt en KU Leuven hebben wij de kans gekregen om onze masterthesis uit te voeren in het onderzoekscentrum ACRO van KU Leuven. Hierbij hebben wij als opdracht gekregen om een ROS-driver te ontwikkelen voor de Human Interface Mate (HIM) van Arkite met als doel deze te kunnen integreren in een framework voor mens-robotsamenwerking. De integratie van deze ROS-driver is vervolgens uitgewerkt aan de hand van een demonstrator. In deze demonstratie assembleert een operator in samenwerking met een robotarm een zesdelig product met een assemblagevolgorde naar keuze.

Er werd gekozen voor dit onderwerp aangezien we beiden een grote interesse hebben in programmeren en robotica. Deze master thesis heeft ons de kans gegeven om ons te verdiepen in deze onderwerpen en deze kennis om te zetten in een werkend product.

Graag danken wij allen die ons geholpen hebben bij het verwezenlijken van onze master thesis. In de eerste plaats Ing. Martijn Cramer, onze interne begeleider die altijd paraat stond om ons te begeleiden in ongekende situaties en te helpen bij onvoorziene omstandigheden. Daarnaast ook onze interne promotor Prof. Dr. Ir. Eric Demeester, die ons werk met een kritische blik heeft beoordeeld en ons iedere keer de juiste richting heeft ingestuurd. Ook danken wij Jasper Dirix, projectingenieur bij Arkite, voor de volledige werking van de HIM uit te leggen en altijd klaar te staan bij problemen betreffende de HIM. Tot slot danken wij onze familie en vriendin voor de grote steun gedurende deze masterproef.

Inhoudsopgave

Woord vooraf ... *1*

Lijst van tabellen ... *5*

Lijst van figuren ... *7*

Abstract ... *9*

Abstract in English ... *11*

Hoofdstuk 1: Inleiding ... *13*

1.1 Situering ... **13**

1.2 Probleemstelling en onderzoeksvraag ... **14**

1.3 Doelstellingen ... **15**

1.4 Materiaal en methode ... **16**

Hoofdstuk 2: Literatuurstudie ... *17*

2.1 Introductie ... **17**

2.2 Human Interface Mate (HIM) ... **17**

2.2.1 Werkschema's HIM ... 19

2.3 Robot Operating System (ROS) ... **20**

2.3.1 Waarom ROS? ... 20
2.3.2 ROS-structuur ... 21
2.3.3 Voorbeeld ROS-project ... 22

2.4 ROS-communicatie met robot controller ... **25**

2.4.1 Robot Description Format (RDF) ... 25
2.4.2 ROS Visualiser (RViz) ... 27
2.4.3 MoveIt! ... 27

2.5 ROS-communicatie via web-API ... **28**

2.5.1 Wat is een web-API ... 28
2.5.2 Eigenschappen van de HIM web-API ... 29
2.5.3 Softwarematige communicatie met de web-API ... 32

Hoofdstuk 3: Ontwikkeling van een ROS-driver voor de HIM *33*

3.1 Beschikbare documentatie van de HIM **33**

3.2 Opbouw van de ROS-driver **34**

3.3 Communicatie met de HIM via ROS **35**

3.3.1 Get-Requests 35

3.3.2 Patch-, Post-, Put- en Delete-Requests 39

3.3.3 Foutmeldingen 40

3.4 Conclusie **41**

Hoofdstuk 4: Opzetten van een framework voor mens-robot-samenwerking *43*

4.1 Detecties via de Human Interface Mate (HIM) **43**

4.2 Aansturing van de UR5-robot **45**

4.3 Robotiq 2F-85 grijper **46**

4.4 Action server en client **47**

4.5 Conclusie **48**

Hoofdstuk 5: Demonstratie van de ROS-driver voor de HIM in mens-robotsamenwerking *49*

5.1 Opbouw van de demonstrator **49**

5.2 Opzetten van een demonstrator **53**

5.2.1 Assemblagetoestandsgraaf van Bourjault's balpen 53

5.2.2 Actiedetectie met behulp van de HIM 55

5.2.3 Samenwerking met de UR5-cobot 55

5.3 Conclusie **56**

Hoofdstuk 6: Conclusie *57*

6.1 Behalen van de vooropgestelde doelstellingen **57**

6.2 Mogelijkheden voor toekomstig onderzoek **59**

Referentielijst *61*

Lijst van tabellen

Tabel 1: Functionaliteiten van de web-API van de HIM 29

Tabel 2: Betekenis parameters 'FollowJointTrajectoryGoal' 45

Tabel 3: Betekenis parameters Robotiq grijper 46

Tabel 4: Structuur action messages 47

Tabel 5: Assemblagetoestanden 54

Lijst van figuren

Figuur 1: (a) de Human Interface Mate (b) HIM projectievoorbeeld ... 13

Figuur 2: De complexe (links) en eenvoudige (rechts) versie van de balpen van Bourjault... 15

Figuur 3: Situering ROS HIM-driver (1) in het opgestelde framework voor mens-robotsamenwerking (2) ... 16

Figuur 4: HIM-detectie zonder (links) en met (rechts) het gereedschap ... 18

Figuur 5: Gebruiksvoorbeeld van de HIM ... 18

Figuur 6: Voorbeeld Composite step... 19

Figuur 7: ROS Communicatie-diagram ... 21

Figuur 8: ROS-service schema ... 22

Figuur 9: Voorbeeldtoepassing mobiele robot ... 22

Figuur 10: Schematische voorstelling voorbeeldtoepassing ... 23

Figuur 11: Verschillende links en joints in een robotarm ... 25

Figuur 12: XML-code van een typische robot ... 26

Figuur 13: (a) Weergave in RViz, (b) Weergave in MoveIt!... 27

Figuur 14: Schema web-API en web Services ... 28

Figuur 15: Voorbeeld van een eenvoudig GET-request ... 32

Figuur 16: Voorbeeld request-URL... 34

Figuur 17: Voorbeeld request body... 34

Figuur 18: Volledige code GetProject-request ... 36

Figuur 19: Uitvoeren van het request en verwerken van de opgehaalde data ... 37

Figuur 20: JSON-structuur uitlezen ... 38

Figuur 21: Opbouw client-nodes ... 40

Figuur 22: Detectie-algoritme 44

Figuur 23: Opstelling demonstrator 49

Figuur 24: Onderdelen van Bourjault's balpen 50

Figuur 25: Opslag onderdelen 51

Figuur 26: (a) Houders tussenposities assemblages (b) Doorsnede houders tussenposities assemblages 52

Figuur 27: Assemblagetoestandsgraaf Bourjault's balpen 53

Abstract

De onderzoeksgroep ACRO van KU Leuven focust op automatisatie, computervisie en robotica. In het kader van lopend onderzoek naar intentiegebaseerde mens-robotsamenwerking voor assemblage, wensen de onderzoekers gebruik te maken van de Human Interface Mate (HIM) ontwikkeld door Arkite voor het registreren van operatoracties. De HIM is momenteel enkel te configureren en uit te lezen via de bijbehorende softwareapplicatie. Het doel van deze masterproef is het ontwikkelen van een ROS-driver voor de HIM, het valideren ervan in een framework voor mens-robotsamenwerking en op basis hiervan een demonstrator uitwerken.

Het ontwikkelen van deze ROS-driver bestaat uit twee onderdelen: enerzijds het uitwisselen van informatie met de HIM en anderzijds het communiceren van deze informatie naar ROS. Het uitwisselen van informatie geschiedt via de web-API van de HIM op basis van het Hypertext Transfer Protocol (HTTP). Het framework maakt gebruik van de ROS-driver om de gedetecteerde assemblagehandelingen uit te lezen, waarna de robotactieplanner deze informatie gebruikt om de UR-robot en Robotiq-grijper aan te sturen.

Gebruikmakende van dit framework is een demonstrator ontwikkeld waarbij een mens zonder voorgaande kennis in samenwerking met een robot een zesdelig product kan assembleren via een assemblagevolgorde naar keuze. Hierbij past de robot zich continu aan naar de keuzes van deze persoon. Uit deze experimentele validatie blijkt dat het framework samen met de ROS-driver een goede basis vormt voor een robuuste mens-robotsamenwerking.

Abstract in English

The research group ACRO of KU Leuven focuses on automation, computer vision and robotics. As part of ongoing research into intention-based human-robot collaboration in assembly, they intend to make use of the Human Interface Mate (HIM) developed by Arkite to recognize and register operator actions. The HIM is currently only configurable and readable through its software application. The goal of this master thesis is to develop a ROS driver for the HIM, validate it in a framework for human-robot collaboration and develop a relevant demonstrator based on this.

The development of the ROS driver for the HIM consists of two parts: exchanging information with the HIM and communicating this information to ROS. The exchange of information takes place via the HIM's web API based on Hypertext Transfer Protocol (HTTP). The framework uses the ROS driver to read the assembly operations, after which the robot action planner uses this information to control the UR robot and Robotiq gripper.

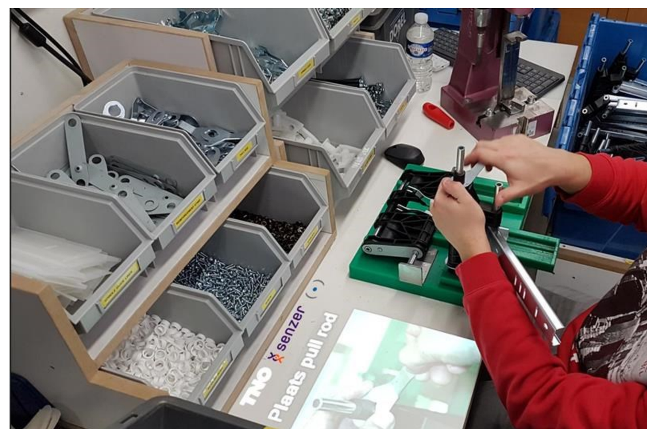
Using this framework, a demonstrator has been developed in which a human with no previous knowledge can assemble a six-part product in an assembly sequence of choice together with a robot. In doing so, the robot continuously adapts itself to the choices of this operator. This experimental validation shows that the framework, together with the ROS driver, forms a good basis for a robust human-robot collaboration.

Hoofdstuk 1: Inleiding

1.1 Situering

Deze masterproef is voorgedragen door de onderzoeksgroep ACRO van KU Leuven te campus Diepenbeek in samenwerking met het technologiebedrijf Arkite NV. ACRO focust op automatisatie, computervisie en robotica en doet onderzoek binnen een breed toepassingsgebied. De onderzochte technologieën variëren van fabricage, assemblage, logistiek, landbouw, gezondheidszorg, bouwnijverheid en thuisgebruik tot end-of-life behandelingen. Het Limburgse technologiebedrijf Arkite heeft een systeem ontworpen dat productieoperatoren helpt om hun assemblagetaken foutloos uit te voeren. Dit systeem heet de 'Human Interface Mate' (HIM) (Zie Figuur 1(a)) en gebruikt 3D-sensortechnologie om de positie van de assemblageonderdelen en de status van de huidige taak bij te houden. Via projectoren kan de HIM de operator informeren, begeleiden en onderwijzen tijdens het assemblageproces door middel van augmented-reality-instructies (Zie Figuur 1(b)). Ook is de HIM in staat het werkproces te standaardiseren onder alle werknemers en foute handelingen te detecteren. Het doel van de HIM is de werkefficiëntie te verhogen en de kwaliteitskosten te reduceren.

(a)



(b)

Figuur 1: (a) de Human Interface Mate (b) HIM projectievoorbeeld [1]

1.2 Probleemstelling en onderzoeksvraag

Tot op heden heeft Arkite zich gefocust op de supervisie en ondersteuning van menselijke operatoren. Het bedrijf beoogt echter ook deze HIM-technologie uit te breiden naar het domein van mens-robotsamenwerking. Voordien was het namelijk zo dat robots in industriële omgevingen volledig afgesloten waren. Het was niet veilig om mensen in de buurt te laten komen van deze robots zonder afscherming. Deze robots konden namelijk niet detecteren wanneer ze contact maken met een mens. De laatste jaren is echter veel vooruitgang gekomen in deze sector en staat een nieuw tijdperk te beginnen. Hierbij staat mens-robotinteractie centraal en ligt de focus op het samenwerken van mens en cobot (EN: collaborative robot).

De HIM communiceert met menselijke operatoren onder de vorm van visuele assemblage-instructies. Om de HIM te integreren in mens-robotsamenwerking is het vereist communicatie op te zetten tussen de HIM en de assisterende cobot. De HIM dient namelijk een signaal te sturen naar de cobot wanneer de operator een specifieke handeling heeft ondernomen waarbij ondersteuning van de cobot vereist is. Dit signaal moet de juiste informatie en parameters bevatten om de cobot aan te sturen. Het mogelijk maken van de informatiestroom in het open source Robot Operating System (ROS) tussen de HIM en een robotarm zoals de UR5-cobot vormt de focus van deze masterproef.

De uitdaging van de Human Interface Mate is dat deze momenteel als een afgesloten systeem werkt, en informatie delen met de andere softwareapplicaties nog een struikelblok is. Momenteel kan communicatie met de buitenwereld wel via de interface van de web-API, maar deze is in vele opzichten en voor vele toepassingen te beperkt. Een web-API (Application Programming Interface) is een interface met een aantal ingebouwde functies die het voor de softwareontwikkelaar mogelijk maakt om toegang te krijgen tot bepaalde data of applicaties van de software.

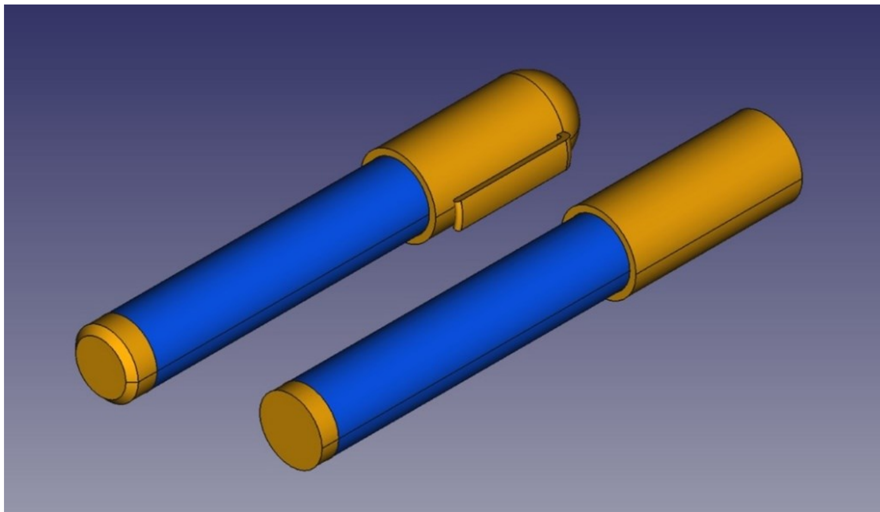
Pas wanneer de communicatie met de buitenwereld aangepast wordt en een samenwerking met externe robotica mogelijk wordt, kan er een efficiënte coöperatie tussen mens en robot tot stand komen. Het voordeel van zo'n samenwerking is dat men het beste van beide werelden kan gebruiken. Zo kan de persoon zich focussen op taken die grote behendigheid vereisen of waarbij het gebruik van een speciaal gereedschap of systeem nodig is, terwijl de robot fysiek zware of repetitieve taken voor zijn rekening neemt. Dit betekent echter wel dat extra veiligheidsmaatregelen en eventueel bijkomende opleidingen nodig zijn om erop toe te zien dat deze samenwerking veilig en vlot verloopt [2].

1.3 Doelstellingen

Het hoofddoel van de masterproef is de extensie naar en de integratie van de HIM-technologie in het domein van mens-robotsamenwerking. Hiervoor is een mogelijkheid tot communicatie nodig tussen de HIM en de robotica, die nu nog niet bestaat.

Deze hoofddoelstelling is op te splitsen in meerdere deeldoelstellingen. Ten eerste dient er een communicatie tussen de HIM en robotica opgezet te worden. Deze communicatie moet bidirectioneel zijn en in staat zijn om alle mogelijke parameters en taken van de HIM door te sturen en aan te passen. Daarna kan deze communicatie gebruikt worden in een framework voor mens-robotsamenwerking. Hierbij leest de communicatiedriver de huidige situatie van de assemblage in en stuurt deze op basis van de informatie de robotica aan. Vervolgens kan dit framework gebruikt worden om een assemblage te verwezenlijken. In deze masterproef zal de zesdelige assemblage, getoond in Figuur 2, nader worden bestudeerd. Deze balpen werd ontworpen door de Franse onderzoeker Bourjault in zijn studie naar assemblagevolgorde generatie en is uitgegroeid tot een frequent gebruikte use-case in de academische wereld [3].

Een maatstaaf voor succes is wanneer iemand zonder diepgaande kennis van de HIM en/of robotica deze balpen kan assembleren via een assemblagevolgorde naar keuze in samenwerking met een robot en waarbij deze robot zich continu aanpast aan de assemblageintentie van deze persoon.



Figuur 2: De complexe (links) en eenvoudige (rechts) versie van de balpen van Bourjault

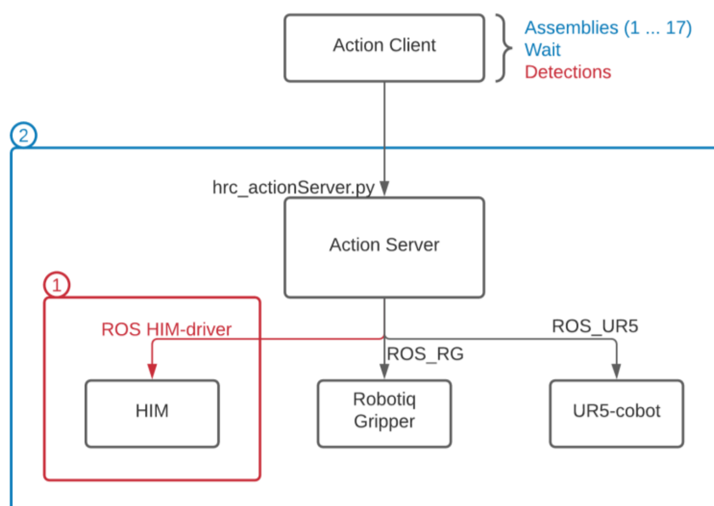
1.4 Materiaal en methode

Allereerst gaat er bij het ontwikkelen van communicatie met de HIM gebruikgemaakt worden van de web-API. Hiermee is het mogelijk om alle belangrijke parameters en projectgegevens in te stellen en op te vragen. Verder kan deze ook acties op projectniveau uitvoeren zoals het starten en stoppen van stappen in een proces en het inladen van bepaalde projecten.

Om deze functionaliteit bruikbaar te maken in mens-robotsamenwerking kan de communicatiedriver geschreven worden in ROS of URCaps. Beide zijn mogelijke programmeeromgevingen gefocust op robotica maar er is toch gekozen voor ROS. Dit omdat URCaps enkel communicatie kan maken met UR-robotica, wat ervoor zou zorgen dat deze communicatiedriver enkel bruikbaar zou zijn voor deze specifieke fabrikant. Ook is URCaps geen opensourcesoftware waardoor niet iedereen deze zomaar kan gebruiken. ROS daarentegen is wel open source en wordt wereldwijd veelvuldig gebruikt.

Als toepassing van de communicatiedriver wordt deze geïntegreerd in een framewerk voor mens-robotsamenwerking zoals zichtbaar is onder deel 1 van Figuur 3. Het assembleren van de eerder besproken balpen vereist communicatie tussen deze driver, de robotica en een beslissend orgaan. Dit orgaan kent alle assemblagemogelijkheden, heeft door middel van de communicatiedriver besef van de huidige situatie en geeft hiermee de juiste aansturingen aan de robotica.

Het beslissend orgaan is geschreven in ROS en bestaat uit de combinatie van een *action server* en een *action client*. Naast het uitlezen van de huidige situatie via de HIM bezit de server ook de mogelijkheid om de robotica aan te sturen (zie deel 2 van Figuur 3). De client neemt alle beslissingen door berichten te sturen naar deze action server. In deze berichten kan de client vragen naar informatie of aansturingen doorgeven aan de robotica.



Figuur 3: Situering ROS HIM-driver (1) in het opgestelde framework voor mens-robotsamenwerking (2)

Hoofdstuk 2: Literatuurstudie

2.1 Introductie

Om te beginnen met het ontwerpen van een driver voor het integreren van de HIM in het domein van mens-robotsamenwerking is het nodig om literatuur te bestuderen over dit onderwerp. Deze driver is geschreven in het opensource framework ROS (Robot Operating System). Een framework is een verzameling van softwarecomponenten en afgesproken code-standaarden. Hierdoor is er veel informatie over te vinden in artikels en online forums.

Dit hoofdstuk legt uit hoe de gebruikte software in elkaar zit. Allereerst wordt het algemene framework volledig besproken. Dit is de software/middleware waar de driver op gebaseerd is. Dit framework is ook bepalend voor de structuur en eisen waaraan de driver moet voldoen. Daarna beschrijft dit hoofdstuk alle soorten communicatie die nodig zijn om met de robot controller te communiceren.

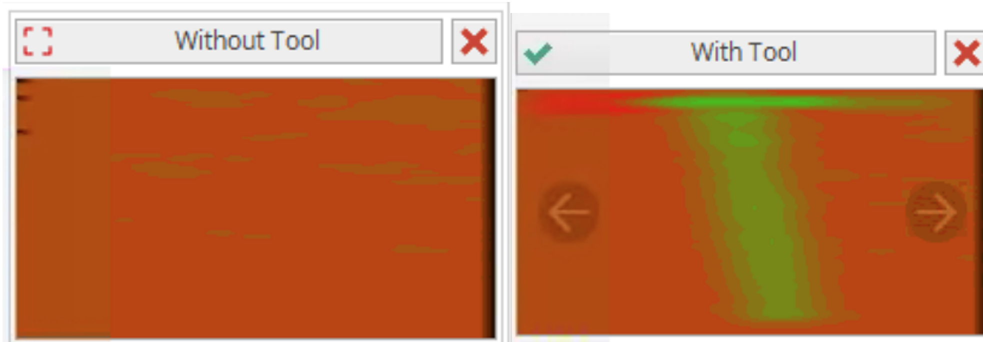
2.2 Human Interface Mate (HIM)

De Human Interface Mate (HIM) transformeert werkstations in een digitale en interactieve omgeving door projecties te combineren met een reeks van sensoren en beeldverwerking. De gebruikte sensoren zijn vergelijkbaar met de Kinect-dieptecamera van Microsoft. Hierbij maakt de sensor gebruik van het Time of Flight-principe, waarbij de diepte van het beeld berekend wordt door de tijd te meten die het uitgezonden licht nodig heeft om de afstand tussen de camera en het object heen en weer af te leggen. Daarnaast bezit de HIM ook een infrarood- en RGB-sensor, deze worden respectievelijk gebruikt voor het uitvoeren van detecties en om een beeld van de werkomgeving vast te leggen (bv. voor het loggen van (foutieve) handelingen) [4].

De HIM is hiermee in staat een groot aantal bewegingen en handelingen van de operator te detecteren, waardoor de HIM op het juiste moment in de assemblage de nodige informatie kan leveren met behulp van een projector (denk aan het tonen van de relevante handleiding of het oplichten van een bepaald gereedschap). Om de detecties uit te voeren, dient men deze eerst aan te leren aan de HIM via het bijhorende softwareprogramma. Hiervoor is het nodig om een sensor te kiezen om de detecties mee aan te maken. Er is namelijk de keuze tussen: de dieptesensor, de infraroodcamera en kleurencamera. Daarnaast is er nog de optie “automatisch”, deze test alle opties en neemt hieruit de beste optie. Het is mogelijk om een beeld te maken van de werkplaats op twee

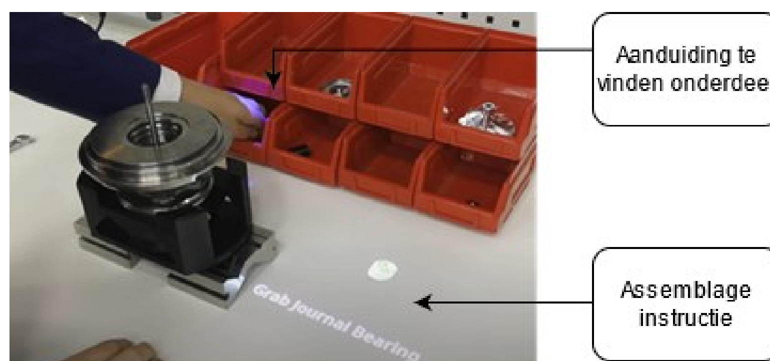
verschillende tijdstippen en hierbij te focussen op één bepaald gebied. Alle veranderingen in dit gebied zullen opgeslagen en verwerkt worden tot één detectie (zie Figuur 4). Dit laat toe om te detecteren wanneer een object aanwezig is of niet. Daarnaast heeft de HIM ook de mogelijkheid om te detecteren of het gedefinieerde object geoccludeerd wordt. Indien dit het geval is, zal de HIM niet kunnen zien of het object nog aanwezig is of niet, aangezien dit object niet meer in zicht is. In deze situatie onthoudt de HIM de laatste status van dit object.

Enkele voorbeelden van detecteerbare handelingen zijn het activeren van een digitale knop, het monitoren van containers met bepaalde materialen, het grijpen en terugleggen van gereedschap en het opvolgen van assemblageactiviteiten. Daarnaast kan het systeem ook gebruikt worden voor het opsporen en oplossen van menselijke fouten [5].



Figuur 4: HIM-detectie zonder (links) en met (rechts) het gereedschap

Het opzetten van de informatie en het aansturen van de hardware gebeurt via de bijhorende HIM-software. Hiermee is het mogelijk verschillende processen aan te maken die bestaan uit meerdere stappen, taken, variabelen en detecties. Deze processen vormen in werkelijkheid de verschillende assemblages. Een voorbeeld van het gebruik van de HIM in een assemblage wordt getoond in Figuur 5, waarbij de operator tegelijk een instructie en het correcte onderdeel te zien krijgt.

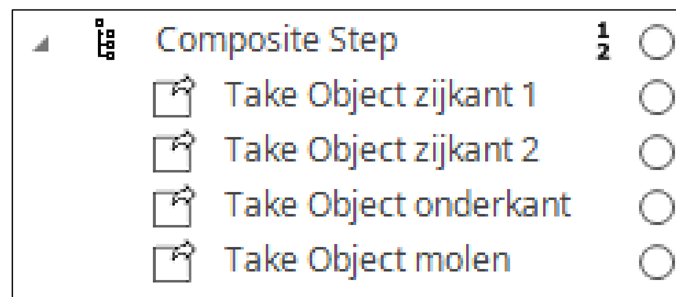


Figuur 5: Gebruiksvoorbeeld van de HIM [6]

2.2.1 Werkschema's HIM

De huidige versie van de HIM kan een werkschema op verschillende manieren voorstellen. Deze bezit namelijk functies die het mogelijk maken om een assemblage flexibel uit te voeren. De HIM-software kan variabelen een waarde geven en deze waardes opvragen en vergelijken, waardoor er een IF-functionaliteit gecreëerd is met bijbehorende AND- en OR-functies. Zo kan een variabele bijvoorbeeld een detectie zijn, waarbij de HIM een bepaalde handeling of instructie weergeeft bij aan- of afwezigheid van desbetreffende detectie.

Ook kan de HIM-software gebruik maken van 'Composite steps' zoals zichtbaar is op Figuur 6. Dit is een soort stap die meerdere onderliggende stappen voorstelt. Verder kan deze software ook gebruik maken van loops die verschillende onderdelen van de assemblage meerdere keren herhalen tot aan een bepaalde voorwaarde voldaan is.



Figuur 6: Voorbeeld Composite step

2.3 Robot Operating System (ROS)

Voor ROS bestond ontwierpen veel mensen code voor hun robots. Dit werd enkel gedaan zonder een algemeen framework. Ze ontwierpen daarom ook enkel code die van toepassing was voor hun specifieke robot. Er waren geen algemene regels of standaarden waardoor iedereen opnieuw moest beginnen bij de start van iedere robot.

ROS kan beschreven worden in vijf punten. Allereerst maakt ROS het mogelijk om modulaire software te ontwikkelen doordat het een grote verzameling heeft aan gratis modulaire code. Verder heeft het een 'Run-Time Environment' die communicatie mogelijk maakt tussen systeemelementen en data. Het bezit daarnaast geen real-time communicatie en is zelf geen standaard voor programmeren. ROS zorgt wel voor conventies voor het ontwerpen van reproduceerbare en betrouwbare code. Deze conventies bevatten verschillende onderdelen zoals naamgeving, communicatie, parametrisatie en het opstellen van de robotmodellen [7]. Ook bezit het verschillende ontwikkelingstools die het mogelijk maken om het proces te controleren en visualiseren. Als laatste heeft ROS een grote gemeenschap doordat het open source is. Dit zorgt ervoor dat er altijd de mogelijkheid is om vragen te stellen op online forums [8].

2.3.1 Waarom ROS?

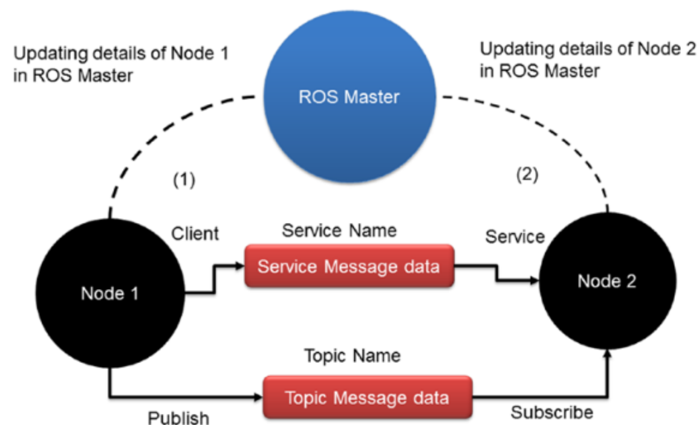
Naast ROS bestaan er nog andere frameworks. Enkele voorbeelden hiervan zijn Urbi, MIRO, YARP en OpenRDK. Er is gekozen voor ROS omdat dit verschillende voordelen heeft ten opzichte van de alternatieven. Allereerst heeft ROS nodes die onderling verbonden zijn via een *distributed message* system. Een node is een zelfstandig programma dat in ROS een bepaalde functionaliteit bezit. Deze kunnen communiceren met andere nodes en aan de hand van hun specifieke functionaliteit bepaalde berichten doorsturen naar elkaar. Een voorbeeld hiervan volgt in paragraaf 2.3.3. Door middel van het distributed message system kan men enkel de nodige nodes aanspreken. Het zorgt er ook voor dat het hele systeem niet faalt zodra er één fout is in het systeem.

Verder is ROS ook ontworpen voor het Linux-besturingssystem. Men heeft echter wel de optie om ROS te installeren op macOS of Windows. Deze alternatieven zijn mogelijk maar zijn niet geadviseerd omdat ze ROS op een onderliggend besturingssysteem uitvoeren via een *virtual desktop*, wat kan leiden tot instabiliteiten. Daarnaast is ROS niet gelimiteerd tot één programmeertaal en is het mogelijk alle nodes te schrijven in C++, Python, Java en LISP.

Zoals eerder vermeld is ROS ook open source. Dit is één van de grote voordelen van ROS en zorgt ervoor dat iedereen er gratis mee kan werken en vermijdt gepatenteerde code.
Als laatste is het ook mogelijk om andere frameworks te integreren in ROS zoals bijvoorbeeld Urbi. Hierdoor is het mogelijk voor ROS-gebruikers om functionaliteiten van andere frameworks te gebruiken in hun ROS-omgeving [9].

2.3.2 ROS-structuur

ROS-communicatie is gebaseerd op peer-to-peer communicatie, die ervoor zorgt dat de functionaliteit binnen ROS opgesplitst is. Ieder onderdeel van een ROS-programma is onderverdeeld in verschillende componenten die met elkaar kunnen communiceren zoals zichtbaar is in Figuur 7. De componenten geven door deze communicatie de functionaliteit aan het ROS-programma en heten knooppunten (*nodes*). Een ROS-programma bestaat vaak uit verschillende nodes.



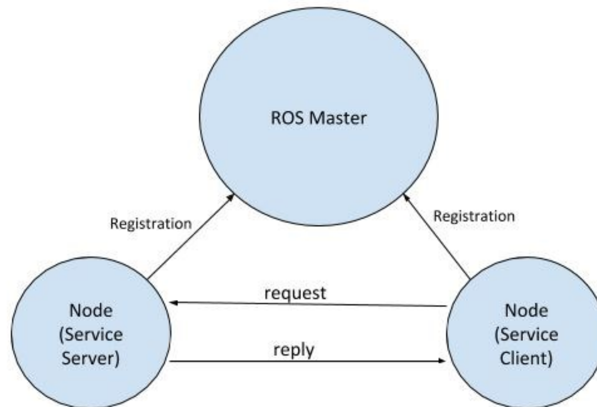
Figuur 7: ROS Communicatie-diagram [9, p. 151]

Naast ROS-nodes bestaan er ook ROS-topics. Een topic is een soort van communicatielijn waarop ROS-nodes hun berichten publiceren (EN: *publishen*). Ook kan een node de informatie lezen van een topic door te abonneren (EN: *subscriben*) op dat topic. Daarnaast is het ook mogelijk voor een node om tegelijk te publishen en te subscriben op verschillende topics. Ook kunnen meerdere nodes op dezelfde topics subscriben en publishen.

Verder gebruiken de nodes ROS-berichten (EN: *messages*) om met de verschillende topics te communiceren. Deze messages zijn gebaseerd op datatypes zoals: int, bool, float en string. Ook kan de gebruiker een eigen berichtstructuur, gebaseerd op de basisdatatypes, opstellen. ROS-messages definiëren de structuur waaraan de berichten van een node moeten voldoen om gepubliceerd te kunnen worden op een bepaald topic, om zo incompatibele dataoverdracht te vermijden.

Om alles met elkaar te laten werken is een ROS-master nodig. Deze bezit alle informatie van de subscribers en publishers. De master is de coördinator tussen alle nodes en topics en zorgt ervoor dat alle messages op de juiste plaatsen aankomen.

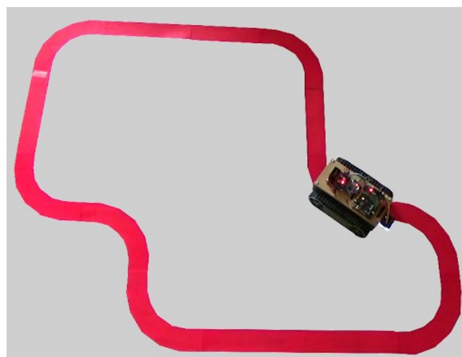
Naast subscribers en publishers bestaat er nog een ander communicatiemodel voor ROS. Dit model is gebaseerd op services. Een service bevat een verzoek (EN: *request*) en een antwoord (EN: *reply*) zoals zichtbaar is in Figuur 8. In tegenstelling tot ROS-topics, stuurt het enkel informatie door naar een node wanneer deze node een request stuurt. De node die een request aanvraagt is een client node en de node die deze service uitvoert is de server node [9].



Figuur 8: ROS-service schema [10]

2.3.3 Voorbeeld ROS-project

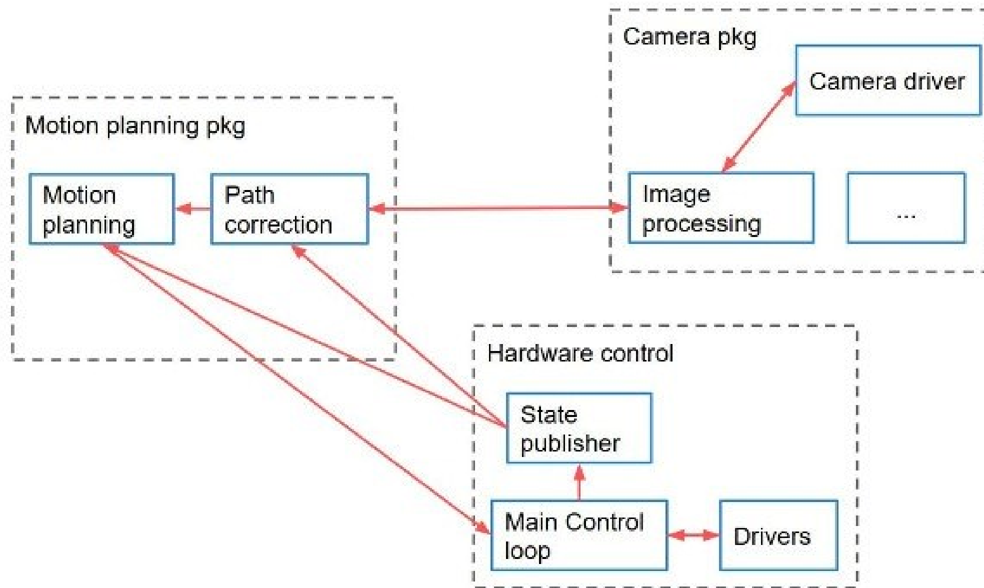
Om de eerder beschreven structuren en onderdelen van ROS nader toe te lichten, is een eenvoudig voorbeeld van een ROS-programma hier besproken. Dit voorbeeld is namelijk een mobiele robot die autonoom bestuurd wordt via een camera. Met deze opstelling is het bijvoorbeeld mogelijk om de robot een eerder uitgelijnd traject af te laten leggen door dit traject te volgen met de camera (zie Figuur 9). Er is gekozen voor dit voorbeeld omdat hier de belangrijkste onderdelen van ROS ter sprake komen zonder overbodige details.



Figuur 9: Voorbeeldtoepassing mobiele robot [11]

Allereerst is het belangrijk om het project op te delen in ROS-packages die de functionaliteit van het programma bevatten zoals zichtbaar is op Figuur 10. Dit voorbeeld bevat drie packages voor: het

bestuderen van de hardware, het plannen van het robotpad en het uitlezen van de camera. De hardware controle is verantwoordelijk voor het besturen van de wielen en andere actuatoren die eventueel aanwezig zijn voor het bewegen van de robot. Daarnaast zorgt de padplanningsmodule voor het controleren van het huidige pad en de planning van het verdere pad van de robot. Als laatste is er het camera-package, dat het mogelijk maakt om beelden te verwerken en hieruit nuttige informatie te halen.



Figuur 10: Schematische voorstelling voorbeeldtoepassing [12]

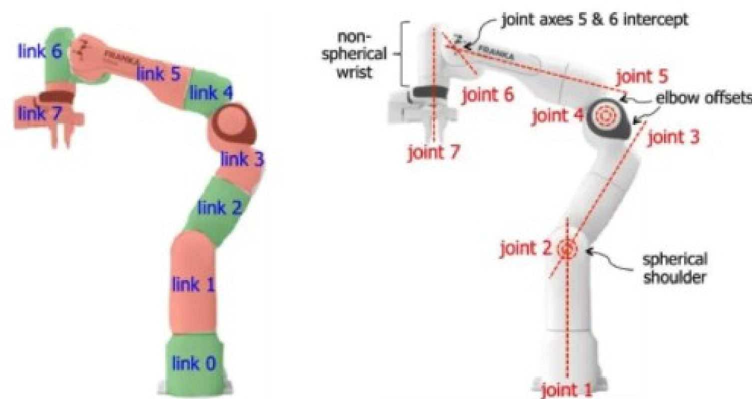
Om de functionaliteit te geven aan de eerder besproken ROS-packages is het vereist om de nodige ROS-nodes aan te maken. Voor de package van de camera is één van de mogelijke oplossingen het onderverdelen van de code in twee verschillende nodes. De eerste node bevat de driver voor de gebruikte camera en zorgt ervoor dat een beeld opgebouwd kan worden, terwijl de tweede node alle code bevat voor de beeldverwerking. Hierbij kan deze node bijvoorbeeld de lijn herkennen op het beeld en de positie hiervan berekenen. Om deze beeldverwerking echter mogelijk te maken is communicatie tussen deze twee nodes vereist. Voor de communicatie is het mogelijk om te kiezen tussen topics, services of actions.

Vervolgens is de opdeling in nodes opnieuw van toepassing voor de padplanning. Hierbij is het bijvoorbeeld een optie om deze op te splitsen in één node die zorgt voor de bewegingsplanning en een andere node die deze planning aanpast aan externe factoren zoals obstakels. Voor het berekenen van deze padplanning is het echter noodzakelijk om communicatie te hebben met de eerder ontwikkelde node voor de beeldverwerking. Dit is, in tegenstelling tot de communicatie tussen de driver voor de camera en de beeldverwerking, een communicatie tussen nodes van een verschillende ROS-package. Ondanks dit verschil is er geen onderscheid in de mogelijke communicatieprotocollen en kan het opnieuw via topics, services en actions.

Als laatste is er de hardware controle waarbij gekozen is voor drie nodes. De eerste is net zoals bij de camera de driver voor de gebruikte motoren. Daarnaast is er de hoofdregelkring die op basis van inkomende informatie de motoren correct aanstuurt. De laatste node zorgt in dit voorbeeld voor het uitschrijven van de status van de motoren. In deze package is er opnieuw communicatie vereist tussen deze drie nodes. De hoofdregelkring communiceert enerzijds bidirectioneel met de driver om de motoren aan te sturen en anderzijds met de status node over de situatie van de motoren. De informatie waarop deze hoofdregelkring zijn regeling baseert is afkomstig uit de padplanning. Deze bepaald namelijk de richting waarin de robot moet bewegen en geeft dit vervolgens door aan de hoofdregelkring. Ook hebben beide nodes van de padplanning informatie nodig van de motoren. Daarom stuurt de status node ook informatie naar de padplanner en de obstakelvermijding [12].

2.4 ROS-communicatie met robot controller

Het model van een robot kan gezien worden als een kinematische ketting van starre schakels (*links*) verbonden door gewrichten (*joints*) zoals zichtbaar is op Figuur 11. Kinematisch, omdat de beweging van de robot bestudeerd wordt zonder de krachten die hiervoor nodig zijn te beschouwen. De vraag is nu hoe deze schakels en gewrichten programmatisch geïmplementeerd kunnen worden. Daarnaast kunnen de fysische eigenschappen, zoals bijvoorbeeld het gewicht en de traagheid van de schakels en het type van gewricht (snelheidslimieten, etc.) in acht genomen worden.



Figuur 11: Verschillende links en joints in een robotarm [13]

Dit gebeurt met behulp van XML-bestanden. XML (extensible markup language) bestanden gebruiken zogenaamde elementen en attributen om objecten en gegevens te structureren. De verschillende gedefinieerde elementen en attributen hangen af van het type XML-bestand. Binnen robotica zijn er verschillende types XML-bestanden die gebruikt worden. De belangrijkste zijn de URDF (Universal Robot Description Format) en SRDF (Semantic Robot Description Format) [14].

2.4.1 Robot Description Format (RDF)

URDF (Universal Robot Description Format) en SRDF (Semantic Robot Description Format) zijn beide tools/packages binnen ROS, geschreven in XML-code, die de verschillende elementen van een robot beschrijven.

De URDF bevat de kinematica en een basis aan fysiche eigenschappen voor de schakels en gewrichten. De belangrijkste tags binnen de URDF zijn *link* en *joint*. In Figuur 12 is een voorbeeld te zien van XML-code in URDF-formaat van een typische robot.

Elk link-element krijgt een naam en drie sub-elementen: visual, collsion en inertial. Visual bevat de visuele beschrijving van de schakels en gebruikt hiervoor primitieve elementen (kubus, bol, cilinder, etc.). De oorsprong van deze elementen wordt relatief ten opzichte van een referentie (meestal de basis waarop de robot staat) geplaatst. Verder kan ook het materiaal en de kleur gekozen worden. Eén link kan meerdere visual-elementen bevatten. Het tweede sub-element, collision, is gelijkaardig aan de visuele beschrijving, maar vereist een lage resolutie voor de mesh van de schakel. Dit element wordt namelijk gebruikt om botsing met andere schakels te detecteren, waarbij een te hoge resolutie te veel berekeningskracht vereist. Tot slot beschrijft inertial de massa van de schakel zodat het massacentrum, massaverdeling en totale massa gekend zijn.

Het joint-element verbindt twee link-elementen; een parent-link en een child-link. De oorsprong van de gewrichten wordt altijd relatief aan de parent-link gedefinieerd. Elk gewricht krijgt een naam en type (revolute, prismatic, fixed of floating). Daarnaast bevat het element ook enkele sub-elementen die de fysische limieten en dynamische eigenschappen bevatten.

Figuur 12: XML-code van een typische robot [15]

De SRDF bevat de informatie die niet omvat zit in de URDF, maar nuttig kan zijn voor een reeks van toepassingen. De bedoeling is hier om semantische informatie toe te voegen over de robotstructuur. Groepen van links kunnen gevormd worden aan de hand van aanliggende joints, langere ketens of eerder gevormde groepen. Verder bevat het ook informatie over hoe de robot beweegt in zijn omgeving en zijn connectie tot de basis van de robot. Een SRDF-bestand kan een URDF-bestand niet vervangen en is geen uitbreiding van dit bestand, maar een URDF moet wel bestaan om de links en joints, waar binnen het SRDF-bestand naar verwezen wordt, te definiëren.

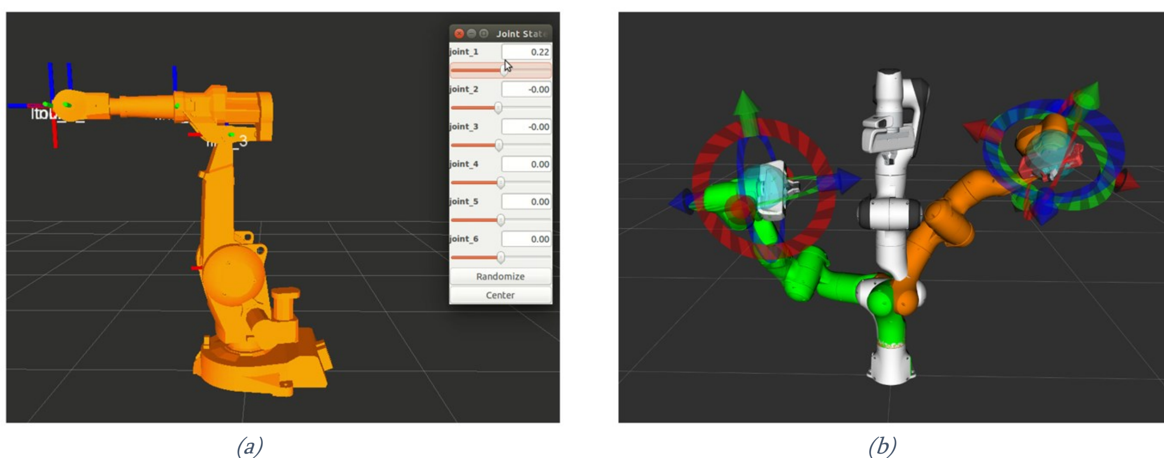
2.4.2 ROS Visualiser (RViz)

RViz is de 3D-visualisatietool van ROS. Deze tool kan (onder andere) de robot met behulp van een URDF en SRDF-bestand in een virtuele omgeving plaatsen. Zo kan men veel informatie over de robot visualiseren en er plugins op toepassen, maar ook de interactie met en positie ten opzichte van andere objecten registreren en aansturen. De aansturing binnen RViz is echter beperkt, maar deze kan uitgebreid worden door het gebruik van *MoveIt*!. Op Figuur 13 zijn beide interfaces te zien. Waar men bij *MoveIt*! (b) de robotarm naar de gewenste positie kan slepen, is de functionaliteit binnen RViz (a) een stuk beperkter.

2.4.3 MoveIt!

MoveIt! is een opensource padplanningssoftware, dat ontworpen werd voor trajectgeneratie en omgevingsmonitoring voor robotarmen. De software draait bovenop ROS en maakt gebruik van zijn berichtensysteem en common tools, zoals RViz en URDF. Het kan worden gezien als een extensie van RViz, gefocust op robotaansturing. Dit zorgt ervoor dat het invoeren van de robot via een URDF-bestand en het aansturen ervan gebruiksvriendelijker verloopt [16].

Eén van de basisdingen die *MoveIt*! kan, is het creëren van noodzakelijke trajecten voor de robotarm om deze naar een vooraf gedefinieerde locatie te verplaatsen. *MoveIt*! stelt een opeenvolging van bewegingen op die alle gewrichten uitvoeren om de gewenste positie te bereiken. Deze bewegingen kunnen met behulp van een animatie geëvalueerd en eventueel aangepast worden. Verder stelt *MoveIt*! robots ook in staat om een representatie van hun omgeving op te bouwen met behulp van sensoren, bewegingsplannen te genereren die de robot effectief en veilig in de omgeving bewegen en de bewegingsplannen uit te voeren terwijl de omgeving voortdurend wordt gecontroleerd op veranderingen [17].



(a) *(b)*

Figuur 13: (a) Weergave in RViz, (b) Weergave in MoveIt!

2.5 ROS-communicatie via web-API

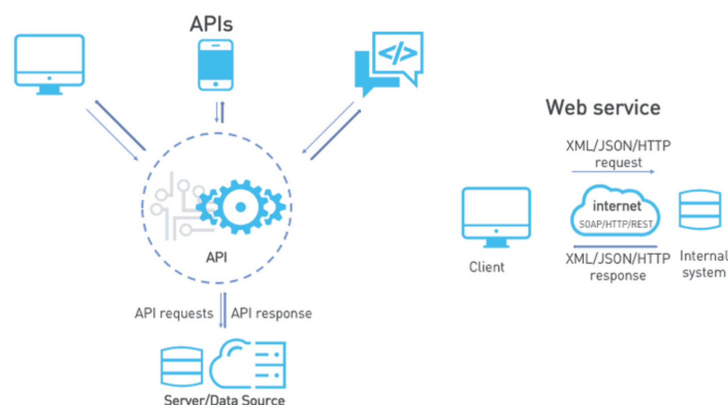
2.5.1 Wat is een web-API

Een Application Programming Interface of API is een software-interface dat de toegang definieert tot de functionaliteiten van een computerprogramma. Door een API kunnen andere applicaties software gebruiken zonder de details van de functionaliteiten of implementatie te kennen. Een web-API is namelijk een interface voor een webserver of webbrowser en is opgesplitst in twee onderdelen: een server- en clientzijde zoals schematisch afgebeeld op Figuur 14.

De serverzijde van de web-API is een programmeerbare interface die bestaat uit één of meerdere publiek gemaakte eindpunten (EN: *end points*) die een gedefinieerde request-response berichtensysteem bevatten. Deze berichten zijn geschreven in JSON of XML. Het uitvoeren van requests gebeurt door gebruik te maken van een HTTP-gebaseerde webserver (bv. Firefox).

Een endpoint is een belangrijk onderdeel van de serverzijde van de web-API. Zij specificeren waar alle informatiebronnen zich bevinden en hoe iemand ze kan oproepen. Deze dienen ook statisch te zijn. Indien dit niet het geval is, gaat code die voordien geschreven is, niet meer werken. Hierdoor is het aangeraden om de versie van de gebruikte API mee te geven bij het opvragen van de informatie.

Daarnaast is er de clientzijde van de web-API. Dit is een programmeerbare interface die de functionaliteit van een programma uitbreidt in een browser of andere HTTP-client. Door de clientzijde te implementeren in een applicatie kan deze gebruik maken van alle functionaliteiten en informatie die de server te bieden heeft. Een voorbeeld hiervan is de Spotify-applicatie. Deze maakt gebruik van softwarematige HTTP-*requests* om muziek af te spelen of om bijvoorbeeld een artiest te zoeken [18].



Figuur 14: Schema web-API en web Services [19]

2.5.2 Eigenschappen van de HIM web-API

In de huidige versie (2021.1) van de HIM web-API is het al mogelijk bijna alle functionaliteiten van de volledige HIM-software te gebruiken. Deze web-API kan momenteel al veel data opvragen en eventueel aanpassen. Alle functionaliteiten staan opgesomd in Tabel 1 [20].

Tabel 1: Functionaliteiten van de web-API van de HIM

Data Requests voor het zenden en ontvangen van data	
GET /units/	Geeft alle units weer
GET /units/{unitId}	Geeft de unit voor een bepaalde ID weer
GET /projects/	Geeft alle projecten weer
POST /projects/	Voegt één of meerdere projecten toe
GET /projects/{projectId}	Geeft het project voor een bepaalde ID weer
PATCH /projects/{projectId}	Past bepaalde eigenschappen van een project aan (bv. naam)
GET /projects/{projectId}/processes	Geeft alle processen van een bepaald project weer
POST /projects/{projectId}/processes	Voegt één of meerdere processen toe aan een bepaald project
GET /projects/{projectId}/processes/{processId}	Geeft het process voor een bepaalde ID weer
GET /projects/{projectId}/processes/{processId}/steps	Geeft de verschillende stappen van een bepaald proces weer
GET /projects/{projectId}/tasks	Geeft alle taken van een bepaald project
GET /projects/{projectId}/tasks/{taskId}	Geeft de taak van een bepaalde ID weer
GET /projects/{projectId}/detections	Geeft alle detecties van een bepaald project weer
POST /projects/{projectId}/detections	Voegt één of meerdere detecties toe aan een bepaald project
GET /projects/{projectId}/detections/{detectionId}	Geeft de detective van een bepaalde ID weer
PACTH /projects/{projectId}/detections/{detectionId}	Past bepaalde eigenschappen van een detectie aan
GET /projects/{projectId}/materials	Geeft alle materialen voor een bepaald project weer
POST /projects/{projectId}/materials	Voegt één of meerdere materialen toe aan een bepaald project

GET /projects/{projectId}/materials/{materialId}	Geeft het materiaal van een bepaalde ID weer
PATCH /projects/{projectId}/materials/{materialId}	Past bepaalde eigenschappen van een materiaal aan
POST /projects/{projectId}/ materials/addMaterialsToContainers	Voegt een materiaal toe aan een container
GET /projects/{projectId}/images	Geeft alle afbeeldingen voor een bepaald project weer
POST /projects/{projectId}/images	Voegt één of meerderen afbeeldingen toe aan een bepaald project
GET /projects/{projectId}/images/{imageId}	Geeft de afbeelding voor een bepaalde ID weer
DELETE /projects/{projectId}/images/{imageId}	Verwijdert de afbeelding voor een bepaalde ID
POST /projects/{projectId}/images/{imageId}/replace	Vervangt één of meerdere afbeeldingen voor een bepaald project
GET /projects/{projectId}/images/{imageId}/show	Geeft de afbeelding voor een bepaald ID weer
GET /projects/{projectId}/steps	Geeft de stappen van een bepaald project weer
POST /projects/{projectId}/steps	Voegt één of meerdere stappen toe aan een bepaald project

Operation Requests voor tijdens een operatie	
POST /units/{unitId}/projects/{projectId}/load	Laad het gegeven project op de (lopende) unit op
GET /units/{unitId}/loadedProject	Toont het gegeven project op de (lopende) unit
GET /units/{unitId}/variables/{variableName}	Toont de huidige waarde van een specifieke variabele
GET /units/{unitId}/variables	Geeft alle variabelen weer
PUT /units/{unitId}/variables	Verandert de waarde van een specifieke variabele
POST /units/{unitId}/processes/control	Controleert een proces op basis van een controle type (Next step, Previous step, Reset, Play, Pause, Restart)
POST /units/{unitId}/processes/ {processId}/scheduledTasks	Plant één of meerdere taken in
GET /units/{unitId}/processes/ {processId}/scheduledTasks	Geeft een lijst terug van de huidige geplande taken
GET /units/{unitId}/processes/{processId}/activeSteps	Geeft een lijst terug van de actieve stappen weer
GET /units/{unitId}/processes/{materialId}/containers	Geeft de containers gelinkt aan een bepaald materiaal weer
GET /units/{unitId}/processes/{containerId}/material	Geeft de materialen gelinkt aan een bepaalde container weer

De huidige versie van de HIM web-API ontbreekt echter nog een aantal informatiestromen en functionaliteiten die nuttig zijn voor mens-robotsamenwerking. Allereerst is het momenteel niet mogelijk om via de aanwezige dieptecamera objecten te herkennen. Indien deze de functionaliteit had om objecten te herkennen, zou de web-API deze informatie kunnen opvragen en de huidige manier van werken veel dynamischer maken. Momenteel worden aan alle gebruikte objecten in een assemblage detectiezones gegeven. Met objectherkenning zou de operator bijvoorbeeld zijn gereedschappen op eender welke positie kunnen plaatsen die comfortabel is voor deze persoon. In de huidige software moet de programmeur de positie van dit gereedschap manueel aanpassen.

Daarnaast kan de huidige versie van de HIM ook geen Cartesische coördinaten aflezen. Hiermee zou de HIM de posities van gebruikte onderdelen weergeven. De HIM zou met deze coördinaten een robotarm kunnen vertellen waar bepaalde objecten zich bevinden indien er ook een hand-eye kalibratie uitgevoerd wordt tussen camera en robot.

2.5.3 Softwarematige communicatie met de web-API

Externe communicatie met de HIM gebeurt via de web-API. Om te vermijden dat elk request handmatig gebeurt, dient dit softwarematig te gebeuren. Het basispakket van ROS is niet in staat om dergelijke HTTP-*requests* uit te voeren, waardoor gewerkt dient te worden met een externe softwarebibliotheek die dergelijke functies bevat.

Zoals eerder vermeld werkt ROS met meerdere programmeertalen. Zowel C++ als Python bevatten een geschikte bibliotheek. Er is gekozen voor C++ boven Python aangezien C++ over het algemeen sneller is dan Python. De C++ bibliotheek 'cpp-httplib' is in staat om alle soorten HTTP-requests uit te voeren en werkt zowel voor server- en client-zijde. Voor deze masterproef is enkel de client-zijde relevant, aangezien Arkite de server van de HIM beheert. Een voorbeeld van de structuur van een eenvoudig request is zichtbaar in onderstaande Figuur 15.

```
1   #include <iostream>
2   #include "./httplib.h"
3
4   int main() {
5     httplib::Client cli("http://cpp-httplib-server.yhirose.repl.co");
6
7     auto res = cli.Get("/hi");
8
9     if (res) {
10      std::cout << res->status << std::endl;
11      std::cout << res->body << std::endl;
12    }
13  }
```

Figuur 15: Voorbeeld van een eenvoudig GET-request

Om gebruik te maken van de externe sofwarebibliotheek dient men deze op te nemen in de code. Dit gebeurt op regel 2 met behulp van '#include "./httplib.h"'. De meeste van de geïmporteerde functies werken in op bepaalde objecten van klassen gedefinieerd door httplib. In dit geval is de klasse van het type 'Client', en noemt men het bepaald object 'cli' (zie regel 5). Hierop kunnen dan functies, zoals '.Get()' (regel 7), worden uitgevoerd. Het resultaat van de functie slaat men op in 'res', die bestaat uit een status (cijfercode over de status van het request) en een body (eigenlijke inhoud/ data van de request).

Naast het Get-request uit bovenstaand voorbeeld ondersteunt httplib ook Post-, Put- en Delete-methodes, welke eveneens gebruikt worden in de web-API van de HIM.

Hoofdstuk 3: Ontwikkeling van een ROS-driver voor de HIM

Dit hoofdstuk beschrijft de ontwikkeling van de ROS-driver voor de Human Interface Mate. Tot voorheen werkte de HIM als een relatief afgesloten systeem, waarbij communicatie met andere softwareapplicaties enkel via de interface van de web-API en enkele (industriële) communicatieprotocollen kan verlopen. De toevoeging van deze ROS-driver zorgt voor een uitbreiding van de externe communicatiemogelijkheden van de HIM. Door deze extensie is de HIM-technologie in staat te integreren in het domein van robotica, en dus ook mens-robotsamenwerking.

Het opzetten van de ROS-driver voor de Human Interface Mate kan opgesplitst worden in twee delen. Ten eerste wisselt de driver informatie uit met de HIM. Hiervoor maakt de ROS-driver gebruik van de aanwezige web-API, om zo de mogelijke Get-, Post-, Patch-, Put- en Delete-requests uit te voeren op de HIM. Daarnaast communiceert de driver deze informatie binnen ROS met behulp van topics en messages.

3.1 Beschikbare documentatie van de HIM

Zoals eerder vermeld werkt de HIM hoofdzakelijk in zijn eigen, afgesloten omgeving. Het aansturen van de HIM gebeurt via een gespecialiseerd softwarepakket waarmee de volledige functionaliteit van de HIM toegankelijk is. Wanneer men de HIM wil aansturen of aanpassen van buiten dit softwarepakket, zijn de mogelijkheden echter beperkt. De HIM bezit enkele (industriële) communicatieprotocollen om bijvoorbeeld extern gereedschap aan te sturen, maar ook een web-API.

De web-API is in staat commando's te sturen naar en informatie te ontvangen van de HIM. Deze werd origineel ontwikkeld om het integreren en testen van nieuwe functies te vergemakkelijken. Een volledige lijst van de functionaliteit van de web-API (versie 2021.1) is terug te vinden in Tabel 1 in paragraaf 2.5.2. Daarnaast kan de web-API bediend worden via een testtool in de vorm van een HTML-pagina. Dit bestand is beschikbaar in de installatiebestanden van de HIM en gebruikt een webpagina om de verschillende commando's op te vragen bij de web-API.

Het opvragen en sturen van informatie via de web-API geschiedt aan de hand van een URL. Deze URL kan parameters, zoals een unit- of project-ID, bevatten zodat de HIM weet op welke unit of project dit betrekking heeft. Onderstaand voorbeeld (Figuur 16) toont de request-URL voor het opvragen van de verschillende processen van een bepaald project.

```
Voorbeeld: Processen opvragen van een bepaald project

http://myhimserverurl:7777/api/v1/projects/{projectId}/processes/
```

Figuur 16: Voorbeeld request-URL

Bij het sturen van commando's (Post-, Patch- en Put-requests) bevat de request, naast de URL, meestal een extra 'request body'. Deze body omvat de nodige informatie voor dat specifiek commando, zoals weergegeven in onderstaand voorbeeld (Figuur 17). Hierbij is voor het creëren van een nieuw proces een bepaalde naam, ID, type en 'dynamic' nodig.

```
Voorbeeld: Toevoegen van een body aan een request

Request URL:
http://myhimserverurl:7777/api/v1/projects/{projectId}/processes/

Request body:
[
  {
    "Name": "Nieuw Proces",
    "Id": 123454321,
    "Type": "Process",
    "Dynamic": false
  }
]
```

Figuur 17: Voorbeeld request body

3.2 Opbouw van de ROS-driver

De ontwikkelde ROS-driver bevat meerdere nodes die gebruik maken van ROS-messages en -parameters voor het behandelen van de binnenkomende en uitgaande data. De Get-requests omvatten één enkele node terwijl de verschillende Post-, Patch-, Put- en Delete-requests ieder gebruik maken van een server- en client-node. Het uitvoeren van de Get- en server-nodes gebeurt bij het opstarten van de driver door middel van een launch-file. Dit zorgt ervoor dat de Get-requests de verworven informatie voortdurend ter beschikking stellen op de toegewezen ROS-topics (i.e. publishen). De server-nodes daarentegen wachten op een commando van de client-zijde via een ROS-service zodat deze niet continu dezelfde informatie doorsturen naar de HIM.

3.3 Communicatie met de HIM via ROS

De web-API van de HIM kan informatie opvragen van en doorsturen via HTTP-requests. Zoals eerder vermeld is het basispakket van ROS/C++ niet in staat om dergelijke requests uit te voeren en is het nodig beroep te doen op een externe softwarebibliotheek, in dit geval 'cpp-httplib'[1]. Bovenaan de code van iedere node die communiceert met de web-API moet men deze softwarebibliotheek importeren.

Indien de nodige parameters voor een bepaald request aanwezig zijn (en indien de client het aanvraagt in het geval van Post-, Patch-, Put- en Delete-requests), voert de node het http-request uit bij de server van de HIM. Een voorbeeld van een dergelijk request is terug te vinden in paragraaf 2.5.3. De bekomen status en body worden weggeschreven en later gebruikt voor respectievelijk het controleren van het succes van het request en het opslaan van de bekomen informatie.

3.3.1 Get-Requests

De huidige versie van de web-API bevat 23 unieke Get-requests. Hoewel deze allemaal verschillen in request-URL en output, volgen ze een gelijkaardige opbouw voor de ROS-driver. Een voorbeeld van dergelijk Get-request is terug te vinden in Figuur 18. Hieronder worden de verschillende onderdelen van de code toegelicht, met in de titels steeds een verwijzing naar de overeenkomende regels in de code.

1. **Includeren van de nodige bestanden en datatypes (regel 1-10)**

   Allereerst zijn de nodige softwarebibliotheken geïmplementeerd: 'cpp-httplib' en 'JSON for Modern C++'. Deze behandelen respectievelijk het uitvoeren van de HTTP-requests en het uitlezen van de body via een JSON-structuur, aangezien de web-API diezelfde structuur hanteert. Daarnaast importeert men hier de nodige ROS-messages.

2. **Initialiseren van de ROS-node en handle(s) (regel 15-16)**

   Het initialiseren van de node zorgt ervoor dat deze argumenten uitleest om zo de node-naam en gebruikte namespace te achterhalen. Het creëren van een *node handle* start de node op.

3. **Definiëren van de nodige parameters (regel 19)**

   Vooraleer de node de nodige parameters kan opvragen aan de server heeft het een plaats nodig om deze weg te schrijven. Het definiëren van node-specifieke objecten met bijhorende klassen gebeurt hier.

[1] Softwarebron 'cpp-httplib': https://github.com/yhirose/cpp-httplib

```
1  #include <sstream>
2  #include <string>
3
4  #include "ros/ros.h"
5
6  #include "himdriver/projects.h"
7  #include "himdriver/projectsArray.h"
8
9  #include <httplib.h>
10 #include <nlohmann/json.hpp>
11
12 int main(int argc, char **argv)
13 {
14         //Initialiseer de nodige node en de handles
15         ros::init(argc, argv, "getProject");
16         ros::NodeHandle nh;
17
18         //Defineer de nodige parameters
19     std::string projectId = "";
20         bool sendParameterError = false;
21
22         //Defineer de publisher en topic
23         ros::Publisher chatter_pub = nh.advertise<himdriver::projectsArray>("himdriver_getProject", 1000);
24         ros::Rate loop_rate(1);
25
26         while (ros::ok())
27         {
28         //Parameters opvragen van server
29         ros::param::get( "/himdriver_projectId" , projectId);
30
31                 //Defineer de http request
32                 httplib::Client cli("10.11.98.105", 7777);
33                 std::string request = "/api/v1/projects/" + projectId + "/";
34
35                 //Defineer de message
36                 himdriver::projectsArray msg;
37                 std::stringstream ss;
38                 std::stringstream ss2;
39
40                 //Controleren op Errors
41                 if(projectId != "") {
42                         auto res = cli.Get(request.c_str());    //data ophalen uit de server
43                         ss2 << res->status;
44                         if(ss2.str() == "404"){
45                                 ROS_ERROR("404: No Project for given ID");
46                                 ROS_INFO("Removing parameter: projectId");
47                                 ros::param::set( "himdriver_projectId", "");
48                         }
49                         else{
50                                 ss << res->body;
51                                 himdriver::projects project;
52
53                                 //json structuur opzetten en uitlezen; toevoegen aan msg
54                                 auto json_response = nlohmann::json::parse(ss.str());
55                                 if(json_response["Description"].is_null()){project.description = "<No Description>";}
56                                 else{project.description = json_response["Description"];}
57                                 project.type = json_response["Type"];
58                                 project.id = json_response["Id"];
59                                 project.name = json_response["Name"];
60                                 msg.projects.push_back(project);
61
62                         }
63                 }
64
65         //Geef mee dat er geen data gevonden is, indien dit zo is
66                 std::string str = ss.str();
67         if(str.find("Id") == std::string::npos && projectId != ""){
68                     if(!sendParameterError){
69                                 ROS_ERROR("getProject: No Data found");
70                                 sendParameterError = true;
71                         }
72         }
73                 else if(projectId == ""){
74                         if(!sendParameterError){
75                                 ROS_ERROR("getProject: Check parameters: projectId");
76                                 sendParameterError = true;
77                         }
78                 }
79                 else{sendParameterError = false;}
80
81                 //Publish de message
82                 chatter_pub.publish(msg);
83
84                 //Uitvoeren en slapen
85                 ros::spinOnce();
86                 loop_rate.sleep();
87
88         }
89  return 0 ;
90 }
```

Figuur 18: Volledige code GetProject-request

4. Definiëren van de publisher en het topic (regel 23-24)

De publisher is een ROS-node die een specifiek type ROS-message op een bepaald ROS-topic plaatst. Geïnteresseerde nodes (nl. de subscribers) kunnen de messages later van deze topics aflezen.

5. Opvragen van parameters bij de parameterserver (regel 29)

De parameterserver is een bibliotheek die parameters (bv. strings, integers, booleans, etc.) opslaat. Verschillende nodes kunnen deze parameters opvragen en aanpassen, om er zo in de driver gebruik van te maken. In deze stap van de code vraagt de node de eerder aangemaakte parameters op uit de server en schrijft deze weg in de eerder aangemaakte objecten.

6. Definiëren van de HTTP-client (regel 32-33) en uitvoeren van het Get-request (regel 42)

Definiëren van de HTTP-client en uitvoeren van het Get-request met behulp van de externe softwarebibliotheek 'cpp-httplib'. Voor meer informatie wordt verwezen naar paragrafen 2.5.3 en 3.3.

7. Definiëren van de ROS-message (regel 41-79)

Het resultaat van de uitgevoerde request (status en body) dient gecontroleerd en verwerkt te worden. Dit gebeurt in drie stappen: controleren op foutmeldingen, uitlezen van de body en meegeven wanneer er geen data gevonden is. Figuur 19 geeft de pseudocode van dit onderdeel weer.

Algorithm 1: Data operations

```
1  if required parameters not empty then
2  |   Execute HTTP-request
3  |   Store status
4  |   if status = error then
5  |   |   Post error message
6  |   else
7  |   |   Store body
8  |   |   Add body to message
9  |   end
10 end
11 if body is empty AND required parameters not empty then
12 |   Error = Available
13 |   Post error message
14 else if required parameters empty then
15 |   Error = Available
16 |   Post error message
17 else
18 |   Error = Unavailable
19 end
```

Figuur 19: Uitvoeren van het request en verwerken van de opgehaalde data

a. Controleren op foutmeldingen (regel 4-5, Figuur 19)

Ieder request bevat een status en body. De status bestaat uit een driecijferige die de toestand van het request beschrijft. Deze stap controleert of de server al dan niet een error terugstuurt. Indien dit niet het geval is, kan men verder gaan met het uitlezen van de data en het toevoegen van deze data aan de ROS-message. Hoofdstuk 3.3.3 beschrijft de mogelijke errors in nader detail.

b. Uitlezen van de data en toevoegen aan de ROS-message (regel 6-8, Figuur 19)

De body is gestructureerd volgens JavaScript Object Notation (JSON). Dit is een tekst gebaseerd formaat voor het doorsturen van gestructureerde data in web applicaties (en dus ook bij de web-API). Het verwerken van de data gebeurt aan de hand van een *JSON-parser*. In dit geval is er gebruik gemaakt van de softwarebibliotheek 'JSON for Modern C++'[2], ontwikkeld door Niels Lohmann.

De JSON-parser wordt geïmplementeerd in stap 1 en vormt de body 'ss.str()' om naar JSON-formaat zodat de verschillende onderdelen van de message eenvoudig te onderscheiden zijn en gestructureerd in de ROS-message geplaatst kunnen worden. Onderstaande Figuur 20 geeft dit weer voor de verschillende onderdelen op basis waarvan een assemblageproces gedefinieerd wordt in de HIM-software.

```
ss << res->body;

//json structuur opzetten en uitlezen; toevoegen aan msg
auto json_response = nlohmann::json::parse(ss.str());
himdriver::process process;
process.dynamic = json_response["Dynamic"];
process.comment = json_response["Comment"];
process.type = json_response["Type"];
process.id = json_response["Id"];
process.name = json_response["Name"];

msg.processes.push_back(process);
```

Figuur 20: JSON-structuur uitlezen

[2] Softwarebron 'JSON for Modern C++': https://json.nlohmann.me/

c. Meegeven dat er geen data gevonden is, indien dit het geval is (regel 11-16, Figuur 19)

Ondanks dat de status van het request geen error geeft, is het toch mogelijk dat de body geen data bevat. Dit heeft twee mogelijke oorzaken: de juiste parameters zijn niet aanwezig en het request is dus nooit uitgevoerd, of de HTTP-server stuurt een leeg antwoord terug. In beide gevallen meldt het topic het probleem in de vorm van een foutmelding aan de gebruiker.

8. **Publishen van de ROS-message (regel 82)**

De eerder gedefinieerde publisher plaatst de samengestelde message op het relevante topic.

3.3.2 Patch-, Post-, Put- en Delete-requests

Naast de reeds besproken Get-requests bevat de web-API van de HIM ook 14 mogelijke Post-, Patch-, Put- en Delete-requests. Zoals eerder vermeld maken deze requests gebruik van een server- en client-node, zodat ze niet continu dezelfde informatie doorsturen naar de HIM.

De server-nodes zijn qua opbouw vergelijkbaar met deze van de Get-requests, met enkele aanpassingen die het doorsturen van informatie mogelijk maken. Zo is het nodig de gebruikte service te definiëren in het begin van het bestand. Dit gebeurt zodat de client een verzoek kan sturen voor het uitvoeren van het request. De server-node stuurt dan enkel informatie door wanneer de client dit aanvraagt. De gebruikte service is voor alle requests dezelfde, de client-node is verantwoordelijk voor het aanspreken van de juiste server. Waar de Get-requests een body uitleest, dienen de Post-, Put-, Patch- en Delete-requests een body op te stellen om door te sturen naar de server van de HIM. Ook hier maken de nodes gebruik van een JSON-structuur voor het vormen en structureren van de body, in een formaat dat compatibel is met de web-API. Elke request heeft vereiste en optionele parameters. In de volgende stap controleert de node of de nodige parameters aanwezig zijn alvorens het Post-request uit te voeren. Indien dit niet het geval is, informeert de node de gebruiker over de vereiste en optionele parameters van dat bepaalde request.

De client-nodes zijn korte bestanden met als doel de overeenkomende servers op te roepen via een ROS-service. Wanneer een server is opgeroepen zal deze zijn specifieke request uitvoeren, zoals hierboven beschreven. Tot slot meldt de node de gebruiker of het contacteren van de server al dan niet succesvol is verlopen. De pseudocode met de opbouw van deze client-nodes is terug te vinden in Figuur 21.

Algorithm 2: Client node

```
1 Initialise node
2 Define nodehandle
3 Define service
4 if Service call = succes then
5 |  Info: Succes
6 else
7 |  Error = Available
8 |  Post error message
9 end
```

Figuur 21: Opbouw client-nodes

3.3.3 Foutmeldingen

Vooraleer de nodes http-requests uitvoeren controleren ze de aanwezigheid van de juiste parameters. Dit gebeurt zodat er geen ongeldige requests gebeuren bij de server van de HIM. De afwezigheid van bepaalde parameters zou namelijk voor een onvolledige request-URL kunnen zorgen.

Het resultaat van het HTTP-request bevat een status en een body. De status bestaat uit een drie-cijferige code die de toestand van de server beschrijft. Bij het uitvoeren van requests op de web-API van de HIM is het mogelijk om vier verschillende codes tegen te komen:

- **200**: “Goed gevolg: OK”
- **204**: “Goed gevolg: Geen inhoud”
- **400**: “Aanvraagfout: Foute aanvraag”
- **404**: “Aanvraagfout: Niet gevonden”

Iedere node controleert, alvorens de body weg te schrijven, of het request succesvol werd afgerond. Het kan zich echter voordoen dat de node een lege body ontvangt vanuit de server van de HIM, ondanks dat het een positieve statuscode ontvangt. De nodes zijn zo opgebouwd dat ze dit kunnen opsporen om problemen met het verwerken van een lege body te vermijden.

Wanneer de statuscode 400 of 404 ontvangen wordt, of er een probleem is met de parameters of body van een request, deelt de node dit mee aan de gebruiker zodat deze weet waar het probleem zich situeert.

3.4 Conclusie

Dit hoofdstuk beschreef het opstellen van de ROS-driver voor de Human Interface Mate. Iedere ROS-node werd geschreven in C++ en behandelt een bepaald request bij de web-API van de HIM met behulp van de softwarebibliotheek ‘cpp-httplib’. De Post-, Patch-, Put- en Delete-requests bevatten naast de uitvoerende server-node een tweede client-node, die de requests op het gepaste tijdstip uitvoert. In het totaal zijn er 37 verschillende requests mogelijk.

De web-API van de HIM bestaat in zijn huidige versie uit 40 requests. Het plaatsen, vervangen en tonen van afbeeldingen in een bepaald project zijn weggelaten bij het opstellen van de ROS-driver, aangezien dit via ROS onnodig gecompliceerd zou zijn en voorlopig niet vereist is voor de gebruikte use-case. Van de 37 aanwezige requests ondervinden twee requests (‘AddMaterialToContainer’ en ‘GetUnitVariables’) problemen bij het uitvoeren. De oorsprong van deze problemen ligt echter bij de huidige versie van de server van de web-API, welke de requests momenteel niet herkent. Arkite is hiervan op de hoogte gesteld.

De verschillende nodes zijn geschreven met oog voor mogelijke foutmeldingen en geven duidelijke berichten aan de gebruiker over de actuele toestand. Dit zorgt voor een uitgebreide, maar gebruiksvriendelijke ROS-driver voor de Human Interface Mate. De volledige code van de ROS-driver voor de HIM is beschikbaar op de GitLab repository van KULeuven[3].

---

[3] GitLab repository: https://gitlab.kuleuven.be/ACRO/masters-theses/mp20-21-brandon-brecht/20-21-brandon_brecht

Hoofdstuk 4: Opzetten van een framework voor mens-robot-samenwerking

Dit hoofdstuk beschrijft de integratie van de ontwikkelde ROS-driver voor de HIM in een framework voor mens-robotsamenwerking. Dit framework zorgt voor een algemene structuur waarin ieder onderdeel, noodzakelijk voor mens-robotsamenwerking, met elkaar verbonden is. Als eerste onderdeel van dit framework is het nodig om de huidige situatie van de assemblage te weten te komen. Hiervoor wordt de ROS-driver van de HIM gebruikt. Dit is een eerste toepassing waarvoor deze driver in realiteit gebruikt kan worden. Verder is er een beslissend orgaan dat aan de hand van de huidige situatie van de assemblage, de robotica de juiste aansturingen bezorgt.

4.1 Detecties via de Human Interface Mate (HIM)

Om de huidige situatie van de assemblage te weten te komen is er gebruik gemaakt van de ROS-driver van de HIM, meer bepaald om de geregistreerde detecties door de HIM op te vragen. Zoals eerder besproken in paragraaf 2.2 is het mogelijk om een beeld te maken van de werkplaats op twee verschillende tijdstippen en hierbij te focussen op één bepaald gebied. Alle veranderingen in dit gebied zullen opgeslagen worden tot één detectie. Dit maakt het mogelijk om te detecteren wanneer een object aanwezig is of niet. Daarnaast heeft de HIM ook de mogelijkheid om te detecteren of er zich iets boven het gedefinieerde object bevindt. Indien dit het geval is, zal de HIM niet kunnen zien of het object nog aanwezig is of niet, aangezien dit object niet meer in zicht is. In deze situatie zal de HIM de laatste status van dit object onthouden. De pseudocode van deze functionaliteit is terug te vinden in Figuur 22.

Vervolgens kan een detectie en zijn status met behulp van de ROS-driver opgevraagd worden aangezien deze status opgeslagen is onder de variabelen van het gebruikte project. Het framework vraagt echter niet één detectie per keer op, maar definieert in regel 2 een lijst van de aanwezige assemblageobjecten. Hierna vraagt de ROS-driver in regel 5-8 voor ieder van deze objecten de status op en slaat deze op in een tweede lijst. Zodra dit doorlopen is wacht het programma vijf seconden op regel 9, waarna deze ieder object opnieuw controleert en zijn status wegschrijft in een derde lijst in regel 10-13. Als laatste vergelijkt onderstaande code deze twee lijsten en geeft hij de verschillen ertussen terug in regel 14-17.

Algorithm 3: Check detections for changes

```
   Result: Array filled with the changed detection after 5 seconds
1  if Asked to perform a check for detection changes then
2  |  Define array with all detections as D
3  |  Initialise check arrays C1 and C2
4  |  Initialise result array R
5  |  for x ∈ D do
6  |  |  Request status(x)
7  |  |  Add status to C1
8  |  end
9  |  Wait 5 seconds for user to grab objects
10 |  for x ∈ D do
11 |  |  Request status(x)
12 |  |  Add status to C2
13 |  end
14 |  for x ∈ lenght(D) do
15 |  |  if C1[x] does not equal C2[x] then
16 |  |  |  Add x to R
17 |  |  end
18 |  end
19 |  return R
20 end
```

Figuur 22: Detectie-algoritme

4.2 Aansturing van de UR5-robot

Daarnaast is het nodig dat het framework de nodige robotica kan aansturen. In het geval van dit framework is er gekozen voor een UR5-robot aangezien deze een grote variëteit van assemblages mogelijk kan maken door zijn reikwijdte en kracht. Daarnaast is de UR5-robot een cobot en dus ontworpen voor mens-robotsamenwerking.

Het aansturen van de robot gebeurt door middel van de beschikbare ROS-driver[4]. Deze werkt met behulp van een action client en server. Hierbij speelt het framework de rol van de client en stuurt het de nodige messages naar de action server die op basis van deze messages de robot aanstuurt.

De gebruikte messages zijn van het type 'FollowJointTrajectoryGoal' (zie Tabel 2). Hierbij is het allereerst nodig om de gebruikte joints te benoemen en deze in de message te plaatsen onder de variabele 'joint_names'. Zodra deze zijn ingevuld is het mogelijk om een traject te definiëren. De messages hebben allemaal de variabele 'trajectory.points', een lijst van hoeken in radialen die de hoeken definiëren tussen de eerder beschreven joints. Ook is ieder punt beschreven met een tijdswaarde. Deze beschrijft de tijd waarin de robot zich naar de gegeven positie beweegt. Op deze manier kan iedere positie van een assemblage gedefinieerd worden om later te verzenden naar de action server. Deze voert op zijn beurt de nodige assemblage uit.

Tabel 2: Betekenis parameters 'FollowJointTrajectoryGoal'

Onderdeel	Parameter	Betekenis
Trajectory	Header	Standaard metadata.
	Joint_names	Lijst van namen van iedere joint.
	Points	Beschrijving van ieder punt in een te volgen beweging als lijst van hoeken en eventuele snelheden/ acceleraties samen met de doorlooptijd van de beweging
Path_tolerance		Beschrijft de toleranties van een joint tijdens het uitvoeren van een beweging.
Goal_tolerance		Beschrijft de toleranties van de eindpositie van een joint.
Duration goal_time_tolerance		Beschrijft de tolerantie op de doorlooptijd van een beweging.

[4] Softwarebron ROS-driver : https://github.com/UniversalRobots/Universal_Robots_ROS_Driver

4.3 Robotiq 2F-85 grijper

Vervolgens is er gebruikt gemaakt van een Robotiq 2F-85 als grijper voor de UR5-robot. Deze grijper is echter niet zoals de UR5 ontwikkeld voor mens-robotsamenwerking en stopt niet met knijpen wanneer deze per ongeluk in contact komt met een mens. Desondanks is er gekozen voor hiermee te werken omdat het een eenvoudige interfacing heeft via USB, een ROS-driver en een grote slag waar mee het mogelijk is om producten met verschillende afmetingen te grijpen.

Het aansturen van deze grijper in het framework gebeurt ook hier door middel van de beschikbare ROS-driver op de productwebsite[5]. Deze driver leest de inkomende messages op het topic 'Robotiq2FGripperRobotOutput' in en stuurt op basis hiervan de grijper aan. In het framework is er een publisher geschreven die op dit topic messages stuurt. De messages zijn van het type 'Robotiq2FGripper_robot_output' en iedere component ervan staat uitgelegd in Tabel 3.

Tabel 3: Betekenis parameters Robotiq grijper

Parameter	Betekenis
rACT	Geef een 1 mee om de grijper te activeren of een 0 om deze te resetten.
rGTO	Geef een 1 om de grijper te laten bewegen naar de aangegeven positie.
rATR	Geef een 1 om de grijper automatisch te openen wanneer de noodstop van de robot ingedrukt is.
rPR	Geef dit een waarde tussen 0 en 255 om de grijper te sluiten of te openen tot een bepaalde positie. Hierbij is 0 volledig open en is 255 volledig gesloten.
rSP	Geef dit een waarde tussen 0 en 255 om de snelheid van het openen of sluiten van de grijper te bepalen. Hierbij staat 0 voor de laagste snelheid en 255 voor de hoogste snelheid.
rFR	Geef dit een waarde tussen 0 en 255 om de kracht te bepalen waarmee de grijper zal drukken op het geklemde object. Hierbij is 0 de laagste kracht en 255 de hoogste kracht.

Om de grijper te besturen zijn er in het framework drie functies aangemaakt: 'gripper_activate', 'gripper_close' en 'gripper_open'. Deze functies stellen ieder een message op en plaatsen deze door middel van de publisher op het eerder besproken topic van de grijper.

[5] Softwarebron ROS-driver : https://github.com/ros-industrial/robotiq

4.4 Action server en client

Als laatste is er nog de action server en client zelf. Deze zorgen voor de verbinding tussen de robotactieplanner, de eerder beschreven robotica en de ROS-driver voor de HIM. Hierbij speelt de gebruikte robotactieplanner de rol spelen van client en roept deze de functies op van de action server.

De client kan drie mogelijke functies oproepen. De eerste is het uitvoeren van een assemblage. Hierbij selecteert de client een assemblage en voert de server de geselecteerde assemblage uit door het sturen van de gepaste commando's naar de robot en grijper. Daarnaast is er de detectiefunctie die, zoals eerder beschreven in paragraaf 4.1, voor een bepaalde periode de aanwezigheid van alle objecten controleert en daarna de veranderingen doorstuurt naar de client. Als laatste is er nog de wachtfunctie. Hierbij doet de server niets en wacht deze op het volgende commando van de action client.

Het oproepen van de functies gebeurt door middel van *action messages*. Deze zijn opgebouwd volgens de structuur uit Tabel 4.

Tabel 4: Structuur action messages

Onderdeel	Type	Parameter	Betekenis
Goal	String	Assembly	Selecteert het nummer van de gewenste assembly
	Bool	Wait	Geeft aan of er gewacht moet worden of niet.
	Bool	Detection	Geeft aan of alle objecten gecontroleerd moeten worden of niet.
Result	Bool	Result	Geeft aan of de actie een succes was of niet.
	StringArray	Detections	Indien bovenstaande Detection-parameter op True staat bevat deze StringArray de naam van alle detecties die van toestand verandert zijn.
Feedback	String	Comment	Geeft informatie terug omtrent de status tijdens het uitvoeren van een actie.

4.5 Conclusie

Dit hoofdstuk bespreekt alle onderdelen van het samengestelde framework voor mens-robotsamenwerking. Allereerst heeft ROS opnieuw een grote rol gespeeld in dit framework aangezien een action server is gebruikt als basis. Voor de communicatie tussen de client en server is er dan ook gebruikt gemaakt van action messages. Verder maakte ROS het ook mogelijk om de UR5-cobot en Robotiq 2F-85 grijper aan te sturen door middel van hun open source ROS-drivers.

Dit framework was een eerste toepassing van de eerder ontwikkelde ROS-driver en toont de flexibiliteit ervan. Enkel de detecties zijn namelijk gebruikt in deze toepassing. Hierbij moet de ROS-driver echter iedere aangegeven detectie aflopen en de status opvragen. Dit komt omdat de huidige versie van de web-API van de HIM het momenteel nog niet toelaat om de situatie van meerdere variabelen tegelijkertijd op te vragen. Arkite is hier echter wel mee bezig en dit zou het ontwikkelde programma versnellen en vereenvoudigen.

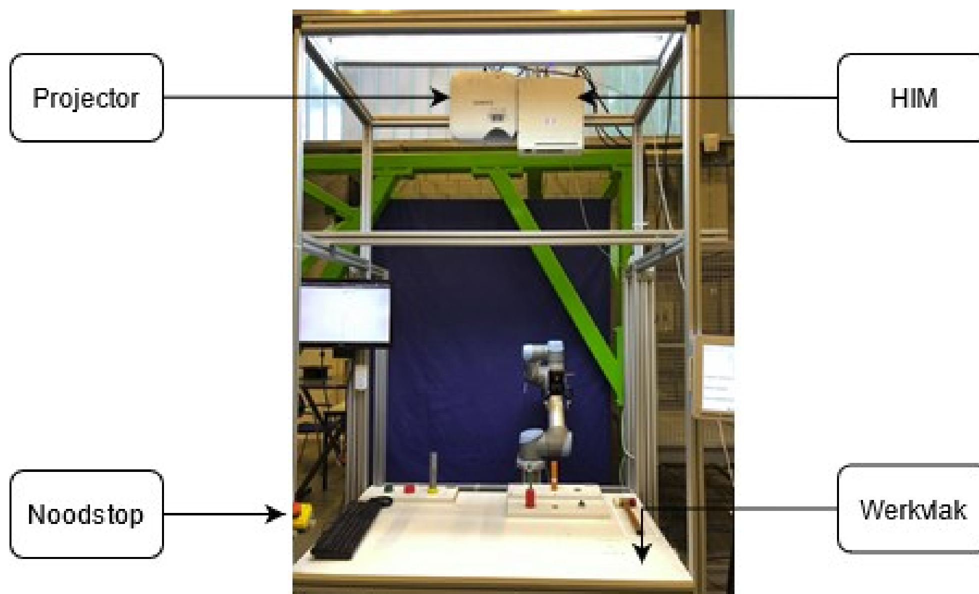
Na ontwikkeling van deze server kan er geconcludeerd worden dat deze modulair is. De detectiefunctie en de gebruikte robotica kan vervangen worden zonder de algemene structuur van het framework aan te passen. Hiervoor kan namelijk ieder onderdeel gebruikt worden met gelijkaardige functionaliteit en een beschikbare ROS-driver.

Hoofdstuk 5: Demonstratie van de ROS-driver voor de HIM in mens-robotsamenwerking

Dit hoofdstuk beschrijft de ontwikkeling en uitvoering van een demonstrator voor de validatie van de ROS-driver voor de HIM in een framework voor mens-robotsamenwerking. Hierbij helpt het eerder beschreven framework uit hoofdstuk 4, met behulp van de UR5-cobot en Robotiq-grijper, een persoon bij het assembleren van een zesdelig product in een assemblagevolgorde naar keuze.

5.1 Opbouw van de demonstrator

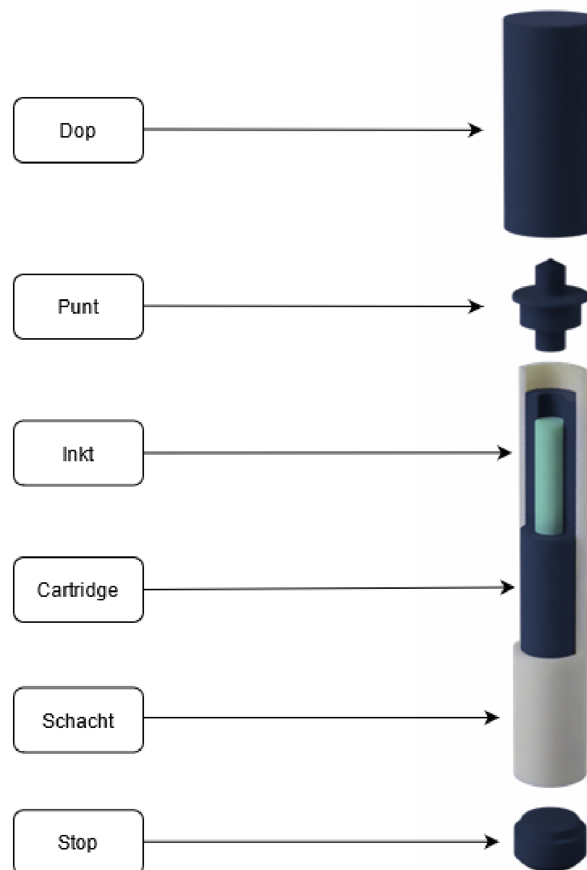
Om de demonstrator mogelijk te maken is er allereerst een opstelling nodig. Deze bestaat uit een vaste positie voor de UR5-cobot, een werkvlak, een frame boven dit werkvlak voor het plaatsen van de HIM en projector waarbij deze een volledig zicht hebben op het werkvlak en een noodstop voor de UR5-cobot. Hiervoor is een eerder ontworpen constructie van ACRO gebruikt die voldoet aan al deze karakteristieken (zie Figuur 23).



Figuur 23: Opstelling demonstrator

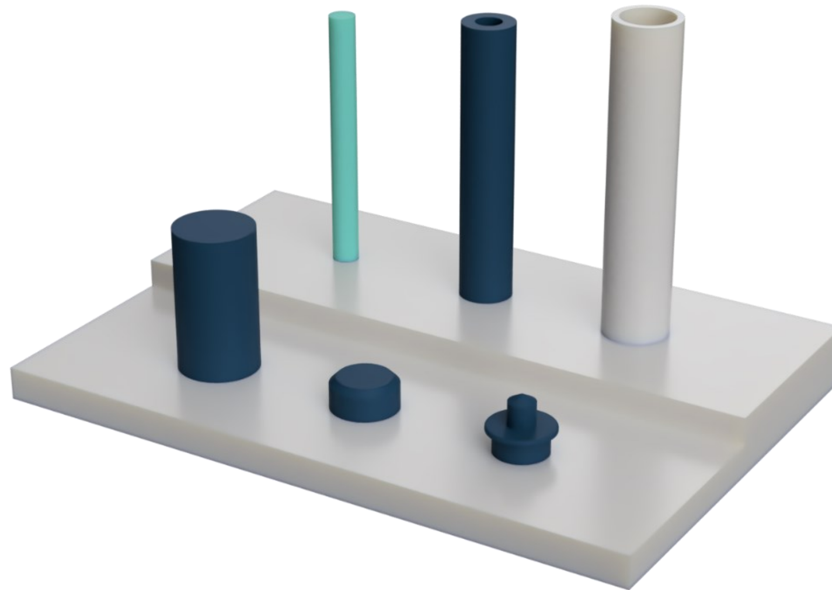
Voor de opstelling is één desktopcomputer gebruikt. Deze is direct aangesloten op de stuurkast van de UR5-cobot en zit verbonden op het zelfde netwerk als de HIM. Dit laatste is nodig om HTTP-requests van de HIM uit te voeren op een andere computer via het IP-adres van de HIM. Verder is de Robotiq-grijper aangesloten via USB met deze computer. Het scherm van deze computer is gemonteerd op het frame van de gebruikte opstelling. Zo heeft de operator een zicht op de beschikbare informatie zoals bijvoorbeeld de gedetecteerde onderdelen en gemaakte keuzes van de robotactieplanner.

Verder is er gebruik gemaakt van de balpen van Bourjault. Dit is een balpen bestaande uit zes onderdelen namelijk: schacht, cartridge, inkt, punt, dop en stop (zie Figuur 24). In de maakindustrie wordt de assemblage van een product ook vaak verdeeld over verscheidene werkstations, waarbij ieder werkstation een subassemblage voor zijn rekening neemt. De complexiteit van een dergelijke subassemblage wordt bepaald door de fysieke afmetingen van het werkstation, het te assembleren product, het aanwezige gereedschap in een dergelijk station en de vaardigheden van de operator. Bij te veel taken (en onderdelen) kan de operator namelijk overweldigd worden door de complexiteit, dat de kwaliteit en productiviteit nadelig zou kunnen beïnvloeden. In dat opzicht is deze zesdelige assemblage een passende representatie van een product dat geassembleerd zou kunnen worden door een industrieel werkstation.



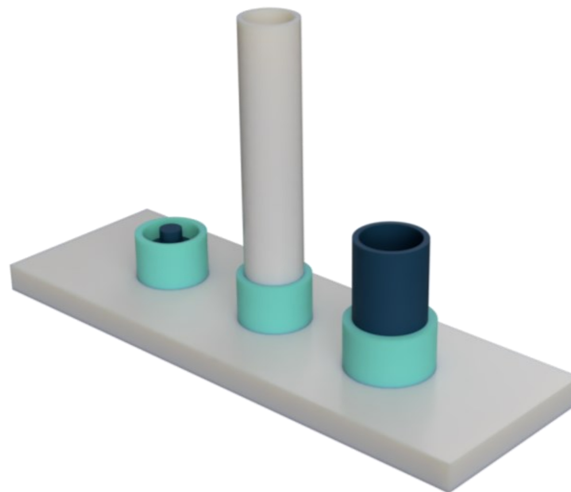
Figuur 24: Onderdelen van Bourjault's balpen

Vervolgens zijn er twee houders ontworpen om de onderdelen in te positioneren. De onderdelen hebben vaak een zeer kleine basis en de UR5-cobot zit vast gemonteerd aan hetzelfde frame. Hierdoor ondervinden deze onderdelen lichte schokken door de bewegingen van deze cobot en is het mogelijk dat deze omvallen. Er is daarom gekozen om alle zes onderdelen te plaatsen in een houten plank waarin specifieke openingen zijn voorzien voor ieder onderdeel (zie Figuur 25). In de industrie gebeurt dit ook, maar dan met klem- en opspangereedschap. De plank is vervolgens vastgemaakt aan het frame en de werktafel met enkele L-ijzers. Door middel van de nauwe toleranties van de openingen is het mogelijk om te garanderen dat de onderdelen op dezelfde posities blijven staan.

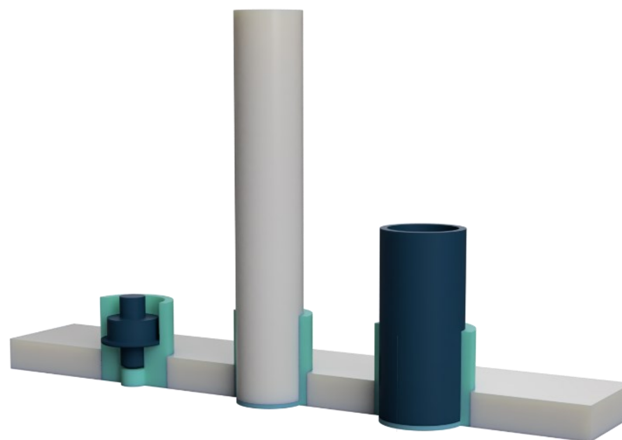


Figuur 25: Opslag onderdelen

Verder is het voor de UR5-cobot niet mogelijk om sommige assemblages van de balpen te assemberen zonder deze eerst te kunnen plaatsen in een tussenpositie. Hierdoor is er gekozen om een tweede houder te ontwerpen waarin de cobot voor iedere mogelijke assemblage de vereiste onderdelen kan plaatsen in een tussenpositie zoals zichtbaar is in Figuur 26(a). Hierbij is er gekozen om te werken met 3D-geprintte houders en deze te plaatsen in de openingen van de houten plank. De kunststoffen houders zorgen ervoor dat ook onderdelen met gebogen steunvlak correct en stabiel gepositioneerd zijn door een aangepaste opening te voorzien (zie Figuur 26(b)). De plank zorgt er op zijn beurt voor dat de houders zelf correct geplaatst zijn en is zelf ook vastgemaakt aan het frame zoals de eerste houder.



(a)



(b)

Figuur 26: (a) Houders tussenposities assemblages
(b) Doorsnede houders tussenposities assemblages

5.2 Opzetten van een demonstrator

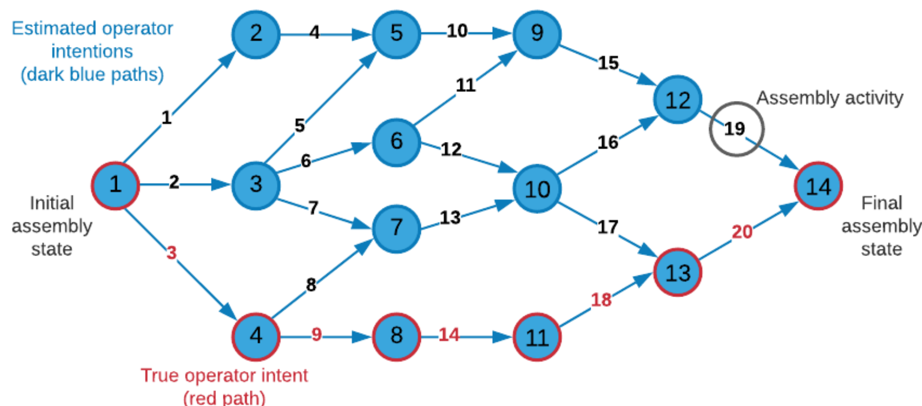
5.2.1 Assemblagetoestandsgraaf van Bourjault's balpen

Allereerst is er voor het aansturen van de gebruikte action server, een action client nodig. Zoals eerder vermeld in hoofdstuk 4 is hiervoor gebruik gemaakt van een robotactieplanner. Om specifiek te zijn is er gekozen voor een POMDP-planner.

Deze schat wat de intentie is van de operator en kiest op basis hiervan een assemblageactie voor de robot. Een intentie is gedefinieerd als een opeenvolging van subassemblages die de operator vanaf de huidige assemblagetoestand wenst uit te voeren om de finale toestand te bereiken. Deze wordt geschat door middel van het observeren van de gegrepen onderdelen en gereedschappen. In iedere assemblagetoestand zijn er namelijk maar enkele handelingen mogelijk en deze zijn herkenbaar aan de hand van hun benodigde onderdelen.

Verder gebruikt de planner ook een beloningsfunctie. Hierbij ontvangt de robot een bepaalde beloning bij iedere uitgevoerde actie. De planner probeert continu de actie uit te voeren met de hoogst mogelijke beloning door daarnaast ook rekening te houden met eventuele beloningen bij toekomstige acties.

De assemblagetoestanden kunnen gevisualiseerd worden in een assemblagetoestandsgraaf zoals zichtbaar is in Figuur 27. Hierbij stellen de cirkels de assemblagetoestanden voor en de pijlen de assemblagehandelingen die nodig zijn om de aangewezen toestanden te bereiken. Ook is hier één mogelijke operatorintentie gevisualiseerd in het rood.



Figuur 27: Assemblagetoestandsgraaf Bourjault's balpen

In onderstaande Tabel 5 staat voor iedere assemblagetoestand opgelijst uit welke sub-assemblages deze situatie bestaat, om een beter beeld te geven van de assemblages.

Tabel 5: Assemblagetoestanden

Assemblagetoestand	Subassemblages
1	Geen
2	[schacht, stop]
3	[punt, cartridge]
4	[punt, schacht]
5	[punt, cartridge] en [schacht, stop]
6	[punt, cartridge, inkt]
7	[punt, cartridge, schacht]
8	[dop, punt, schacht]
9	[schacht, stop] en [punt, cartridge, inkt]
10	[punt, cartridge, inkt, schacht]
11	[dop, punt, schacht]
12	[punt, cartridge, inkt, schacht, stop]
13	[dop, punt, cartridge, inkt, schacht]
14	[dop, punt, cartridge, inkt, schacht, stop]

5.2.2 Actiedetectie met behulp van de HIM

Voor het monitoren van de onderdelen gebruikt het framework de HIM via de eerder ontwikkelde ROS-driver. Hiervoor moeten de detecties echter eerst geconfigureerd zijn. Dit gebeurt in de software van de HIM zelf. Hiervoor is het nodig om een sensor te kiezen om de detecties mee aan te maken. Er is namelijk de keuze tussen: de dieptesensor, de infraroodcamera en kleurencamera. Daarnaast is er nog de optie “automatisch”, deze test alle opties en neemt hieruit de beste optie.

Initieel is er gekozen om iedere detectie te maken met behulp van de automatische optie. Tijdens het testen van deze detecties bleek dat sommige detecties voor onbekende reden inconsistent waren. Deze onderdelen waren namelijk de dop en de stop. Hierdoor is gekozen om reflectieve tape te plaatsen onder deze onderdelen en dan manueel te kiezen voor de infraroodcamera. Dit zorgt voor een duidelijk contrast wanneer de tape zichtbaar is. Hiermee zijn er geen problemen meer ondervonden met de detecties van deze onderdelen.

5.2.3 Samenwerking met de UR5-cobot

Ten slotte is het ook nodig om bij deze demonstrator de UR5-cobot aan te sturen. Hiervoor is gebruik gemaakt van de eerder beschreven code uit hoofdstuk 4. Met deze code is het mogelijk om de robot naar een aangegeven positie te bewegen. Deze posities zijn echter geen punten in een assenstelsel maar hoeken van iedere joint van deze robotarm. Om deze te bepalen is er kinaesthetic teaching gebruikt. Hierbij kan een knop ingedrukt worden op het gebruikspaneel van de cobot die het mogelijk maakt de robot in een gewenste positie te begeleiden. Wanneer de gewenste positie van de cobot bereikt is, kan de knop losgelaten worden en behoudt de cobot zijn positie. Vervolgens zijn de rotaties van de verkregen positie zichtbaar op het gebruikerspaneel en kan de gebruiker deze noteren in het programma.

Door deze procedure te herhalen voor alle posities van een volledige assemblage, zijn alle rotaties hardcoded beschikbaar in het programma. De server kan deze functie dan oproepen om een volledige assemblage te doorlopen.

5.3 Conclusie

Dit hoofdstuk bespreekt de volledige opstelling van de demonstratie, de gebruikte assemblage en instellingen van de HIM en cobot. Voor de opstelling is er gekozen voor de beschikbare opstelling op ACRO met enkele aanpassingen. Aangezien de cobot sommige assemblagehandelingen niet zonder tussenpositie kan uitvoeren is het nodig om twee houders te ontwikkelen. Deze zijn enerzijds gebruikt voor de onderdelen te stockeren en anderzijds voor enkele onderdelen en subassemblages in een tussenpositie te kunnen plaatsen.

Verder is er uitgelegd hoe de robotactieplanner zijn beslissingen maakt. Wanneer er namelijk veranderingen worden gedetecteerd bij de detecties van een object, weet de robotactieplanner welke handeling de operator heeft uitgevoerd. Aan de hand van deze handelingen en de vooraf gekende informatie van de assemblage maakt de gebruikte robotactieplanner in iedere situatie de juiste beslissing. Dit gebeurt snel genoeg en zorgt voor een vloeiende en veilige samenwerking tussen de operator en de robot.

De gebruikte assemblage is een passende representatie van een product dat geassembleerd zou kunnen worden door een industrieel werkstation. Doordat deze ROS action server de communicatie is tussen de robotactieplanner, de robotica en sensoren is het mogelijk om eender welke andere robotica te gebruiken zolang deze een ROS-driver heeft. Een voorbeeld van deze modulariteit is de mogelijkheid om meerdere robotarmen te implementeren aangezien dit geen veranderingen vereist aan de basis van dit framework. Deze aansturingen zouden enkel een extra onderdeel vormen onder de opgestelde assemblage functies.

Daarnaast zijn in de huidige versie van het framework, alle assemblages hardcoded geïmplementeerd. In de huidige softwareversie van de HIM is het namelijk nog niet mogelijk om objecten op een werkplaats te herkennen. Er kunnen enkel vaste locaties gecontroleerd worden op de aanwezigheid van objecten. Arkite is momenteel echter bezig met het implementeren van deze functionaliteit. Dit zou ervoor zorgen dat objecten niet op een vaste locatie moeten staan en zou het volledige systeem veel gebruiksvriendelijker maken.

Als laatste is er een video beschikbaar op het Youtube-kanaal van ACRO[6] waarop deze demonstrator zichtbaar is in actie.

[6] Video van demonstratie: https://www.youtube.com/watch?v=jYPnvDPpmzg

Hoofdstuk 6: Conclusie

Dit hoofdstuk blikt terug op de masterproef en vormt de nodige conclusies. Dit gebeurt in eerste instantie aan de hand van een analyse van de onderzoeksvraag en doelstellingen en in welke mate deze vervolledigd zijn. Tot slot volgt een sectie waarin de mogelijkheden voor toekomstig onderzoek worden besproken.

6.1 Behalen van de vooropgestelde doelstellingen

Het doel van de masterproef was in de eerste plaats de extensie naar en de integratie van de HIM-technologie in het domein van mens-robotsamenwerking. Hierop kan dan verder worden gebouwd door deze technologie te integreren in een framework, om er zo een demonstrator van de toepassingsmogelijkheden mee op te zetten.

Uit het literatuuronderzoek bleek ROS al snel de meest logische keuze als toegangspoort naar het domein van mens-robotsamenwerking. De veelzijdigheid van dit besturingssysteem biedt de mogelijkheid om de functionaliteit van de HIM op een robuuste en betrouwbare manier te integreren. Ook bezit ROS de nodige ontwikkelingstools om het volledige proces te controleren en te visualiseren. Tot slot bepaalt het feit dat ROS open source is dat iedereen er gratis mee kan werken en dat gepatenteerde code vermeden wordt. Dit heeft tot gevolg dat er naast een grote hoeveelheid informatie op online forums ook besturingsprogramma's voor diverse robotica ter onze beschikking staan. Deze vereenvoudigen het tweede deel van deze masterproef, het opzetten van een framework voor mens-robotsamenwerking.

De ontwikkeling van de ROS-driver voor de HIM werd opgesplitst in twee delen. Ten eerste wisselt de driver informatie uit met de HIM. Hiervoor maakt de ROS-driver gebruik van de aanwezige web-API, om zo de mogelijke HTTP-requests uit te voeren op de HIM. De communicatie met de web-API gebeurt in de praktijk met behulp van de externe softwarebibliotheek 'cpp-httplib'. Deze is namelijk eenvoudig in gebruik en bezit de mogelijkheid alle vereiste Get-, Post-, Patch-, Put-, en Delete- requests uit te voeren. Daarnaast communiceert de driver deze informatie binnen ROS met behulp van topics en messages.

De ontwikkelde ROS-driver werd zo in staat gesteld alle requests uit te voeren bij de web-API van de HIM, op uitzondering van enkelen. Zo werd er gekozen om het plaatsen, vervangen en tonen van afbeeldingen in een bepaald project weg te laten, aangezien dit via ROS onnodig gecompliceerd

zou zijn en voorlopig niet vereist was voor de gebruikte use-case. Van de overige 37 requests ondervinden twee requests ('AddMaterialToContainer' en 'GetUnitVariables') problemen bij het uitvoeren. De oorsprong van deze problemen ligt echter bij de huidige versie van de server van de web-API, welke deze requests momenteel niet herkent. Arkite is hiervan op de hoogte gesteld en geeft aan de nodige aanpassingen te zullen doorvoeren in een toekomstige software-update. De verschillende ROS-nodes zijn geschreven met oog voor mogelijke foutmeldingen en geven duidelijke berichten aan de gebruiker over de actuele toestand. Dit zorgt voor een uitgebreide, maar gebruiksvriendelijke ROS-driver voor de Human Interface Mate.

De volgende stap in dit onderzoek bestond uit het integreren van de ontwikkelde ROS-driver in een framework voor mens-robotsamenwerking. Dit framework zorgt voor een modulaire structuur waarin ieder onderdeel, noodzakelijk voor mens-robotsamenwerking, met elkaar verbonden is. Het gebruik van de HIM via de ontwikkelde ROS-driver (om de huidige situatie van de assemblage te weten te komen) bleek meteen een eerste toepassing waarvoor deze driver in realiteit gebruikt kon worden.

Zoals eerder vermeld is ROS een opensourcebesturingssysteem. Dit bood de mogelijkheid de UR5-cobot en Robotiq 2F-85 grijper aan te sturen door middel van hun open source ROS-drivers. De aansturing van de verschillende onderdelen van het framework gebeurt via een ROS action server en client. Hierbij stuur de client via een action message de nodige informatie naar de server, om zo één van de drie mogelijke acties uit te voeren: observeren van de operator, assembleren van een product of wachten.

Finaal werd er een demonstrator ontwikkeld om de ROS-driver voor de HIM te valideren in het framework voor mens-robotsamenwerking. Hierbij helpt het eerder beschreven framework, met behulp van de UR5-cobot en Robotiq-grijper, een persoon bij het assembleren van een zesdelig product (i.e. Bourjault's balpen) in een assemblagevolgorde naar keuze.

Voor de opstelling is er gekozen voor de beschikbare opstelling op ACRO met enkele aanpassingen. Aangezien de cobot sommige assemblagehandelingen niet zonder tussenpositie kan uitvoeren was het nodig om twee houders te ontwikkelen. Deze zijn enerzijds gebruikt voor de onderdelen te stockeren en anderzijds voor enkele onderdelen en subassemblages in een tussenpositie te kunnen plaatsen. De gebruikte assemblage is een passende representatie van een product dat geassembleerd zou kunnen worden door een industrieel werkstation. In de huidige versie van het framework zijn alle assemblages hardcoded geïmplementeerd. De huidige softwareversie van de HIM is namelijk nog niet in staat om objecten op een werkplaats te herkennen en te lokaliseren. Er kunnen enkel vaste locaties gecontroleerd worden op de aanwezigheid van objecten. Uit de demonstratie is gebleken dat ieder onderdeel van het opgestelde framework naar behoren functioneerde, wat resulteerde in een vloeiende en veilige samenwerking tussen de operator en de robot. Tijdens de demonstratie kon de operator de assemblage van de balpen van Bourjault vervolledigen via een assemblagevolgorde naar keuze. Een link naar de video van deze demonstratie is terug te vinden in paragraaf 5.3.

6.2 Mogelijkheden voor toekomstig onderzoek

Wat de ROS-driver voor de HIM betreft is deze inzake functionaliteit bijna volledig up-to-date met de huidige versie van de web-API. Enkel het plaatsen, vervangen en tonen van afbeeldingen ontbreekt hier. Wat de optimalisatie van de code betreft is er echter nog wat mogelijk. Zo worden de statuscodes momenteel gecontroleerd door deze te vergelijken met een 'string' datastructuur. In werkelijkheid bezitten ze een eigen datastructuur waardoor bepaalde softwarebibliotheken het gebruik robuuster en eenvoudiger kunnen maken.

Daarnaast is het, doordat deze ROS action server de communicatie is tussen de robotactieplanner, de robotica en sensoren, mogelijk om eender welke andere robotica te gebruiken zolang deze een ROS-driver heeft. Een voorbeeld van deze modulariteit is de mogelijkheid om meerdere robotarmen te implementeren aangezien dit geen veranderingen vereist aan de basis van dit framework. Deze aansturingen zouden enkel een extra onderdeel vormen onder de opgestelde assemblage functies.

Verder kunnen de objecten momenteel enkel op vaste locaties gecontroleerd worden door de HIM. Arkite is momenteel echter bezig met het implementeren van objectherkenning om detecties dynamischer te maken. Deze informatie kan doorgegeven worden aan de robot voor het grijpen van de assemblageonderdelen. Dit zou ervoor zorgen dat objecten niet op een vaste locatie moeten staan en zou het volledige systeem veel gebruiksvriendelijker maken.

Tot slot maakt de ontwikkelde driver het mogelijk de HIM-detecties te gebruiken in ROS. Waar voorheen ArUco-markers of andere alternatieven nodig waren om visuele detecties uit te voeren, biedt de HIM een robuust alternatief aan. In tegenstelling tot ArUco-markers is de HIM in staat de aanwezigheid van verschillende soorten gereedschap, objecten en containers te herkennen door middel van aangeleerde diepte- en/of IR-beelden. Daarbovenop biedt de HIM de mogelijkheid om extra boodschappen en eventuele handleidingen te projecteren via augmented reality.

Referentielijst

[1] Arkite NV, "Arkite - Productvideo 2020," *Youtube*, Jul. 10, 2020. https://www.youtube.com/watch?v=-MKZuQFItNA (accessed May 29, 2021).

[2] I. Maurtua, A. Ibarguren, J. Kildal, L. Susperregi, and B. Sierra, "Human-robot collaboration in industrial applications: Safety, interaction and trust," *International Journal of Advanced Robotic Systems*, vol. 14, no. 4, pp. 1–10, 2017, doi: 10.1177/1729881417716010.

[3] Bourjault A, "Methodology of assembly automation: a new approach," *Abstracts of 2nd International Conference on Robotics and Factories of the Future*, pp. 37–45, 1987.

[4] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus Time-of-Flight Kinect," *Computer vision and image understanding*, vol. 139, pp. 1–20, 2015.

[5] Arkite, "Human Interface Mate," *Arkite*, Mar. 2020. https://arkite.com/nl/human-interface-mate (accessed Nov. 20, 2020).

[6] The Sempre Group, "Arkite HIM System," Oct. 14, 2019. https://www.youtube.com/watch?v=_5ivIHBMtz4 (accessed Dec. 14, 2020).

[7] I. Saito, "ROS/Patterns - ROS Wiki," May 2021. http://wiki.ros.org/ROS/Patterns (accessed May 28, 2021).

[8] A. Koubaa, *Robot Operating System (ROS) - The Complete Reference*, vol. 4. 2020.

[9] J. Lentin, *Robot Operating System for Absolute Beginners*. Apress, Berkeley, CA, 2018.

[10] A. Sears-Collins, "How to Create a Service in ROS Noetic Automatic Addison," May 2021. https://automaticaddison.com/how-to-create-a-service-in-ros-noetic (accessed May 28, 2021).

[11] B. Venneker, "Robot Lijnvolger," May 2021. http://www.bartvenneker.nl/Arduino/index.php?art=0022 (accessed May 29, 2021).

[12] "What is a ROS Node?," *Robotics Back-End*, Nov. 2020. https://roboticsbackend.com/what-is-a-ros-node (accessed May 25, 2021).

[13] J. Lentin, “Robot Kinematics in a Nutshell,” Apr. 2020. https://robocademy.com/2020/04/21/robot-kinematics-in-a-nutshell (accessed Feb. 23, 2021).

[14] M. Arruda, “My Robotic Manipulator: Basic URDF & RViz,” *Construct*, May 2019. https://www.theconstructsim.com/ros-projects-robotic-manipulator-part-1-basic-urdf-rviz (accessed Nov. 30, 2020).

[15] ROS.org, “XML Robot Description Format (URDF),” Nov. 2020. http://ros.fei.edu.br/roswiki/urdf(2f)XML(2f)model.html#CA-890e8e177d59c485933caec80294fd3b407057ca_2 (accessed Nov. 30, 2020).

[16] J. Sumon, “What is MoveIt! ROS? A Jump-Start guide to MoveIt! - Jonathan Sumon - Medium,” *Medium*, May 2019. https://medium.com/@jonathansumon/what-is-moveit-ros-a-jump-start-guide-to-moveit-873e0102d7e4 (accessed Dec. 01, 2020).

[17] S. Chitta, I. Sucan, and S. Cousins, “MoveIt!,” *IEEE Robotics and Automation Magazine*, vol. 19, no. 1. pp. 18–19, Mar. 2012, doi: 10.1109/MRA.2011.2181749.

[18] Spotify, “Web API Spotify for Developers,” Aug. 2020. https://developer.spotify.com/documentation/web-api (accessed Oct. 20, 2020).

[19] R. Sangubotle, “Web Services or Web API,” *TA Digital Labs*, Nov. 2018. https://labs.tadigital.com/index.php/2018/10/29/web-services-or-web-api (accessed Oct. 23, 2020).

[20] Arkite, “HIM Server Communication (HIM 2019.3),” Nov. 2020. https://app.swaggerhub.com/apis/Arkite/HIM.SERVER.API/1.0.0-oas3#/info (accessed Feb. 12, 2021).